Integrity Constraints and Security

Juliana Freire

Some slides adapted from L. Delcambre, R. Ramakrishnan, G. Lindstrom, J. Ullman and Silberschatz, Korth and Sudarshan
Why Integrity Constraints?

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.

Why Security?

- To guard against unauthorized, malicious, access
Integrity Constraints (ICs)

• **IC**: condition that must be true for *any* instance of the database
  – ICs are specified when schema is defined.
  – ICs are checked when relations are modified.

• A *legal* instance of a relation is one that satisfies all specified ICs.
  – DBMS should not allow illegal instances.

• If the DBMS checks ICs, stored data is more faithful to real-world meaning.
  – Avoids data entry errors, too!
Kinds of Constraints

- Keys
- Foreign-key, or referential-integrity
- Value-based constraints
  - Constrain values of a particular attribute.
- Tuple-based constraints
  - Relationship among components.
- Assertions: any SQL boolean expression
Primary Key Constraints

• A set of fields is a *key* for a relation if :
  1. No two distinct tuples can have same values in all key fields, and
  2. This is not true for any subset of the key.
     – Part 2 false? A *superkey*.
     – If there’s more than 1 key for a relation, one of the keys is chosen (by DBA) to be the *primary key*.

• E.g., *sid* is a key for Students.
  The set \{sid, gpa\} is a superkey.
Primary and Candidate Keys in SQL

• Possibly many candidate keys (specified using UNIQUE), one of which is chosen as the primary key.

• “For a given student and course, there is a single grade.”

CREATE TABLE Enrolled
    (sid CHAR(20),
    cid CHAR(20),
    grade CHAR(2),
    PRIMARY KEY (sid,cid) )
Primary and Candidate Keys in SQL

- “For a given student and course, there is a single grade.”

- “Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade.”

Used carelessly, an IC can prevent the storage of database instances that arise in practice!
Primary and Candidate Keys in SQL

Which among the following tuples respect the specified constraints?

- (s1,c1,g1)
- (s2,c1,g1)
- (s2,c2,g2)
- (s3,c2,g3)

- “Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade.”

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARİY KEY (sid),
UNİQUE (cid, grade) )

Used carelessly, an IC can prevent the storage of database instances that arise in practice!
Foreign Keys, Referential Integrity

- **Foreign key**: Set of fields in one relation that is used to `refer` to a tuple in another relation.
  - Must correspond to primary or candidate key of the second relation.
  - *Like a `logical pointer`*.
- If all foreign key constraints are enforced, **referential integrity** is achieved, i.e., no dangling references.
- Can you name a data model w/o referential integrity?
  - **Links in HTML!**
Foreign Keys in SQL

• Only students listed in the Students relation should be allowed to enroll for courses.

CREATE TABLE Enrolled
  (sid CHAR(20), cid CHAR(20), grade CHAR(2),
   PRIMARY KEY (sid,cid),
   FOREIGN KEY (sid) REFERENCES Students )

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>
Foreign Keys in SQL

- Only students listed in the Students relation should be allowed to enroll for courses.

```
CREATE TABLE Enrolled
    (sid CHAR(20), cid CHAR(20), grade CHAR(2),
    PRIMARY KEY (sid, cid),
    FOREIGN KEY (sid) REFERENCES Students (sid))
```

<table>
<thead>
<tr>
<th>Enrolled</th>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>cid</td>
</tr>
<tr>
<td>53666</td>
<td>Carnatic101</td>
</tr>
<tr>
<td>53666</td>
<td>Reggae203</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
</tr>
</tbody>
</table>
Enforcing Referential Integrity

• Consider Students and Enrolled; \( sid \) in Enrolled is a foreign key that references Students.
• What should be done if an Enrolled tuple is inserted with a non-existent student id? (Reject it!)
• What should be done if a Students tuple is deleted? Some options:
  1. Also delete all Enrolled tuples that refer to it.
  2. Disallow deletion of a Students tuple that is referred to.
  3. Set \( sid \) in Enrolled tuples that refer to it to a default \( sid \).
  4. Set \( sid \) in Enrolled tuples that refer to it to a special value \texttt{null}, denoting \texttt{unknown} or \texttt{inapplicable}.
• Similar if primary key of Students tuple is updated.
Referential Integrity in SQL

- SQL/92 and SQL:1999 support all 4 options on deletes and updates.
  - Default is **NO ACTION** *(delete/update is rejected)*
  - **CASCADE** *(also delete all tuples that refer to deleted tuple)*
  - **SET NULL / SET DEFAULT** *(sets foreign key value of referencing tuple)*

```sql
CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid),
FOREIGN KEY (sid) REFERENCES Students
ON DELETE CASCADE
ON UPDATE SET DEFAULT
)`
Referential Integrity in SQL

- SQL/92 and SQL:1999 support all 4 options on deletes and updates.
  - Default is NO ACTION reject delete
  - CASCADE
    - Delete student → delete enrolled tuples that refer to student
    - Update student → update enrolled tuples that refer to student
  - SET NULL / SET DEFAULT
    - Sid = NULL
    - Sid = <default value>

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid),
FOREIGN KEY (sid)
REFERENCES Students
  ON DELETE CASCADE
  ON UPDATE SET DEFAULT )
**Example: Cascade**

- **Delete the (123,John,…) tuple from Students:**
  - Then delete all tuples from Enrolled that have sid=123.

- **Update the (123,John,…) tuple by changing Sid to ’1234’:**
  - Then change all Enrolled tuples with sid= ’123’ to sid= 1234.
Example: Set NULL

• Delete the (123, John, …) tuple from Students:
  – Change all tuples of Enrolled that have sid = '123' to have sid = NULL.

• Update the (123, John, …) tuple by changing 123 to 1234:
  – Same change as for deletion.
Where do ICs Come From?

- ICs are based upon the semantics of the real-world enterprise that is being described in the database relations.
- We can check a database instance to see if an IC is violated, but we can **NEVER** infer that an IC is true by looking at an instance.
  - An IC is a statement about *all possible* instances!
  - From example, we know *name* is not a key, but the assertion that *sid* is a key is given to us.
- Key and foreign key ICs are the most common; other ICs supported too.
Domain Constraints

• They test values inserted in the database, and test queries to ensure that the comparisons make sense.

• New domains can be created from existing data types
  – E.g. `create domain Dollars numeric(12, 2) NOT NULL`
  – `create domain Pounds numeric(12,2)`
  – We can assign or compare a value of type Dollars to a value of type Pounds!
  – SQL:1999 introduced distinct types
    • `create type Dollars numeric(12, 2) NOT NULL`
Attribute-Based and Tuple-Based Checks

• Put a constraint on the value of a particular attribute or set of attributes:

  CREATE TABLE Enrolled
  (sid CHAR(20) CHECK ( sid IN
   (SELECT sid FROM Students)),
  cid CHAR(20),grade CHAR(2) CHECK ( grade IN
   (‘A’,’B’...)),...)

  CREATE TABLE Enrolled
  (sid CHAR(20), cid CHAR(20),grade INT NOT NULL,...
  CHECK (sid = 12345 or grade < 4.0))

• Checked on insert or update
  – Not when tuple is deleted!

• NOTE: SQL forbids nulls in primary keys
Timing of Checks

- Attribute-based checks are performed only when a value for that attribute is inserted or updated.
  - Example: `CHECK (price <= 5.00)` checks every new price and rejects the modification (for that tuple) if the price is more than $5.
  - Example: `CHECK (sname IN (SELECT name FROM Students))` not checked if a student is deleted from Students (unlike foreign-keys).
Tuple-Based Checks

• CHECK (<condition>) may be added as a relation-schema element.
• The condition may refer to any attribute of the relation.
• Checked on insert or update only.
Example: Tuple-Based Check

- Only John can get a grade higher than 4:

```sql
CREATE TABLE Sells (  
sid INT,  
sname CHAR(20),  
grade REAL,  
CHECK (sid= 123 OR  
       grade <= 4.00)
);
```
The CHECK Clause

- Can be applied to relation and domain declarations
- Ensure attribute values satisfy given conditions
  - Relation declaration:
    ```
    CHECK (grade <= 4.0)
    ```
    Does not allow any grades greater than 4.0
  - Domain declaration:
    ```
    create domain hourly-wage numeric(5,2)
    constraint value-test check(value >= 4.00)
    ```
    The domain has a constraint that ensures that the hourly-wage is greater than 4.00
Schema-Level Constraints (Assertions) and Triggers

• Assertion: Boolean-valued SQL expression that must be true at all times
  – DBMS must *infer* whether any given modification may affect the truth of an assertion
  – Hard to implement efficiently

• Triggers: series of actions associated with certain events (e.g., insertion of a tuple into a relation)
  – Tells exactly *when* the DBMS must deal with them!
Assertions

- These are database-schema elements, like relations or views.
- Defined by:
  
  ```sql
  CREATE ASSERTION <name>
  CHECK ( <condition> );
  
  ```
- Removed by: DROP ASSERTION <name>
- Condition may refer to any relation or attribute in the database schema: Must *always* be true – any modification that causes it to be false is rejected
- Example: CREATE ASSERTION NoOverworkedStudent CHECK ( 
  
  ```sql
  NOT EXISTS ( 
    SELECT sid,COUNT(cid) AS TotCourses FROM Enrolled 
    GROUP BY sid 
    HAVING TotCourses > 4 
  )
  
  ```
  
  Condition must be Boolean – often aggregate
Assertion: Example

CREATE ASSERTION NoOverworkedStudent CHECK (NOT EXISTS (SELECT sid,COUNT(cid) AS TotCourses FROM Enrolled GROUP BY sid HAVING TotCourses > 4));

Condition must be Boolean
Example: Assertion

• In Students(sid,sname, address) and Professors(pid,pname, address), there cannot be more professors than students.

CREATE ASSERTION FewProfs CHECK (  
   (SELECT COUNT(*) FROM Professors)  
   <=  
   (SELECT COUNT(*) FROM Students)  
);

Timing of Assertion Checks

• In principle, we must check every assertion after every modification to any relation of the database.
• A clever system can observe that only certain changes could cause a given assertion to be violated.
  – Example: Changes to a Course (e.g., course name) do not affect FewProfs

CREATE ASSERTIONFewProfs CHECK ( 
  (SELECT COUNT(*) FROM Professors) <= 
  (SELECT COUNT(*) FROM Students)
);

• Testing the validity of assertions for every database modification may introduce substantial overheads
  – Use assertions with care!
CREATE TABLE Enrolled
(sid CHAR(20), cid CHAR(20), grade CHAR(2),
CHECK (sid = 12345 or grade < 4.0))

CREATE ASSERTION SpecialStudent CHECK (
NOT EXISTS (
SELECT sid FROM Enrolled
WHERE sid <> 12345 AND grade >= 4.0));
Assertions vs. CHECK Constraints

CREATE TABLE Enrolled
    (sid CHAR(20), cid CHAR(20), grade CHAR(2),…
     CHECK (sid = 12345 or grade < 4.0)
    )

CREATE ASSERTION SpecialStudent CHECK ( NOT EXISTS ( SELECT sid FROM Enrolled WHERE sid <> 12345 AND grade >=4.0 ));

Are these equivalent?

Difference: when constraint is checked!
CREATE TABLE Enrolled
    (sid CHAR(20), cid CHAR(20), grade CHAR(2),…
    CHECK (grade < 4.0 AND
    sid IN (SELECT sid FROM Students))
    )

What happens if a student is deleted from the student table?

Attribute-based constraints are only checked on inserts and updates to that relation, NOT on modifications to relations referenced in subqueries -- significant "hole" in constraint-checking.
Constraints in Oracle

• When declaring a foreign key constraint at the end of a table declaration *it is always necessary to put the list of referencing attributes in parentheses*

```sql
create table foo (... foreign key (<attr_list>) references (<attr_list>));
```
Constraints in Oracle (cont.)

- CHECK conditions cannot use subqueries: attribute- and tuple-based constraints can only reference the attribute or tuple that is being inserted or updated.

```sql
CREATE TABLE Enrolled
    (sid CHAR(20), cid CHAR(20), grade CHAR(2), ...
     CHECK (grade < 4.0 AND
             sid IN (SELECT sid FROM Students))
    )
```

- Oracle does not support general ASSERTIONs
The Chicken and Egg Problem

• What happens if we execute the following statements?
CREATE TABLE chicken (cID INT PRIMARY KEY, 
eID INT REFERENCES egg(eID));
CREATE TABLE egg(eID INT PRIMARY KEY, 
cID INT REFERENCES chicken(cID));

eID INT REFERENCES egg(eID))
ERROR at line 2:
ORA-00942: table or view does not exist

cID INT REFERENCES chicken(cID))
ERROR at line 2:
ORA-00942: table or view does not exist
The Chicken and Egg Problem: A Solution

- Create chicken and egg without foreign key declarations:
  CREATE TABLE chicken(cID INT PRIMARY KEY, eID INT);
  CREATE TABLE egg(eID INT PRIMARY KEY, cID INT);

- Then, we add foreign key constraints:
  ALTER TABLE chicken ADD CONSTRAINT chickenREFegg
    FOREIGN KEY (eID) REFERENCES egg(eID)
    INITIALLY DEFERRED DEFERRABLE;
  ALTER TABLE egg ADD CONSTRAINT eggREFchicken
    FOREIGN KEY (cID) REFERENCES chicken(cID)
    INITIALLY DEFERRED DEFERRABLE;
Deferring Constraint Checking

• INITIALLY DEFERRED DEFERRABLE tells Oracle to do deferred constraint checking. For example, to insert (1, 2) into chicken and (2, 1) into egg, we use:

\[
\begin{align*}
\text{INSERT INTO chicken VALUES}(1, 2); \\
\text{INSERT INTO egg VALUES}(2, 1); \\
\text{COMMIT;}
\end{align*}
\]

• DEFERRED constraints are only checked at the COMMIT point!
Dropping Inter-Dependent Tables

- Must drop the constraints first---Oracle won't allow us to drop a table that's referenced by another table.

```
ALTER TABLE egg DROP CONSTRAINT eggREFchicken;
ALTER TABLE chicken DROP CONSTRAINT chickenREFegg;
DROP TABLE egg;
DROP TABLE chicken;
```
Triggers: Motivation

- Attribute- and tuple-based checks have limited capabilities
- Assertions are sufficiently general for most constraint applications, but they are hard to implement efficiently
  - The DBMS must have real intelligence to avoid checking assertions that couldn’t possibly have been violated.
- Oracle does not support assertions!
  - http://www-db.stanford.edu/~ullman/fcdb/oracle/or-nonstandard.html#constraints
Triggers: Solution

• A trigger allows the programmer to specify *when* the check occurs and under which *conditions* a set of *actions* should be performed
  – I.e., insert, delete, update

• Also called event-condition-action rules (ECA)

• Like an assertion, a trigger has a general-purpose condition and also can perform any sequence of SQL database modifications.
  – E.g., if balance of checking account falls below 0, perform a transfer from savings into checking

• Part of SQL standard since SQL:1999
Event-Condition-Action Rules

- **Event**: typically a type of database modification, e.g., “insert on Enrolled.”
- **Condition**: Any SQL boolean-valued expression.
- **Action**: Any SQL statements.
CREATE TRIGGER KeepGradeTrigger
AFTER UPDATE OF Enrolled
REFERENCING
OLD TABLE AS OldEnrolled
NEW TABLE AS NewEnrolled
FOR EACH STATEMENT
WHEN (NewEnrolled.grade > 4.0)
UPDATE Enrolled
SET grade = OldEnrolled.grade
WHERE Enrolled.sid = OldEnrolled.sid
Options: The Event

• AFTER can be BEFORE.
• UPDATE can be DELETE or INSERT.
  – And UPDATE can be UPDATE . . . ON a particular attribute.
Options: FOR EACH ROW

- Triggers are either “row-level” or “statement-level.”
- FOR EACH ROW indicates row-level; its absence indicates statement-level.
- **Row level triggers**: execute once for each modified tuple.
- **Statement-level triggers**: execute once for a SQL statement, regardless of how many tuples are modified.
Options: REFERENCING

• INSERT statements imply a new tuple (for row-level) or new table (for statement-level).
  – The “table” is the set of inserted tuples.
• DELETE implies an old tuple or table.
• UPDATE implies both.
• Refer to these by [NEW OLD][TUPLE TABLE] AS <name>
Options: The Condition

- Any boolean-valued condition.
- Evaluated on the database as it would exist before or after the triggering event, depending on whether BEFORE or AFTER is used.
  - But always before the changes take effect.
- Access the new/old tuple/table through the names in the REFERENCING clause.
Options: The Action

• There can be more than one SQL statement in the action.
  – Surround by BEGIN . . . END if there is more than one.

• But queries make no sense in an action, so we are really limited to modifications.
Before and After Triggers

• Different purposes
  – Before: powerful form of constraint
  – After: general application logic

Before:
  – E.g., if column must be non-NULL, before trigger can generate a value for the column
  – Cannot modify database \(\rightarrow\) never activates another trigger!
  – Example: trigger that computes the starting salary and bonus of a newly hired employee using a table of starting salaries based on job code. *Advantage:* no need to embed this policy in every application program that inserts new employees
Before and After Triggers (cont.)

After:

– Executed after triggering statement and all its constraints (e.g., cascaded deletes)
– Can modify database ➔ may activate other triggers!
– Do you see any potential problem?
  • Infinite loops
  • Solution: impose a limit on the nesting level of triggers
– Example: trigger that re-computes the GPA of a student every time a new grade is added or updated in the takes table, and updates the GPA column of the student table accordingly
Another Trigger Example

```
SQL> CREATE TABLE salary_sum
  2     (total_salary integer);

Table created.

SQL> INSERT INTO salary_sum
  2     select sum(salary) from employee;

1 row created.

SQL> CREATE OR REPLACE TRIGGER insert_emp
  2  BEFORE INSERT ON employee
  3  FOR EACH ROW
  4  BEGIN
  5     UPDATE salary_sum
  6        SET total_salary = total_salary + :NEW.salary;
  7  END;
  8  /

Trigger created.
```
SQL> select * from salary_sum;

TOTAL_SALARY
-------------
  5805000

SQL> insert into person values('Fox', 'Elm', '555-1212');
1 row created.

SQL> insert into employee values('Fox', 10000, 987);
1 row created.

SQL> select * from salary_sum;

TOTAL_SALARY
-------------
  5815000
Triggers: Maintain summary data

• Using triggers: When new tuple added to Enrolled, re-compute Student.GPA

• Alternative: create materialized view for student’s GPA – DBMS takes care of view maintenance!

CREATE MATERIALIZED VIEW salary_sum AS
(SELECT SUM(salary) FROM Employee)
Triggers: Word of Caution

• Triggers should be written with care
• Errors detected at runtime cause failure of the update that set off the trigger
• You can run into infinite loops
  – E.g., an insert trigger on relation R causes a new tuple to be inserted into R
  – Databases limit the length of trigger chains
Constraints vs. Triggers

- Constraints are written in a less procedural way and give more opportunities for optimization.
- Constraints are enforced at the time of their creation for all existing data in the database.
- Constraints protect data to be placed into an invalid state by any kind of statement.
Database Security

• Protection from malicious attempts to steal or modify data.

• DB, OS, network, physical, human-level (see textbook)
  – Need all of these to protect database!

• ICs protect the integrity of data

• Authorization control complements ICs
  – Data: grant read, insert, update delete
  – Schema: grant index, resource, alteration, drop
Authorization: Data

Forms of authorization on parts of the database:

- **Read authorization** - allows reading, but not modification of data.
- **Insert authorization** - allows insertion of new data, but not modification of existing data.
- **Update authorization** - allows modification, but not deletion of data.
- **Delete authorization** - allows deletion of data.
Authorization: Schema

Forms of authorization to modify the database schema:

- **Index authorization** - allows creation and deletion of indices.
- **Resources authorization** - allows creation of new relations.
- **Alteration authorization** - allows addition or deletion of attributes in a relation.
- **Drop authorization** - allows deletion of relations.
Security Specification in SQL

• The grant statement is used to confer authorization

  grant <privilege list>
  on <relation name or view name> to <user list>

• <user list> is:
  – a user-id
  – public, which allows all valid users the privilege granted
  – A role (more on this later)

• Granting a privilege on a view does not imply granting any privileges on the underlying relations – more later

• The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).
Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users $U_1$, $U_2$, and $U_3$ **select** authorization on the `branch` relation:

  ```sql
  grant select on branch to $U_1$, $U_2$, $U_3$
  ```

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **references**: ability to declare foreign keys when creating relations.
- **usage**: In SQL-92; authorizes a user to use a specified domain
- **all privileges**: used as a short form for all the allowable privileges
Privilege To Grant Privileges

- **with grant option**: allows a user who is granted a privilege to pass the privilege on to other users.
  - Example:
    
    ```sql
    grant select on branch to U_1 with grant option
    ```
    
    gives $U_1$ the **select** privileges on branch and allows $U_1$ to grant this privilege to others
Granting of Privileges

- The passage of authorization from one user to another may be represented by an authorization graph.
- The root of the graph is the database administrator.
- The nodes of this graph are the users.
- An edge $U_i \rightarrow U_j$ indicates that user $U_i$ has granted update authorization to $U_j$. 

![Diagram]

- $DBA$ is the root node.
- $U_1$ has an edge to $U_4$.
- $U_2$ has edges to $U_3$ and $U_5$.
- $U_3$ does not have any outgoing edges.
- $U_4$ has an edge to $U_1$.
- $U_5$ has an edge to $U_2$. 

This diagram illustrates the relationships and授权传递 in the context of database security.
Authorization Grant Graph

• **Requirement**: All edges in an authorization graph must be part of some path originating with the database administrator

• If DBA revokes grant from $U_1$:
  – Grant *must be revoked* from $U_4$ since $U_1$ no longer has authorization
  – Grant *must not be revoked* from $U_5$ since $U_5$ has another authorization path from DBA through $U_2$
Revoking Privileges

Can still update!
Authorization Grant Graph

- Must prevent cycles of grants with no path from the root:
  - DBA grants authorization to U₂
  - U₂ grants authorization to U₃
  - U₃ grants authorization to U₂
  - DBA revokes authorization from U₂

- Must revoke grant U₂ to U₃ and from U₃ to U₂ since there is no path from DBA to U₂ or to U₃ anymore.
Authorization Grant Graph

No path from DBA!
Roles

• Roles permit common privileges for a class of users can be specified just once by creating a corresponding “role”
• Privileges can be granted to or revoked from roles, just like user
• Roles can be assigned to users, and even to other roles
• SQL:1999 supports roles

```sql
create role teller
create role manager

grant select on branch to teller
grant update (balance) on account to teller
grant all privileges on account to manager

grant teller to manager

grant teller to alice, bob
grant manager to avi
```
The “cs5530” Role

CREATE ROLE "CS5530"

GRANT ALTER ANY INDEX TO "CS5530"
GRANT ALTER ANY TABLE TO "CS5530"
GRANT CREATE ANY INDEX TO "CS5530"
GRANT CREATE ANY TABLE TO "CS5530"
GRANT CREATE ANY VIEW TO "CS5530"
GRANT CREATE PROCEDURE TO "CS5530"
GRANT DROP ANY TABLE TO "CS5530"
GRANT EXECUTE ANY PROCEDURE TO "CS5530"
GRANT "CONNECT" TO "CS5530"

GRANT cs5530 TO <each of you>
Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  ```sql
  revoke <privilege list>
  on <relation name or view name> from <user list> [restrict | cascade]
  ```

- Example:

  ```sql
  revoke select on branch from U_1, U_2, U_3 cascade
  ```

- Revocation of a privilege from a user may cause other users also to lose that privilege; referred to as cascading of the **revoke**.

- We can prevent cascading by specifying **restrict**:

  ```sql
  revoke select on branch from U_1, U_2, U_3 restrict
  ```

  With **restrict**, the **revoke** command fails if cascading revokes are required.
Revoking Authorization in SQL (Cont.)

• <privilege-list> may be all to revoke all privileges the revokee may hold.
• If <revokee-list> includes public all users lose the privilege except those granted it explicitly.
• If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
• All privileges that depend on the privilege being revoked are also revoked.
Authorization and Views

• Users can be given authorization on views, without being given any authorization on the relations used in the view definition.

• Ability of views to hide data serves both to simplify usage of the system and to enhance security by allowing users access only to data they need for their job.

• A combination or relational-level security and view-level security can be used to limit a user’s access to precisely the data that user needs.
View Example

• Suppose a bank clerk needs to know the names of the customers of each branch, but is not authorized to see specific loan information.
  – Approach: Deny direct access to the loan relation, but grant access to the view cust-loan, which consists only of the names of customers and the branches at which they have a loan.
  – The cust-loan view is defined in SQL as follows:

```sql
create view cust-loan as
  select branchname, customer-name
  from borrower, loan
  where borrower.loan-number = loan.loan-number
```
View Example (Cont.)

- The clerk is authorized to see the result of the query:
  
  ```
  select *
  from cust-loan
  ```

- When the query processor translates the result into a query on the actual relations in the database, we obtain a query on borrower and loan

- Authorization must be checked on the clerk’s query before query processing replaces a view by the definition of the view
Authorization on Views

• Creation of (virtual) view does not require **resources** authorization since no real relation is being created

• The creator of a view gets only those privileges that provide no additional authorization beyond that he already had
  
  – E.g. creator cannot get write access to view if she does not have write access to the underlying relation!
  
  – E.g. if creator of view `cust-loan` had only **read** authorization on `borrower` and `loan`, he gets only **read** authorization on `cust-loan`
Limitations of SQL Authorization

• SQL does not support authorization at a tuple level
  – E.g. we cannot restrict students to see only (the tuples storing) their own grades

• With the growth in Web access to databases, database accesses come primarily from application servers.
  – End users don't have database user ids, they are all mapped to the same database user id
  – All end-users of an application (such as a web application) may be mapped to a single database user

• The task of authorization in above cases falls on the application program, with no support from SQL
  – Benefit: fine grained authorizations, such as to individual tuples, can be implemented by the application.
  – Drawback: Authorization must be done in application code, and may be dispersed all over an application
  – Checking for absence of authorization loopholes becomes very difficult since it requires reading large amounts of application code