Views & Authorization

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Simplifying Query Syntax
2. Views
3. Providing View Semantics
4. Database Authorization

Some material from lectures created by Professor Greiner (used with permission)

Context

- Our SQL skill-set is increasing rapidly ☺
  - "advanced SQL" + JOIN
- Today: introduce techniques to simplify table use
  - Syntactically: the WITH clause and temporary tables
  - Semantically: views

Simplifying Query Syntax

WITH Clause: Motivation

- The WITH clause allows you to define a table that will only be used in your current query
- Motivation: simplify syntax → more understandable, easier to maintain code

Implement: *"For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000."*

SQL implementation from a previous lecture

```
SELECT Dno, COUNT (*)
  FROM EMPLOYEE
  WHERE Salary>40000 AND Dno
      IN
  (SELECT Dno
    FROM EMPLOYEE
    GROUP BY Dno
    HAVING COUNT (*) > 5)
  GROUP BY Dno;
```

SQL implementation using WITH

```
WITH BIGDEPTS (Dno) AS
  (SELECT        Dno
    FROM         EMPLOYEE
    GROUP BY     Dno
    HAVING       COUNT (*) > 5)
SELECT Dno, COUNT (*)
FROM EMPLOYEE
WHERE Salary>40000 AND Dno IN
    BIGDEPTS
GROUP BY Dno;
```

A More Complicated Example

I don't want to "over-hype" the benefits of WITH: all it does is enable you to break down complicated queries into simpler parts

```sql
WITH
   regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
  ),
  top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM
        regional_sales)
  )
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

Temporary Tables

- A temporary table is a short-lived table that exists only for the duration of a database session
- So: lifetime extends beyond the scope of a WITH clause
  - But: DBMS ensures that all temporary tables are dropped at the end of a session or transaction
- As with the WITH clause, the benefit exists *iff* its usage will simplify your code syntax

```
1  CREATE TEMPORARY TABLE temp1 AS
2    SELECT dataid, register_type, timestamp_localtime,
       read_value_avg
3      FROM rawdata.egauge
4      WHERE register_type LIKE '%gen%'
5      ORDER BY dataid, timestamp_localtime;
```

How to simplify complex queries like this?

```
SELECT crn
FROM (SELECT crn, Avg(rating) AS avg_rating1
        FROM Enrollment
        GROUP BY crn
       ) AS Average1
WHERE avg_rating1 = (SELECT Max(avg_rating2)
                     FROM (SELECT Avg(rating) AS avg_rating2
                             FROM Enrollment
                             GROUP BY crn
                            ) AS Average2
                    );
```

Simplifying **one** query – WITH

Common Table Expression (CTE)

```
WITH Average AS
        (SELECT crn, Avg(rating) AS avg_rating
         FROM Enrollment
         GROUP BY crn)
SELECT crn
FROM Average
WHERE avg_rating = (SELECT Max(avg_rating)
                    FROM Average);
```

```
WITH Table1 AS (SELECT …),
        Table2 AS (SELECT …),
        Table3 AS (SELECT …)
SELECT ...;
```

Preferred syntax

```
WITH Average (crn, avg_rating) AS
        (SELECT crn, Avg(rating)
         FROM Enrollment
         GROUP BY crn)
SELECT crn
FROM Average
WHERE avg_rating = (SELECT Max(avg_rating)
                    FROM Average);
```

Simplifying **multiple** queries – temp tables

Automatically dropped at end of session.

```
CREATE TEMPORARY TABLE MyTable (
   …);
```

```
SELECT INTO TEMPORARY TABLE MyTable
…;
```

Adapting software engineering lessons

	SQL	Typical PLs
Basic unit of computation	Queries	Functions/methods
Naming subcomputation	WITH/temp tables	“Helper” functions

Name subcomputations when …

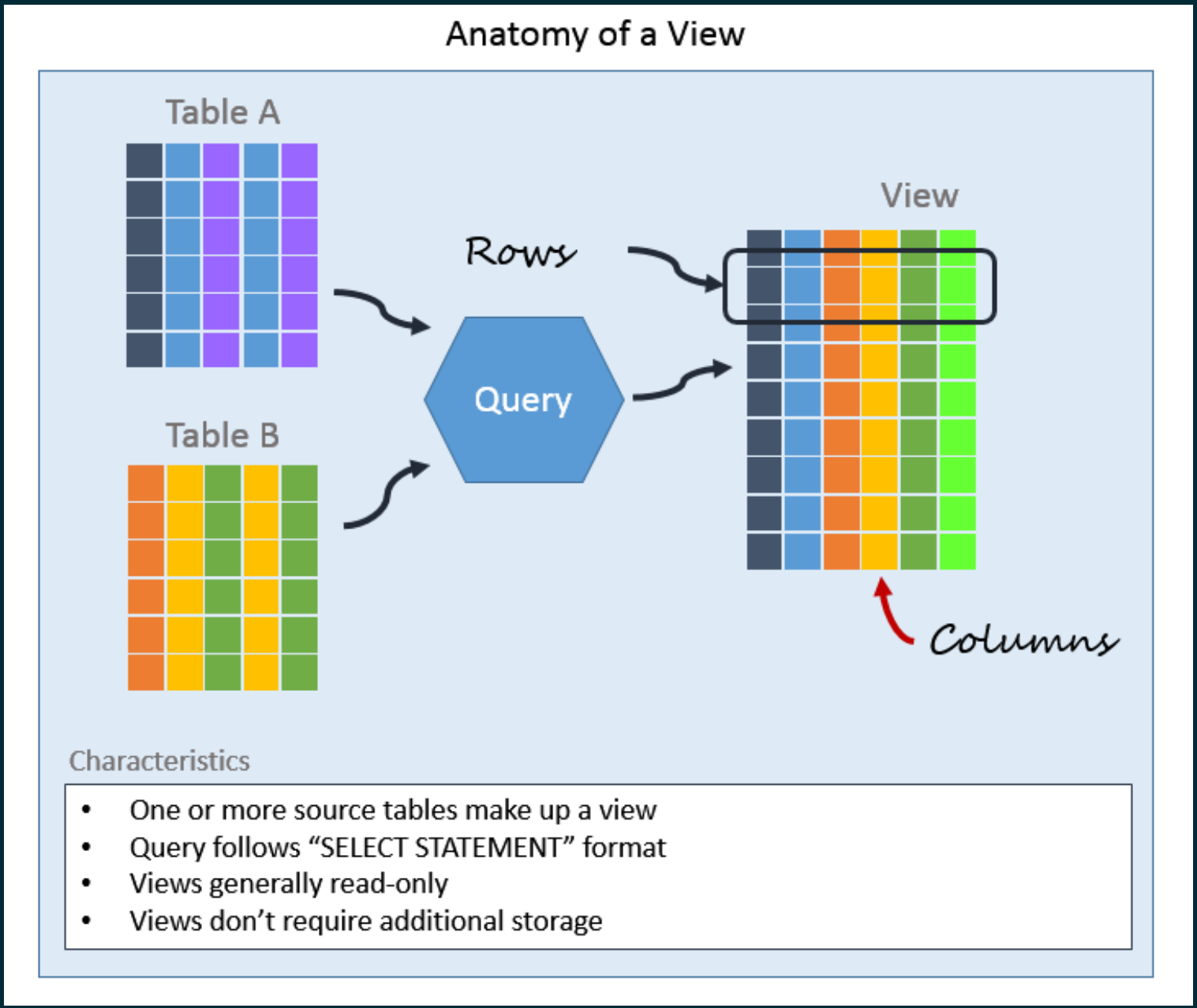
- Writing code – decomposing problem
- Rewriting code – refactoring code

Views

Introduction

- Previous lectures on SQL & database tables have tacitly assumed that “tables are real”
  - Meaning: your query accesses actual tuples, that are stored in actual relations that are stored in an actual database ☺
- Concept of views shakes that up a little
  - Views are “virtual relations”
  - Derived from “actual relations”
  - Many views can be derived from a single relation
  - Key idea: from the client viewpoint, there is not that much of a distinction between views and relations
- Ideally, once defined, a “view” should be semantically identical to a “real table”
  - “Ideally” . . . ☺

What Is A View?



View Creation: Some Examples

View instructor data without their salary

```
1       CREATE VIEW faculty
2          AS SELECT ID, name, dept_name FROM instructor
```

Accountant view: just department salary totals

```
1  CREATE VIEW departments_total_salary(dept_name, total_salary
       )
2     AS
3        SELECT dept_name, sum (salary) FROM instructor
4        GROUP by dept_name
```

Using Views

Note: it seems that, once defined, you can treat a view just like an “actual” table

Creation:

```
1     CREATE VIEW faculty AS
2        SELECT ID, name, dept_name FROM instructor
```

Use:

```
1     SELECT name FROM faculty
2     WHERE dept_name = 'Biology'
```

Defining a View From Another View

Given the above semantics, it shouldn't surprise you that a view definition can be based on another view definition

```
CREATE VIEW physics_fall_2009 AS
   SELECT course.course_id, sec_id, building, room_number
      FROM course, section
      WHERE course.course_id = section.course_id
            AND course.dept_name = 'Physics'
            AND section.semester = 'Fall'
            AND section.year = '2009'
```

```
CREATE view physics_fall_2009_watson AS
    SELECT course_id, room_number
       FROM physics_fall_2009
       WHERE building= 'Watson'
```

Benefits Of Views

- Some of the benefits are similar to the benefits we've discussed for the WITH clause and temporary tables: simplify your code!
- Views make it possible to
  - Enable different users to focus on different sets of specific data
    - Each user is presented only with the subset of attributes that's relevant to them
  - Simplify your query syntax!
    - Transform a complex query into a view
    - No need to ever refer to that complex query again ☺
    - Lifetime persists beyond original database session: until you explicitly DROP the view
  - Provide an interface layer between queries & tables
    - Allows table definitions to change
    - Or create views for backward compatibility when tables have changed

Additional Benefit: Security (I)

- Views are more than a "syntactic" construct for your code
  - When coupled with DBMS security mechanisms, views enable granular protection of the underlying relation
  - Beyond the benefits of "only presenting relevant data" (previous slide)
- Key scenarios: some users should not be able to access parts of the overall logical model
  - Example: person preparing course catalog needs to know an instructor's name and department, but not instructor's salary
  - Their "view" should allow the equivalent of (only) access to

```
1  SELECT ID, name, dept_name FROM instructor
```

  - The view is a mechanism to hide certain parts of the model from certain users
    - But: are we referring to specific users? Specific "classes" of users?
    - Stay tuned ...

Additional Benefit: Security (II)

```
1  CREATE VIEW DEPT5EMP AS
2  SELECT *
3    FROM EMPLOYEE
4    WHERE Dno = 5;
```

- Previous slide shows how a view can restrict a user's access to specific attributes
- The code above shows how a view can restrict a user's access to specific rows
  - In this example: user is only allowed to see data associated with *department #5*

Providing View Semantics

Semantics of Views

- It is tempting, but wrong, to think of a view as "the result set of a stored query on the data"
  - That would imply that view semantics are (only) a "point in time" snapshot
- Rather: a view is equivalent to its source query
  - The view is dynamic: it changes in sync with the state of its defining relations
- The responsibility for keeping a view "up-to-date" is placed on the DBMS
- This suggests the following strategy: implement a view as a stored SQL query
  - The DBMS recomputes query results whenever a user references the view

Implementation Strategy #1

- Computation "on demand": DBMS does not store any view artifacts (beyond the original definition)
- When client query refers to a view, DBMS modifies that query into the equivalent query on underlying base tables
- Note (yet again) the benefits of an elegant relational model!
  - Queries are applied to relations ("tables")
  - Queries return relations
  - So: a view is a query result that is transparently substitutable into another query
  - (Even better than a C macro) ☺

Only one problem with this implementation strategy: it's inefficient for views defined via complex queries that are time-consuming to execute

Implementation Strategy #2

- Because databases place great weight on good performance, they devised an alternate "view implementation" strategy: view materialization
- Key idea: create a "physical" temporary view table when the view is first queried
- Keep that table "materialized" on the assumption that other queries on the view will follow
  - Advantage: no need to (expensively) recompute from scratch when processing the second query
  - Disadvantage: we've just compromised the elegance of the view concept ☹
- Q: beyond the compromised elegance, can you spot a serious problem with this implementation strategy?
- A: the materialized view can become stale
- Solution: View maintenance
  - The task of updating the view whenever the underlying relations are updated

View Materialization: Dealing With Staleness

- View materialization requires a strategy for automatically updating the view table when the base tables are updated
- DBMS already has computed the initial "view" version ...
- Now it needs to incrementally update the view as necessary
  - Requires the DBMS to determine what new tuples must be inserted, deleted, or modified in a materialized view table as CUD operations take place on the defining tables
- Key point: a materialized view is essentially a cache
- We therefore have the classic tradeoff between
  - Eager approach: update a view as soon as the base tables are changed ...
  - Lazy approach: update the view when needed
- Can also use a periodic update strategy: only update the view periodically
  - Pay the price that a view query may get results that are not up-to-date
  - Tradeoff "staleness" against computational expense (and time)

Updating a View

- Q: should it be possible to update a view tuple (or insert a view tuple)?
- Naive answer: "sure, why not?" ...
  - Simply apply the operation to the underlying relation(s)
- More sophisticated answer: "probably not!" ... ☹
  - Although the mapping from $R_i \rightarrow View$ is completely defined ...
  - In most general form, multiple reverse mappings from $V \rightarrow R_i$ are impossible
- Meta-comment: perhaps we're getting trapped by the elegance of the view concept into working too hard to maintain the semantics that a "view can be substituted into any SQL statement"

View Updates: Simple Cases ("Sort Of") Work

```
CREATE VIEW faculty AS
   SELECT ID, name, dept_name FROM instructor
```

- Now: add a new tuple to faculty view
- Note: the *faculty* view has the same schema as *instructor* relation
  - Except that it's missing the *salary* attribute
- So: fairly easy to "push down" the view operation to underlying relation
- Translate

```
INSERT INTO faculty
VALUES ('30765', 'Green', 'Music');
```

- Into

```
INSERT into instructor
VALUES ('30765', 'Green', 'Music', null);
```

- But: does the client intend these semantics?
  - Note: rejecting the operation also violates the client's expectations of "correct behavior"

View Updates: More Complicated Cases Don't Work (I)

Consider this view definition:

```
1 CREATE VIEW instructor_info AS
2    SELECT ID, name, building
3      FROM instructor, department
4      WHERE instructor.dept_name= department.dept_name
```

Now consider this update:

```
1 INSERT INTO instructor_info VALUES ('69987', 'White', '
    Taylor');
```

- If multiple departments are located in 'Taylor', which department gets the insert?
- What if no department is located in 'Taylor'?
- Or: what if no instructor with an ID of '69987'?

View Updates: More Complicated Cases Don't Work (II)

- Simply inserting corresponding NULL values into the two defining tables doesn't solve our problem!
  - (Besides the fact that it's an ugly "hack" ☺)
  - The instructor_info view is still not going to manifest the new tuple
- We certainly can't expect clients to understand why the update doesn't behave as she expects
  - After all: a major motivation for views is to hide the base table definitions from clients!

Bottom Line: Major Restrictions on Updatable Views

- Most SQL implementations allow updates only on simple views
- Defined as:
  - The **from** clause refers to only one database relation
  - The **select** clause contains
    - Only attribute names of the defining relation
    - Does not have any expressions, aggregates, or DISTINCT specification
  - No GROUP BY or HAVING clauses
  - Any attribute not listed in the select clause can be set to NULL in the defining relations
    - Meaning: not a primary key attribute and no **not null** constraint
- Definitely: read the manual & "trust-but-verify" before relying on this feature!

Even With Restrictions: Problems Still Remain

Consider this view definition:

```
1  CREATE VIEW history_instructors AS
2     SELECT *
3       FROM instructor
4       WHERE dept_name= 'History';
```

Now user does an update:

```
1       INSERT INTO history_instructors
2          VALUES ('25566', 'Brown', 'Biology', 100000)
```

- This update will satisfy the above restrictions
  - We understand the semantics with respect to the instructor relation
- But: the update will not (because it cannot) be visible to the history_instructor view

Database Authorization

- By now should be clear that databases are some of the "crown jewels" of a business's assets
- The value of these assets implies that we must consider risks to these assets
- Examples
  - Loss of integrity (data that are improperly modified)
  - Loss of availability (legitimate users cannot access data)
  - Loss of confidentiality (unauthorized disclosure of data)

Database Security & The DBA

- The database administrator (or DBA) plays an important role here
- You're the DBA on your laptop ☺
- In real life™, a person (or team) is designated as the DBA for a business's database
  - In OS terminology, a DBA is the superuser or is the "system account"
- Some DBA-commands
  - Account creation
  - Privilege granting
  - Privilege revocation
  - Security level assignment
- Q: what are these privileges you speak of?
- A: (next slide . . .)

Authorization Privileges

- So far, our discussion of database CRUD operations has implicitly considered these operations to be equally important
- However: from the perspective of business processes, the implications of these operations are sufficiently different that they are associated with different privileges
  - Read: allows reading, but not modification of data
  - Insert: allows insertion of new data, but not modification of existing data
  - Update: allows modification, but not deletion of data
  - Delete: allows deletion of data
- This hierarchy reflects the fact that users may not be trusted "equally"
  - Maybe all users can be trusted to "read" data
  - But very few users can be trusted to "delete" data
- Database systems grant users any number (including zero) privileges

Access Control

- I'm using the term "access control" to refer to DBMS granting access to data after a user's account identity has previously been authenticated
- OS example: a particular user, or group of users, may only be permitted access to specific files after logging into a system
  - Even after being authenticated, the user (or group) may be denied access to all other resources

Key point: access control mechanisms are needed to allow certain users to access specified portions of the database while not enabling access to other portions of the database

Different Approaches To Access Control (I)

- Discretionary Access Control (DAC): based on granting privileges to users
  - Control is based on the (group) identity of the user
  - "Discretionary": a user with a given privilege can pass that privilege to other users
- DAC-based systems associate an access control list (ACL) with every resource (may be hierarchical for different granularities)
  - An ACL contains a list of "users and groups" to which the resource owner has permitted access
  - Different levels of access may be assigned to different users or groups
- By definition, the DAC approach allows users to pass their privileges to other users
- This "weakness" (or "flexibility") of DAC can only be restrained through use of Mandatory Access Control (MAC) (next slide)

Different Approaches To Access Control (IIa)

- The MAC approach first classifies both resources and users into security classes (levels)
- Typical examples:
  - Top secret
  - Secret
  - Confidential
  - Unclassified
- Basic idea (see Bell–LaPadula model for more detail) has two rules
  - A user is only granted read access to a resource if their security classification is at least as great as the resource's classification
  - A user is only granted write permission to a resource if their security classification is no greater than the resource's classification
    - The second property is referred to as the star property (or *-property)

Different Approaches To Access Control (IIb)

- The first MAC property is intuitive: users cannot read data that have a higher security classification than they do
- The second MAC property is less intuitive: why not let a user write data that have a lower security classification than they do?
- Key insight: without this rule, we're back to the situation where information can flow from higher → lower classifications
- Example:
  1. User with top-secret clearance makes a copy of top-secret data
  2. User classifies the data as unclassified
  3. User has just made the data visible to all users ☹
- This approach is characterized by the phrase "write up, read down"
  - Users can create content only at or above their own security level
  - Users can view content only at or below their own security level

Access Control: Back To Relational Databases

- (Returning to earlier slide) ...
- Some DBA-commands
  - Account creation: authentication
  - Privilege granting: DAC access control (also known as "authorization")
  - Privilege revocation: DAC access control (also known as "authorization")
  - Security level assignment: MAC access control
- Traditionally, relational databases only provided DAC access control
- Using SQL commands, users (beginning with the DBA), every relation is assigned an owner account
  - Typically, the account used to create the relation
- The owner can grant (pass) privileges to other users

SQL Privilege Granularity (I)

- The available privilege granularities are a superset of our classic CRUD operations
- Read the manual before usage!
- The following screen-shots are from the POSTGRESQL documentation

SELECT: Allows SELECT from any column, or specific column(s), of a table, view, materialized view, or other table-like object. Also allows use of COPY TO. This privilege is also needed to reference existing column values in UPDATE or DELETE. For sequences, this privilege also allows use of the currval function. For large objects, this privilege allows the object to be read.

INSERT: Allows INSERT of a new row into a table, view, etc. Can be granted on specific column(s), in which case only those columns may be assigned to in the INSERT command (other columns will therefore receive default values). Also allows use of COPY FROM.

UPDATE: Allows UPDATE of any column, or specific column(s), of a table, view, etc. (In practice, any nontrivial UPDATE command will require SELECT privilege as well, since it must reference table columns to determine which rows to update, and/or to compute new values for columns.) SELECT ... FOR UPDATE and SELECT ... FOR SHARE also require this privilege on at least one column, in addition to the SELECT privilege. For sequences, this privilege allows use of the nextval and setval functions. For large objects, this privilege allows writing or truncating the object.

DELETE: Allows DELETE of a row from a table, view, etc. (In practice, any nontrivial DELETE command will require SELECT privilege as well, since it must reference table columns to determine which rows to delete.)

SQL Privilege Granularity (II)

TRUNCATE: Allows TRUNCATE on a table, view, etc.

REFERENCES: Allows creation of a foreign key constraint referencing a table, or specific column(s) of a table.

TRIGGER: Allows creation of a trigger on a table, view, etc.

CREATE: For databases, allows new schemas and publications to be created within the database.

For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object and have this privilege for the containing schema.

For tablespaces, allows tables, indexes, and temporary files to be created within the tablespace, and allows databases to be created that have the tablespace as their default tablespace. (Note that revoking this privilege will not alter the placement of existing objects.)

CONNECT: Allows the grantee to connect to the database. This privilege is checked at connection startup (in addition to checking any restrictions imposed by pg_hba.conf).

TEMPORARY: Allows temporary tables to be created while using the database.

EXECUTE: Allows calling a function or procedure, including use of any operators that are implemented on top of the function. This is the only type of privilege that is applicable to functions and procedures.

USAGE: For procedural languages, allows use of the language for the creation of functions in that language. This is the only type of privilege that is applicable to procedural languages.

Access Control: To What Database Resources?

- We've focused on privileges with respect to what operations can be performed by a given user
- But as you've seen: privileges also specify the "resource" to which a given operation can be applied (i.e., we're not restricted to only controlling tuple resources)

Privilege	Abbreviation	Applicable Object Types
`SELECT`	`r` ("read")	`LARGE OBJECT`, `SEQUENCE`, `TABLE` (and table-like objects), table column
`INSERT`	`a` ("append")	`TABLE`, table column
`UPDATE`	`w` ("write")	`LARGE OBJECT`, `SEQUENCE`, `TABLE`, table column
`DELETE`	`d`	`TABLE`
`TRUNCATE`	`D`	`TABLE`
`REFERENCES`	`x`	`TABLE`, table column
`TRIGGER`	`t`	`TABLE`
`CREATE`	`C`	`DATABASE`, `SCHEMA`, `TABLESPACE`
`CONNECT`	`c`	`DATABASE`
`TEMPORARY`	`T`	`DATABASE`
`EXECUTE`	`X`	`FUNCTION`, `PROCEDURE`
`USAGE`	`U`	`DOMAIN`, `FOREIGN DATA WRAPPER`, `FOREIGN SERVER`, `LANGUAGE`, `SCHEMA`, `SEQUE`

Granting & Revoking Privileges: Syntax Examples

- Users can pass their privileges to other users via GRANT command
  - Assuming that she was originally granted privileges WITH GRANT OPTION clause

```
GRANT SELECT, UPDATE(price) ON Sells TO sally
```

Now Sally has the right to issue any query on Sells and can update the price component only

The *list of authorization ID's* can be one of

- a user id such as "Sally" in previous example
- PUBLIC keyword: grants privilege to all valid users
- A role (more on this later)

Privileges can also be revoked

```
REVOKE SELECT ON department FROM Bob, Sally
```

Authorization & Views

- Views (first part of lecture) are part of the database authorization story ☺
- We can specify privileges by using views
- Example: user *A* owns relation *R*, wants user *B* to have access to a subset of *R* attributes

```
1    CREATE VIEW faculty
2       AS SELECT ID, name, dept_name FROM instructor
3    GRANT SELECT ON faculty TO B
```

Authorization: Runtime

- When a user issues a query or update operation
  1. DBMS looks up the user's privileges for the given resource
  2. Operation is rejected if user is not authorized
- How does a user get privileges in the first place?
- Grantor of privilege must already hold the privilege on the specified item
  - Example: the database administrator
  - User that created a new relation automatically given all privileges for that relation

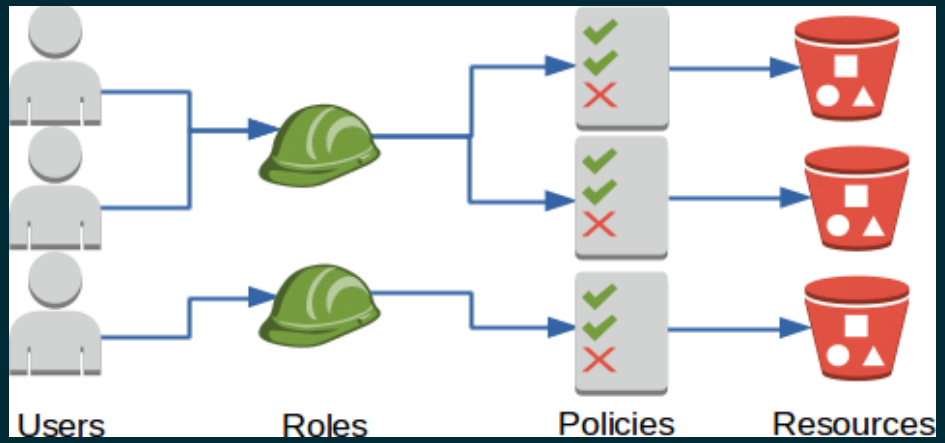
Authorization & Roles: Introduction

- Role-based access control (RBAC) adds an orthogonal semantic to both DAC & MAC
  - The concept of a role, in contrast to a user
- Note: the benefits of RBAC are not limited to database systems
  - They apply more generally to e.g., operating systems
- With user-based authorization, individual users are authenticated to the system
  - The system has previously provisioned (e.g., *via* the GRANT facility) users with appropriate privileges
  - The approach we've been describing so far ☺
- This approach is easy to understand: but does it work in real-life™?

Roles: Motivation

- Typically, more than one person can do the same job
  - Example: instructors in the same department
  - Example: secretaries in the same office
- It's painful to have to grant individual privileges when you really want role-based privileges
  - You're not interested in modeling "Bob can access the *finance* table"
  - You're interested in modeling "All members of the accounting department can access the *finance* table"
    - And (at this time) "Bob is in the accounting department"
- RBAC makes it much easier to modify the privileges associated with a role (no need to individually modify members of the set)
- RBAC makes it much easier to revoke individual privileges, especially if users have multiple roles

RBAC: Other Advantages



- Note: we continue to use user-based authorization
  - This allows the DBMS to log "who updated my salary to one million dollars?" in an audit table
- We simply add a mapping from users → roles
- We can construct hierarchical role structures to reflect real-life ontologies
  - Example: instructor → teaching assistant → student
  - Use hierarchy to simplify privilege specification: "an instructor's privileges are a superset of a teaching assistant's privileges"

Examples & Syntax

```
CREATE ROLE instructor
GRANT instructor TO Amit
```

```
-- Privileges can be granted to roles
GRANT SELECT ON takes TO instructor
```

```
-- Roles can be granted to users,
-- as well as to other roles

CREATE ROLE teaching_assistant
--  Instructor inherits all privileges of teaching_assistant
GRANT teaching_assistant TO instructor;
```

```
-- Can create hierarchical chain of roles

CREATE ROLE dean
GRANT instructor TO dean
GRANT dean TO
 Satoshi
```

Today's Lecture: Wrapping it Up

Simplifying Query Syntax

Views

Providing View Semantics

Database Authorization

Readings

You're now responsible for all of Chapter 4