CS 1555

www.cs.pitt.edu/~nlf4/cs1555/

SQL as a VDL
Three level schema review

- External Schema 1
- External Schema 2
- External Schema 3

Conceptual Schema

Physical Schema
Creating views

- **CREATE [OR REPLACE] VIEW** `viewname`
  
  **AS** `query`
  
  **[WITH CHECK OPTION]**

- **CREATE VIEW** `CS_Students` **AS**
  
  **SELECT** `*`
  
  **FROM** `Students`
  
  **WHERE** `Major = 'CS'`;

- **Views can be queried just like base tables:**
  - **SELECT** `COUNT(*)`
    
    **FROM** `CS_Students`
    
    **WHERE** `GPA > 3.5`;
So why not just create a new table?

- What about if a new student declares a CS major?
  - E.g.,
    
    ```sql
    INSERT INTO Students
    VALUES (444444, 'Tom', '4444444', 'CS', NULL);
    ```
Views should stay up to date

- Reflect the current state of the base tables they are built on
- Can be seen as almost like a macro
- May see them described as "virtual tables"
How do we accomplish this?

* Option 1: recompute the view on every query
  * *Query modification*, the query on the view is re-written to be over the base tables that are used in the view by the DBMS
    * $\text{SELECT COUNT(*)}$
      $\text{FROM CS_Students}$
      $\text{WHERE GPA > 3.5;}$
    * becomes:
      $\text{SELECT COUNT(*)}$
      $\text{FROM Students}$
      $\text{WHERE GPA > 3.5 AND Major = 'CS';}$
Modifying views

- **CREATE OR REPLACE VIEW**
  
  `CS_Students(Stud_ID, Stud_Name, Major, GPA)`

  **AS SELECT** `ID, Name, Major, GPA`

  **FROM** `Students`

  **WHERE** `Major = 'CS';`

- **We can of course get rid of views as well**
  
  - **DROP VIEW** `CS_Students;`
Option 2: view materialization

- Realize a physical copy of the view on disk
  - Advantages of this approach (vs query modification)?
  - Disadvantages?
Materializing views

- CREATE MATERIALIZED VIEW *viewname*
  
  AS *query*
  
  [WITH [NO] DATA]

- To repopulate the view, REFRESH MATERIALIZED VIEW must be used:
  - REFRESH MATERIALIZED VIEW *viewname*
    
    [WITH [NO] DATA]

- WITH DATA is default
  - View cannot be *scanned* unless it has been populated
Are views read only?

- In Postgres materialized view are...
  - But what about non-materialized?
- Should updates be propagated to the base tables defining the view?
  - *Can* they be propagated to the base tables?
    - Generally, views on a *single table* with no *aggregate functions* can be mapped to an update on the underlying base table
      - Can only update views where such a mapping is unambiguous, however
        - E.g., you have a key to the single table in the view
View update example

- **UPDATE CS_Students**
  
  `SET GPA = 3.5`

  `WHERE Stud_ID = 334322;`

- **Can be mapped to:**
  
  - **UPDATE Students**
    
    `SET GPA = 3.5`

    `WHERE ID = 334322 AND Major='CS';`
Another view update example

- CREATE VIEW Average_GPA
  AS SELECT Major, COUNT(*) AS Nstud, AVG(GPA) AS GPA
  FROM Students
  GROUP BY Major;

- UPDATE Average_GPA
  SET GPA = 4.0
  WHERE Major = 'CS';

- Can be mapped to:
  - ???
CREATE VIEW WORKS_ON_V
AS SELECT Fname, Lname, Pname
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE Ssn = Essn AND Pno = Pnumber;

UPDATE WORKS_ON_V
SET Pname = 'ProductY'
WHERE Lname = 'Smith' AND Fname = 'John' AND
Pname = 'ProductX';
Further concerns with view updates

- Should the following INSERT be allowed?
  - `INSERT INTO CS_Students
    VALUES (555555, 'Tom', 'Math', NULL);`

- What about this UPDATE?
  - `UPDATE CS_Students
    SET Major = 'English'
    WHERE Stud_ID = 444444;`

- Both of these actions migrate tuples out of the view
  - This can be disallowed with WITH CHECK OPTION
Temporary tables

- Basically, a temporary work space
  - Visible to the current SQL session
  - Automatically dropped at some point
    - At end of session
    - At end of transaction
Access control basics

- **Subjects**
  - E.g., users of the DBMS

- **Objects**
  - E.g., tables

- **Authorizations**
  - What actions is a subject allowed to take on a given object?
Granting authorizations to subjects

- **GRANT** `privilege_list` | ALL PRIVILEGES
  ON `object_list`
  TO `subject_list` | PUBLIC
  [WITH GRANT OPTION]

- Privileges:
  - SELECT: read the table
  - DELETE: remove tuples
  - INSERT[(attribute_list)]: add tuples with allowed columns
  - UPDATE[(attribute_list)]: update allowed columns of tuples
  - REFERENCE[(attribute_list)]: can use the listed columns in integrity constraints

- WITH GRANT OPTION
  - Can grant other users this authorization
What about creating a table?
Grant all undergraduates SELECT on a given table

● That's a lot of GRANTs
  ○ But does it have to be?

● RBAC
  ○ Role-based access control
  ○ CREATE rname;
  ○ GRANT rname to uname;
  ○ DROP rname;
  ○ Roles are subjects that can be GRANTed privileges just like users!
The DBA giveth, and the good DBA, he taketh away

- `REVOKE [GRANT OPTION FOR] privilege_list | ALL PRIVILEGES ON object_list FROM subject_list | PUBLIC [CASCADE | RESTRICT]
- CASCADE further revokes the privilege from all users that were granted the privilege by the user currently having it revoked
- Can also revoke roles
  - `REVOKE rname FROM uname;`
Access control examples

● DBA:
  ○ GRANT SELECT,INSERT ON Students TO Alice WITH GRANT OPTION;
  ○ CREATE ROLE Readers;
  ○ GRANT Readers TO Bob;
  ○ GRANT Readers TO Charlie;
  ○ GRANT SELECT ON Students TO Readers;

● Alice:
  ○ GRANT SELECT,INSERT ON Students TO Bob, Charlie;

● DBA:
  ○ REVOKE Readers FROM Bob;
  ○ REVOKE Readers FROM Charlie;
  ○ GRANT SELECT ON Students TO Charlie;
  ○ REVOKE ALL PRIVILEGES ON Students FROM ALICE CASCADE;
Fine-grained access controls

- What if we want to grant Alice the right to read only a subset of a table?
  - We have controls on only being able to insert into certain attributes of a table, but what about reads?