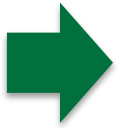
CS 61:
Database Systems

Intermediate SQL

Agenda



1. Views
2. Transactions
3. Integrity constraints

Views create virtual tables based on underlying database tables

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database)
- Consider a person who needs to know an instructor's name and department, but not the salary. This person should see a relation described, in SQL, by

  SELECT *ID*, *name*, *dept_name*
  FROM *instructor*

- A **view** can provide a mechanism to hide certain data from the view of certain users

Format

CREATE VIEW v **AS**
  SELECT A_1, A_2, ..., A_n
  FROM r

- **Why not just save data into a table?**
- **Database saves query but not data!**
- **That way if data in underlying tables changes, view is automatically up to date**
- **Can query view (or define other views) as if it were a relation**
- **An alternative is a Materialized View where data is actually stored, but MySQL does not have them**

Practice

use nyc_inspections;

Imagine you are creating a web site that provides health inspection results for fruit and vegetable restaurants in Manhattan. Each restaurant is displayed on a map and shows the average health inspection score. Create a view for this map-based data:

- You are focused on Manhattan fruit and vegetable restaurants only
- You don't need all the info in the Restaurants and Inspections tables
- Make a view for your map-based website called MapData that has attributes:
  - RestaurantID
  - RestaurantName
  - Latitude
  - Longitude
  - Average health inspection score (NULL if no inspections for this restaurant)
- Make sure you list the new restaurant you created yesterday (Tim's Tasty Treats) listed, even though it has no health inspection reports yet (NULL avg score, or for more challenge, make NULLs zero)
- Check your view works by querying it with a SELECT statement as if your view were a table

Agenda

1. Views
2. Transactions
3. Integrity constraints

Transactions allow us to write multiple statements and treat them as atomic

Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a "unit" of work
- The transaction must end with one of the following statements:
  - **COMMIT [work]** The updates performed by the transaction become permanent in the database
  - **ROLLBACK [work]** All the updates performed by the SQL statements in the transaction are undone
- Atomic transaction
  - Either fully executed or rolled back as if it never occurred

MySQL example:

SET AUTOCOMMIT = 0;

START TRANSACTION;

<SQL Statements>

COMMIT (or **ROLLBACK**)

SET AUTOCOMMIT = 1;

In MySQL by default all statements executed immediately
Turn off auto commit

Begin atomic transaction

Commit makes updates permanent, if power failed before COMMIT statement, changes would not affect database (or could use ROLLBACK to cancel changes)

Turn autocommit back on

Practice

use nyc_inspections;

Assume health inspectors have an app that allows them to enter new restaurants, and some of those new restaurants may have a new type of cuisine

- SQL allows us to create variables using SET @var
- Use the following values for this practice

  SET @RestaurantName = 'Tim''s Untasty Treats';
  SET @Building = 180; SET @Street = 'Riverside Blvd';
  SET @Boro = 'Manhattan'; SET @CuisineDescription = 'Sludge-based drinks';
- Make a new entry for Sludge-based drinks in the Cuisine table
  - The Cuisine table uses auto_increment so you'll need a way to find what ID was given for this new type of cuisine
- Make a new entry for Tim's Untasty Treats in the Restaurants table
  - Set its CuisineID to the new cuisine you just created, Sludge-based drinks
  - Set its RestaurantID to be one greater than the max RestaurantID already in the Restaurants table
- Make sure both Cuisine and Restaurant entries happen or neither operation happens, test by using ROLLBACK and COMMIT

Agenda

1. Views
2. Transactions
3. Integrity constraints

Integrity constrains ensure the data is consistent with our expectations

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

We've already seen foreign keys, but we can also ensure:

- A checking account must have a balance greater than $10,000
- The salary of an employee must be at least $15.00 an hour
- A customer must have a (non-null) phone number

Example:

```
CREATE TABLE Employees (
    EmployeeID  INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(20) NOT NULL,
    Phone INT UNIQUE,
    Salary INT,
    PRIMARY KEY (EmployeeID),
    CHECK (Salary > 0)
);
```

EmployeeID set as PRIMARY KEY (can not be NULL) and Name is not NULL as we've seen before

New constraints:

- **Phone must be unique (e.g., multiple people cannot have the same phone number)**
- **CHECK on Salary ensures Salary is greater than 0, but NULL is accepted!**

INSERT and UPDATE queries fail if constraints not met

Constraints can affect other tables as well

When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation

An alternative, in case of delete or update is to cascade

```
CREATE TABLE course (
 (…
  dept_name VARCHAR(20),
  FOREIGN KEY (dept_name) REFERENCES department(dept_name)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
   . . .)
```

Using CASCADE, if foreign key is deleted in foreign key table, cascade to this table and delete row in this table also

Instead of **CASCADE** we can also use :

SET NULL,
SET DEFAULT

Can also set this row to NULL or the attribute's default value if foreign key changes

If foreign key is changed (updated) in foreign key table, cascade to this table and update row in this table also

Practice

1. The Score for Inspections is a mess! Let's clean it up
   - Count how many times each score value appears (e.g., a score of 20 occurs 2,821 times, a score of 21 occurs 2,359 times)
   - NULL makes sense if a score was not awarded, but there are some values that do not make intuitive sense (e.g., -1)
   - Implement a strategy to deal with values that do not make sense, ensure you can get a score that is always greater than or equal to zero, but do not loose the original score values already in the table
2. Restaurants have a foreign key into the Cuisine table on Cuisine ID. What if a Cuisine is deleted?
   - Say 'Sludge-based drinks' is no longer a type of cuisine. We can delete it from the Cuisine table, but some restaurants may have an invalid CuisineID
   - Remove the NOT NULL constraint on CuisineID in the Restaurants table
   - Create a new foreign key constraint that sets a restaurant's CuisineID to NULL if that cuisine is deleted from the Cuisine table
   - Delete 'Sludge-based drinks' from Cuisine and ensure Restaurant CuisineID updates appropriately