CSE 180, Database Systems I
Shel Finkelstein, UC Santa Cruz
Lecture 5

Chapter 4 : Intermediate SQL, Part 1

Database System Concepts, 7th Edition.

Modified by Shel Finkelstein for CSE 180

Important Notices

- **Reminder:** Midterm is on **Wednesday, February 12.**
  - No make-ups, no early Midterms, no late Midterms … and no devices.
  - You may bring a **single two-sided 8.5" x 11" sheet of paper** with as much info written (or printed) on it as you can fit and read unassisted.
    - No sharing of these sheets will be permitted.
  - "Practice Midterm" from Fall 2019 has been posted on Piazza under Resources→Exams.
    - We'll be covering more material before Midterm this quarter.
    - Solution has also been posted under Resources→Exams
      … but I strongly suggest that you take it yourself first, rather than just reading the solution.
  - Hope that all requests for DRC accommodation have been submitted.
    - DRC students will receive access to webcasts only if that specific accommodation has been requested by DRC.
  - Piazza announcement about Midterm describes required seating pattern for Midterm.

Important Notices

- Lab2 solution has been posted on Piazza as lab2.zip under Resources→Lab2.
- Lab3 has been posted on Piazza, with a lab3_create.sql file that creates tables for Lab3.
  - A lab3_data_loading.sql file that is **needed** to do Lab3 will be posted soon.
  - This Lecture, Lecture 5, includes material for Lab3.
  - A transaction (from Lecture 6) is also used in Lab3.
  - Lab3 is due on **Sunday, Feb 23, 11:59pm**. You have 3 weeks to complete Lab3 because the CSE 180 Midterm is on Wednesday, Feb 12.
- The third Gradiance Assignment, ”CSE 180 Winter 2020 #3", will be assigned after the Midterm.
- Access to all webcasts went away on Wednesday, January 29.
  - Attend Lectures, Lab Sections, Office Hours and LSS Tutoring.
- See Piazza notices about LSS Tutoring with Jeshwanth Bheemanpally.

Chapter 4: Intermediate SQL

- Views
- Indexes
- Integrity Constraints
- Transactions
- Join Expressions
- SQL Data Types and Schemas
- Authorization

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 provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a view.

Some Advantages of Views

- Short-hand/encapsulation: You can treat a view as a table (for queries), which allows you to define a short-hand for a concept that might involved a complicated query.
- Re-use: You can re-use a view as often as you like; other people who can access the view may also re-use it.
- Authorization: People may be granted access to a view, even though they don't have access to the underlying tables.
  - They may not even know that the view isn't a table.
  - Even if they know that it's a view, they may not know which tables underlie it.
- Logical data independence: Even if the tables underlying a view change, the view may still be used in queries (or applications), without re-writing the queries (or applications).

View Definition

- A view is defined using the **CREATE VIEW** statement which has the form

  CREATE VIEW *v* **AS** < query expression >

  where <query expression> is any legal SQL query.
  - The view name is *v.*
- After a view has been defined, the view name can be used to refer to the virtual relation corresponding to the view.
- View definition does not create a new relation!
  - Rather, creating a view saves the expression defining the view in the DBMS's system catalog.
  - The DBMS automatically substitutes that expression into any SQL statements that use the view.

View Definition and Use

- Define a view of instructors that doesn't give their salaries.

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

- Find all instructors in the Biology department using the view.

faculty

```
SELECT name
FROM faculty
WHERE dept_name = 'Biology';
```

- Create a view that provides total salary for each department.

```
CREATE VIEW departmentTotalSalary(dept_name, total_salary) AS
    SELECT dept_name, SUM(salary)
    FROM instructor
    GROUP BY dept_name;
```

or

```
CREATE VIEW departmentTotalSalary AS
    SELECT dept_name, SUM(salary) AS total_salary
    FROM instructor
    GROUP BY dept_name;
```

Views Defined Using Other Views

Views may be used (together with other views and tables) to define another view.

- **CREATE VIEW** *physics_fall_2017* **AS**
  SELECT *C.course_id*, S.*sec_id*, S.*building*, S.*room_number*
  FROM *course C*, *section S*
  WHERE *C.course_id* = *S.course_id*
  and *C.dept_name* = 'Physics'
  and *S.semester* = 'Fall'
  and *S.year* = '2017';

- **CREATE VIEW** *physics_fall_2017_watson* **AS**
  SELECT *course_id*, *room_number*
  FROM *physics_fall_2017*
  WHERE *building* = 'Watson';

Evaluating a View Defined on a View

- The view:

```
CREATE VIEW physics_fall_2017_watson AS
   SELECT course_id, room_number
   FROM physics_fall_2017
   WHERE building = 'Watson';
```

- Becomes:

```
CREATE VIEW physics_fall_2017_watson AS
   SELECT P.course_id, P.room_number
   FROM ( SELECT C.course_id, S.building, S.room_number
            FROM course C, section S
            WHERE C.course_id = S.course_id
                AND C.dept_name = 'Physics'
                AND S.semester = 'Fall'
                AND S.year = '2017' ) AS P
    WHERE P.building = 'Watson';
```

Evaluating a Query Defined on a View

- The query:

```
SELECT P.course_id, P.room_number, Teach.ID
FROM physics_fall_2017 P, teaches Teach
WHERE P.course_id = Teach.course_ID
    AND Teach.semester = 'Fall'
    AND Teach.year = '2017';
```

- Becomes:

```
SELECT P.course_id, P.room_number, Teach.ID
FROM ( SELECT C.course_id, S.building, S.room_number
        FROM course C, section S
        WHERE C.course_id = S.course_id
            AND C.dept_name = 'Physics'
            AND S.semester = 'Fall'
            AND S.year = '2017' ) AS P,
      teaches Teach
WHERE P.course_id = Teach.course_ID
    AND Teach.semester = 'Fall'
    AND Teach.year = '2017';
```

DROP VIEW

CREATE VIEW *physics_fall_2017* **AS**
  SELECT *C.course_id*, S.*sec_id*, S.*building*, S.*room_number*
  FROM *course C*, *section S*
  WHERE *C.course_id* = *S.course_id*
        and *C.dept_name* = 'Physics'
        and *S.semester* = 'Fall'
        and *S.year* = '2017';

DROP VIEW *physics_fall_2017* ;

- What happens if after DROP VIEW you execute the following?
  - **SELECT** * **FROM** *physics_fall_2017*;
  - **SELECT** * **FROM** *course*;
  - **SELECT** * **FROM** *section*;
  - **SELECT** * **FROM** *physics_fall_2017_watson*;

Materialized Views

- Normally, only the view definition is stored in the database; the contents of the view are not stored.
- Certain database systems allow some view relations to be physically stored, if explicitly requested.
  - A physical copy of the contents is stored when the view is defined.
  - Such views are called **Materialized Views**.
- If relations used in the query defining the view are updated, then the Materialized View contents becomes out-of-date.
  - **Maintaining** the Materialized View would require changing the view contents whenever its underlying relations are updated.
  - This can be expensive, so it's unusual to maintain Materialized Views.
  - Moreover, some views are not updatable … as we'll see.

"Update" of a View

- Add a new tuple to *faculty* view which we defined earlier

  INSERT INTO *faculty*

  VALUES ('30765', 'Green', 'Music');

- This insertion must be represented by the insertion into the *instructor* relation
  - … so it must have a value for salary.
- Two approaches
  - Reject the insert.
  - Insert the following tuple into the *instructor* relation:

    ('30765', 'Green', 'Music', **NULL**)
- Which approach is used?
  - What if salary can't be **NULL?**
  - What if salary can be **NULL?**

But Some "Updates" Can't be Translated!

- **CREATE VIEW** *instructor_info* **as**
  SELECT *instructor.ID*, *instructor.name*, *department.building*
  FROM *instructor*, *department*
  WHERE *instructor.dept_name= department.dept_name*;

- **INSERT INTO** *instructor_info*
  VALUES ('69987', 'White', 'Taylor');
- What tuples should be inserted to make this work?
  - Easy … insert tuple into instructor that has the dept_name of the department whose building is Taylor!
  - But which department, if multiple departments are in Taylor?
  - And what if no department is in Taylor?

View Updates in SQL

- Most SQL implementations allow updates only on very simple views.
  - The **FROM** clause has only one database relation.
  - The **SELECT** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **DISTINCT** specification.
  - Any attribute not listed in the **SELECT** clause can be set to **NULL.**
  - The query does not have a **GROUP BY** or **HAVING** clause.

Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **CREATE INDEX** command

  CREATE INDEX <name> **ON** <relation-name> (attributes);

Index Creation Example

- **CREATE TABLE** *student*
  (*ID* **VARCHAR** (5),
  name **VARCHAR** (20) **NOT NULL**,
  dept_name **VARCHAR** (20),
  tot_cred **NUMERIC** (3,0) **DEFAULT** 0,
  PRIMARY KEY (*ID*))
- **CREATE INDEX** *studentID_index* **ON** *student*(*ID*)
- The DBMS can execute the query:

  SELECT *
  FROM *student*
  WHERE *ID* = '12345'

  by using the index to find the required record, without looking at all records of *student.*

Table Modifications and Indexes

- If a table is updated, all Indexes on that table are immediately automatically updated within the same transaction.
  - Which indexes do you need to change on INSERT and DELETE?
  - What about UPDATE?
- SQL statements can be executed regardless of which indexes (if any) exist in the database.
  - Physical Independence!
  - Applications don't have to be modified when indexes are created or dropped!
- What gets impacted when indexes are created or dropped?
  - Performance of SQL statements
  - Some may run faster, some may run slower.

Benefits of Indexes for Queries

SELECT *
FROM student
WHERE name = 'Lee' AND tot_cred = 90;

How much would each of these indexes help?

CREATE INDEX Cred_Index ON student(tot_cred);

CREATE INDEX NameIndex ON student(name);

CREATE INDEX CNIndex ON student(tot_cred, name);

CREATE INDEX NCIndex ON student(name, tot_cred);

How much would each of the indexes help if the WHERE clause was just tot_cred = 90?

Benefits of Indexes for Another Query

SELECT *
FROM student
WHERE name = 'Lee' AND **tot_cred < 90**;

How much would each of these indexes help?

CREATE INDEX CredIndex ON student(tot_cred);

CREATE INDEX NameIndex ON student(name);

CREATE INDEX CNIndex ON student(tot_cred, name);

CREATE INDEX NCIndex ON student(name, tot_cred);

How much would each of the indexes help if the WHERE clause was just tot_cred < 90?

Disadvantages of Indexes?

- Why not put indexes on every attributes, or even on every combination of attributes that you might query on?
  - Huge number of indexes
  - Space for indexes
  - Cache impact of searching indexes
  - Update time for indexes when table is modified

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.
  - Every checking account must have a balance greater than $100.00
  - The salary of a bank employee must be at least $15.00 an hour.
  - A customer must have a (non-null) phone number.
  - Only known students can take classes.
  - If a student is taking a 5 credit course, then their GPA must be at least 3.0

Constraints on a Single Relation

- **NOT NULL**
- **PRIMARY KEY**
- **UNIQUE**
- **CHECK** (P), where P is a predicate

NOT NULL Constraints

- **NOT NULL**
  - Declare *name* and *budget* to be **NOT NULL**

    name **VARCHAR**(20) **NOT NULL**
    budget **NUMERIC**(12,2) **NOT NULL**

UNIQUE Constraints

- **UNIQUE** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes A_1, A_2, …, A_m form a candidate key.
  - Attributes in candidate keys may be **NULL** (unlike **PRIMARY KEY**).
    - If all the attributes A_1, A_2, …, A_m are not **NULL**, then there can't be multiple tuples that have the same value.
  - There may be multiple UNIQUE specifications for a table (unlike **PRIMARY KEY**).

CHECK Constraint

- A **CHECK** (P) constraint specifies that a predicate P that must be satisfied by every tuple in a relation.
- Example: Ensure that semester is one of fall, winter, spring or summer.

```
CREATE TABLE section
   ( course_id VARCHAR (8),
     sec_id VARCHAR (8),
     semester VARCHAR (6)
          CHECK ( semester IN ('Fall', 'Winter', 'Spring', 'Summer') ),
     year NUMERIC (4,0),
     building VARCHAR (15),
     room_number VARCHAR (7),
     time slot id VARCHAR (4),
     PRIMARY KEY (course_id, sec_id, semester, year) );
```

- A **CHECK** constraint is violated if its value is FALSE. It is not violated if its value is TRUE or if its value is UNKNOWN.

Multiple Attributes in a CHECK Constraint

CHECK can refer to multiple attributes in the tuple. But if it does, then the **CHECK** must be a separate schema element.

- Ensure that if year is after 2019, then the building is not Watson.

```
CREATE TABLE section
   ( course_id VARCHAR (8),
     sec_id VARCHAR (8),
     semester VARCHAR (6),
     year NUMERIC (4,0),
     building VARCHAR (15),
     room_number VARCHAR (7),
     time slot id VARCHAR (4),
     PRIMARY KEY (course_id, sec_id, semester, year),
     CHECK ( year <= 2019 OR building <> 'Watson' ) );
```

Complex CHECK Constraints

- The SQL standard allows a predicate in a **CHECK** constraint to reference other tuples or relations, using a subquery.

Example:

CHECK (*time_slot_id* **IN**
(**SELECT** *time_slot_id* **FROM** *time_slot*))

- However, this capability in not implemented in most relational DBMS, so we won't discuss it.

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If 'Biology' is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for 'Biology'.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the PRIMARY KEY of S. A is said to be a **FOREIGN KEY** of R if for any values of A appearing in R these values also appear in S.

Specifying Referential Integrity

- ***Foreign keys*** *can be* specified as part of the SQL **CREATE TABLE** statement

  FOREIGN KEY (*dept_name*) **REFERENCES** *department*

- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly, so the attribute names don't have to be the same.
  - For example, if the key of *department* was *dname* instead of *dept_name*, we could write

  FOREIGN KEY (*dept_name*)

  REFERENCES *department* (*dname*)

Specifying Referential Integrity (Cont.)

If department's Primary Key was dname instead of dept_name, then we could write either of the following:

```
CREATE TABLE instructor (
        ID          char(5),
        name        VARCHAR(20) NOT NULL,
        dept_name   VARCHAR(20),
        salary      NUMERIC(8,2),
        PRIMARY KEY (ID),
        FOREIGN KEY (dept_name)
                REFERENCES department(dname) );
```

```
CREATE TABLE instructor (
        ID          char(5) PRIMARY KEY,
        name        VARCHAR(20) NOT NULL,
        dept_name   VARCHAR(20) REFERENCES department(dname),
        salary      NUMERIC(8,2) );
```

Specifying Referential Integrity (Cont.)

How would the **FOREIGN KEY** specification in *takes* change, if attribute names were different in *section*?

CREATE TABLE *takes* (
ID **VARCHAR**(5),
course_id **VARCHAR**(8),
sec_id **VARCHAR**(8),
semester **VARCHAR**(6),
year **NUMERIC**(4,0),
grade **VARCHAR**(2),
PRIMARY KEY *(ID, course_id, sec_id, semester, year)* ,
FOREIGN KEY (*ID*) **REFERENCES** *student,*
FOREIGN KEY (*course_id, sec_id, semester, year*)
REFERENCES *section*);

FOREIGN KEY (*course_id, sec_id, semester, year*)
REFERENCES *section(course, sec, sem, year*)

Actions to Ensure Referential Integrity

- When a referential-integrity constraint is violated, the default action 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
        ON DELETE CASCADE
        ON UPDATE CASCADE,
      . . .);
```

- Instead of **CASCADE** we can specify:
  - **SET NULL**
  - **SET DEFAULT**

Argh; a Foreign Key can be NULL, so there won't be a corresponding value for it in the Referenced table!

- The specifications for **DELETE** and **UPDATE** may be different.
  - If either (or both) are omitted, default action (reject) is followed.

Use of ALTER for Referential Integrity

- If instructor had been created without specifying dept_name as a Foreign Key (unusual but possible):

```
CREATE TABLE instructor (
        ID          char(5),
        name        VARCHAR(20) NOT NULL,
        dept_name   VARCHAR(20),
        salary      NUMERIC(8,2),
        PRIMARY KEY (ID) );
```

then we could do the following:

```
ALTER TABLE instructor
    ADD FOREIGN KEY (dept_name)
    REFERENCES department;
```

- What do you think would happen if an *instructor* tuple had a *dept_name* that isn't a *dept_name* in *department*?

Circular Integrity Constraints are a Problem!

- Consider:

  CREATE TABLE *person* (
  ID **char**(10),
  name **char**(40),
  mother **char**(10) **REFERENCES** *person*,
  father **char**(10) **REFERENCES** *person*,
  PRIMARY KEY *ID*);

- How can we insert any tuples without causing a constraint violation?
  - Insert father and mother of a person before inserting person,
  - … or set father and mother to **NULL** initially, update after inserting all persons (not possible if father and mother attributes are declared to be **NOT NULL**),
  - or **DEFER** constraint checking.

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- The following constraints, can be expressed using assertions:
- For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot
- An assertion in SQL HAS the form:

  CREATE ASSERTION <assertion-name> **CHECK** (<predicate>);
- Database systems do not implement assertions because they're too complicated and expensive.
  - We won't discuss them in this course.