CSCB20
Introduction to Databases and Web Application

Week 3 - SQL

Dr. Purva R Gawde

Topics

- Last Week:
  - Mapping from Relational Algebra to SQL
  - Intro to SQL
  - Focused on queries to start – assumed tables and database exist.
- This Week:
  - DDL
    - Creating tables, setting constraints…
    - Inserting and updating tables
  - DML
    - More query commands
    - Creating views
    - Joins
    - More on NULL values

DDL - Data Definition Language

Data Definition

- DDL (Data Definition Language) allows specification of
  - Schema of relation
    - Create Table – show you how to create a new table in the database.
    - Insert values
  - Type of values associated with each attribute
    - Domain of relation
  - Integrity constraints
    - NOT NULL, UNIQUE, CHECK, Primary Key, Foreign Key
  - Data Modification:
    - Alter Table – show you how to use modify the structure of an existing table.
    - Rename column – learn step by step how to rename a column of a table.
    - Drop Table – guide you on how to remove a table from the database.

Domain Types in SQL

- `char(n)`:
  - A fixed-length character string with user-specified length n.
- `varchar(n)`:
  - A variable length character string with user-specified maximum length n.
- `int`:
  - An integer(a finite subset of the integers that is machine dependent).
- `smallint`:
  - A small integer (a machine-dependent subset of the integer type).
- `numeric(p,d)`:
  - The number consists of p digits (plus a sign), and d of the p digits are to the right of the decimal point.
- `real, double precision`:
  - Floating-point and double-precision floating-point numbers with machine-dependent precision.
- `float(n)`:
  - A floating-point number with precision of at least n digits.

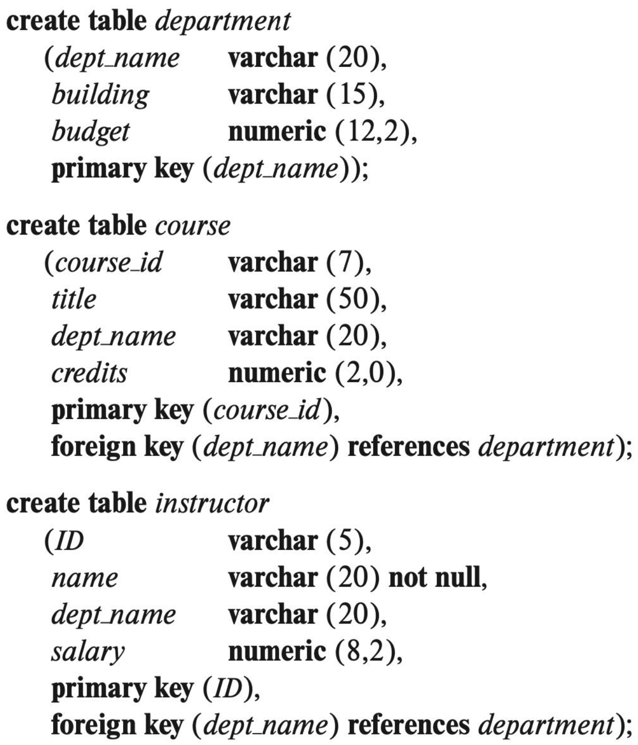
Creating Table

```
CREATE TABLE [IF NOT EXISTS] [schema_name].table_name

(     column_1 data_type PRIMARY KEY,

      column_2 data_type NOT NULL,

      column_3 data_type DEFAULT 0,

      table_constraints

)
```

Insert Values:

INSERT INTO Tablename(colname1, colname2, ….) VALUES(value1, value2, ….);

INSERT INTO Tablename VALUES(value1, value2, ….);



```
create table department
   (dept_name     varchar (20),
    building      varchar (15),
    budget        numeric (12,2),
    primary key (dept_name));

create table course
   (course_id     varchar (7),
    title         varchar (50),
    dept_name     varchar (20),
    credits       numeric (2,0),
    primary key (course_id),
    foreign key (dept_name) references department);

create table instructor
   (ID            varchar (5),
    name          varchar (20) not null,
    dept_name     varchar (20),
    salary        numeric (8,2),
    primary key (ID),
    foreign key (dept_name) references department);
```

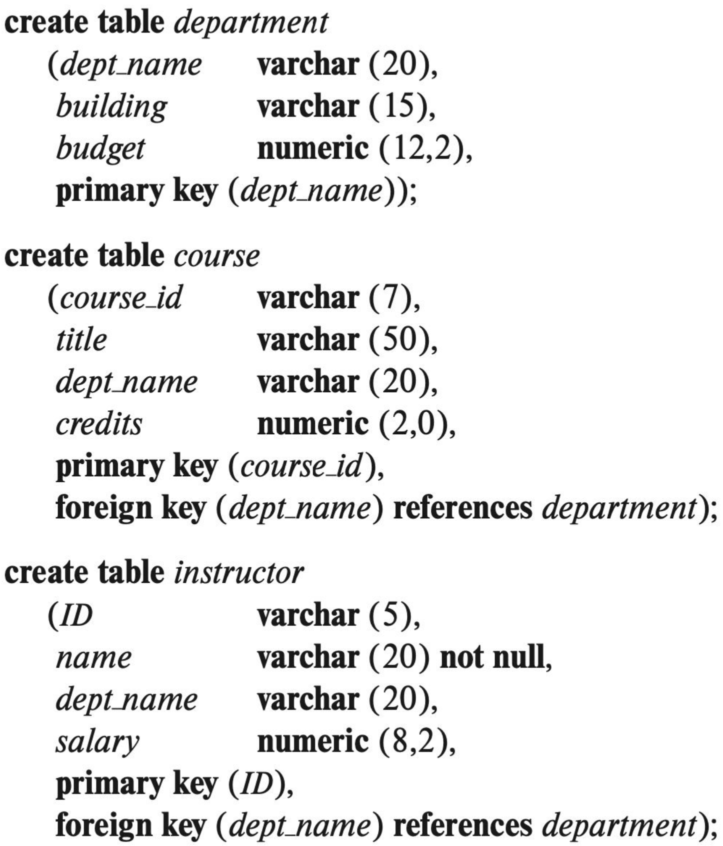
SQL constraints

- SQL Create Constraints
- Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.
- Syntax

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ....
);
```

SQL constraints

- Primary Key
  - show you how to define the primary key for a table.
- NOT NULL constraint
  - learn how to enforce values in a column are not NULL.
- UNIQUE constraint
  - ensure values in a column or a group of columns are unique.
- CHECK constraint
  - ensure the values in a column meet a specified condition defined by an expression.
- Domain integrity constraint
- Foreign Key constraints



```
create table department
    (dept_name      varchar (20),
     building       varchar (15),
     budget         numeric (12,2),
     primary key (dept_name));

create table course
    (course_id      varchar (7),
     title          varchar (50),
     dept_name      varchar (20),
     credits        numeric (2,0),
     primary key (course_id),
     foreign key (dept_name) references department);

create table instructor
    (ID             varchar (5),
     name           varchar (20) not null,
     dept_name      varchar (20),
     salary         numeric (8,2),
     primary key (ID),
     foreign key (dept_name) references department);
```

Editing Tables

```
DROP TABLE table_name;            //remove the table

DELETE FROM table_name            //delete tuples satisfying the predicate
WHERE predicate;

ALTER TABLE table_name            //add a column
ADD column type;

ALTER TABLE table_name            // remove a column
DROP column;
```

Inserting in SQLite

Inserting Single row:

```
INSERT INTO table (column1,column2 ,..)
VALUES( value1, value2 ,...);
```

Inserting multiple rows:

```
INSERT INTO table1 (column1,column 2 ,..)
VALUES
   (value1,value2 ,...),
   ...
   (valuen,valuen ,...);
```

Inserting using SELECT query:

```
INSERT INTO table_name
SELECT QUERY
```

For example:

```
INSERT INTO instructor
SELECT ID, name, dept_name, 30000
FROM student
WHERE dept_name = ‘Music’ AND tot_cred > 30;
```

Updating in SQLite

```
UPDATE table_name
    SET attribute = new_value
OR
UPDATE table_name
    SET attribute = new_value
    WHERE predicate or select statement;
OR
UPDATE table_name
    SET attribute = CASE
        WHEN predicate$_1$ THEN result$_1$
        WHEN predicate$_2$ THEN result$_2$
        …
        WHEN predicate$_n$ THEN result$_n$
        ELSE result$_0$
       END
```

Deleting and Dropping in SQLite

- DELETE statement allows you to delete one row, multiple rows, and all rows in a table.

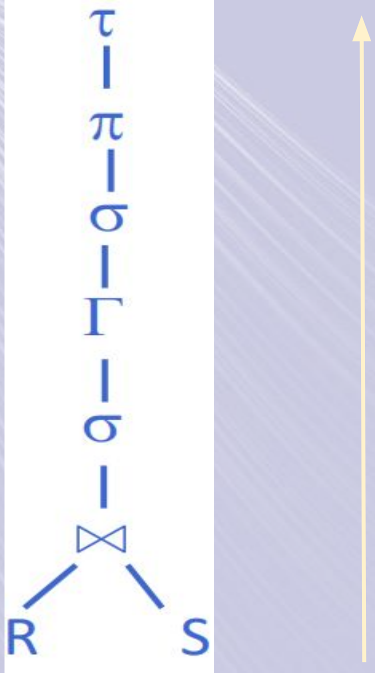
```
DELETE FROM table
WHERE search_condition;
```

- First, specify the name of the table which you want to remove rows after the DELETE FROM keywords. Second, add a search condition in the WHERE clause to identify the rows to remove. The WHERE clause is an optional part of the DELETE statement.
- To remove a table in a database, you use SQLite DROP TABLE statement

```
DROP TABLE [IF EXISTS] [schema_name.]table_name;
```

DML - Data Manipulation Language

Simple SQL Query



ORDER BY

SELECT

HAVING

GROUP BY

WHERE

FROM

Simple Query

- Select – query data from a single table using SELECT statement.
- Querying data from a table using the SELECT statement

```
SELECT DISTINCT column_list
FROM table_list JOIN table ON join_condition
WHERE row_filter
ORDER BY column
LIMIT count OFFSET offset
GROUP BY column
HAVING group_filter;
```

SELECT DISTINCT Clause

- The DISTINCT clause is an optional clause of the SELECT statement. The DISTINCT clause allows you to remove the duplicate rows in the result set.

```
SELECT DISTINCT select_list
FROM table;
```

```
SELECT DISTINCT dept_name
FROM instructor;
```

WHERE Clause

- The WHERE clause is an optional clause of the SELECT statement. It appears after the FROM clause as the following statement:

```
SELECT
    column_list
FROM
    table
WHERE
    search_condition;
```

```
SELECT name
FROM student
WHERE dept_name = 'Comp. Sci.';
```

```
SELECT *
FROM instructor
WHERE salary BETWEEN 40000 AND 80000;
```

ORDER BY Clause

- SQLite stores data in the tables in an unspecified order. It means that the rows in the table may or may not be in the order that they were inserted.
- Use Order by to sort the result set

```
SELECT
   select_list
FROM
   table
ORDER BY
    column_1 ASC,
    column_2 DESC;
```

```
SELECT name
FROM student
WHERE dept_name = 'Comp. Sci.'
ORDER BY name ASC;
```

LIMIT Clause

- The LIMIT clause optional part of the SELECT statement to constrain the number of rows returned by the query.

```
SELECT
    column_list
FROM
    table
LIMIT row_count;
```

```
SELECT name
FROM student
WHERE dept_name = 'Comp. Sci.'
ORDER BY name ASC
LIMIT 2;
```

Additional Basic Operations

Rename

Consider following query:

```
SELECT name, course_id
FROM instructor, teaches
WHERE instructor.ID= teaches.ID;
```

Renaming the resulting attributes:

```
SELECT name AS instructor_name, course_id
FROM instructor, teaches
WHERE instructor.ID= teaches.ID;
```

You can also rename tables

```
SELECT name, course_id
FROM instructor AS I, teaches AS T
WHERE I.ID = T.ID;
```

Comparing tuples of same relation:
Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.

```
SELECT T.name
FROM instructor AS T, instructor AS S
WHERE T.salary > S.salary AND
       S.dept_name = 'Biology';
```

Pattern Matching - like operator

- To query data based on partial information, you use the LIKE operator in the WHERE clause of the SELECT statement as follows:

  ```
  SELECT column_list
  FROM table_name
  WHERE column_1 LIKE pattern;
  ```
- Patterns are described by using two special characters:
  - Percent (%): The % character matches any substring.
    - 'Intro%' matches any string beginning with “Intro”.
    - '%Comp%' matches any string containing “Comp” as a substring, for example,
      - 'Intro. to Computer Science', and 'Computational Biology'.
  - Underscore (_): The _character matches any character.
    - ' ___ ' matches any string of exactly three characters.
    - ' ___ %' matches any string of at least three characters.

```
select *
from department
where building like '%Watson%';
```

WHERE clause predicates

- Between comparison operator

```
select name
from instructor
where salary <= 100000 and
salary >= 90000;
```

↔

```
select name
from instructor
where salary between 90000 and 100000;
```

- IN operator

```
Select *
from department
where dept_name IN
       ('Biology', 'Comp. Sci.');
```

↔

```
SELECT *
FROM department
WHERE dept_name = 'Biology' OR
dept_name = 'Comp. Sci.';
```

Set Operations

Union, Intersect, Except
(No Duplicates)

Union

- The set of all courses taught either in Fall 2017 or in Spring 2018, or both

```
select course_id
from section
where semester = 'Fall' and year= 2017
union
select course_id
from section
where semester = 'Spring' and year= 2018;
```

- How to retain duplicates?

```
select course_id
from section
where semester = 'Fall' and year= 2017
union all
select course_id
from section
where semester = 'Spring' and year= 2018;
```

Intersect

- The set of all courses taught in both Fall 2017 and in Spring 2018

```
SELECT course_id
FROM section
WHERE semester = 'Fall' and year= 2017
INTERSECT
SELECT course_id
FROM section
WHERE semester = 'Spring' and year= 2018;
```

- How to retain duplicates?

```
SELECT course id
FROM section
WHERE semester = 'Fall' and year= 2017
INTERSECT ALL
SELECT course id
FROM section
WHERE semester = 'Spring' and year= 2018;
```

`INTERSECT ALL is not supported in SQLite`

Except

- The set of all courses taught in both Fall 2017 but not in Spring 2018

```
select course_id
from section
where semester = 'Fall' and year= 2017
except
select course_id
from section
where semester = 'Spring' and year= 2018;
```

- How to retain duplicates?

```
select course_id
from section
where semester = 'Fall' and year= 2017
except all
select course_id
from section
where semester = 'Spring' and year= 2018;
```

EXCEPT ALL is not supported in SQLite

NULL values

NULL Values

- Every type can have the special value null.
- A value of null indicates the value is unknown or that it may not exist at all.
- Sometimes we do not want a null value at all – we can add such a constraint.
- Because of NULL, we need three truth-values:
    - If one or both operands to a comparison is NULL, the comparison always evaluates to UNKNOWN.
    - Otherwise, comparisons evaluate to TRUE or FALSE.

NULL Values

- We can check for NULL values using:
  - IS NULL
  - IS NOT NULL
- Because we have NULL, we need three truth values for comparisons:
  - TRUE, FALSE and UNKNOWN
  - If one or both operands is NULL, the comparison always evaluates to UNKNOWN.
  - Otherwise, comparisons evaluate to TRUE and FALSE.

NULL Values

A	B	A AND B	A OR B
T	T	T	T
TF or FT		F	T
F	F	F	F
TU or UT		U	T
FU or UF		F	U
U	U	U	U

A	NOT A
T	F
F	T
U	U

IS NULL operator

- To check if a value is NULL or not, you use the `IS NULL` operator
- To find all instructors who appear in the instructor relation with null values for salary

```
SELECT name
FROM instructor
WHERE salary IS NULL;
```

Data Modification

Drop, Alter Table

- Insert :
  - `insert into instructor values ('10211', 'Smith', 'Biology', 66000);`
- Delete
  - Remove all tuples from the student relation
    - `delete from student;`
- Drop Table

```
DROP TABLE students;
```

- Alter Table
  - to rename:

```
ALTER TABLE students
RENAME TO students_compsci;
```

to add column and rename column:

```
ALTER TABLE students
ADD COLUMN age INT;
ALTER TABLE students
RENAME COLUMN Address TO Streets;
```

Deletion

- A delete request is expressed in much the same way as a query.

  ```
  delete from r
  where P;

  delete from instructor;
  ```

- Other examples:

  -
    ```
    delete from instructor
    Where dept_name = ‘Finance’;
    ```

  -
    ```
    delete from instructor
    where salary between 13000 and 15000;
    ```

Insertion

- Suppose that we wish to insert the fact that there is a course CS-437 in the Computer Science department with title “Database Systems” and four credit hours.

  -
    ```
    insert into course
        values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
    ```
- OR

  -
    ```
    insert into course (course_id, title, dept_name, credits)
        values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
    ```
- OR

  -
    ```
    insert into course (title, course_id, dept_name, credits)
        values ('Database Systems', 'CS-437', 'Comp. Sci.', 4);
    ```

Insertion contd..

- General format of insert: insert tuples on the basis of the result of a query.
- Suppose that we want to make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

    -
      ```
      insert into instructor
          select ID, name, dept_name, 18000
          from student where dept_name = 'Music' and tot_cred > 144;
      ```

Updates

- change a value in a tuple without changing all values in the tuple
- salaries of all instructors are to be increased by 5 percent

```
update instructor
set salary= salary * 1.05;
```

- If a salary increase is to be paid only to instructors with a salary of less than $70,000, we can write

```
update instructor
set salary= salary * 1.05
where salary < 70000;
```

Updates - with case construct

- SQL provides a case construct that we can use to perform multiple updates
  General form:

```
case
     when pred1 then result1 when pred2 then result2 ...
     when predn then resultn else result0
end
```

- Instructors with salary over $100,000 receive a 3 percent raise, whereas all others receive a 5 percent raise.

```
update instructor set salary = case
                                    when salary <= 100000 then salary * 1.05
                                    else salary * 1.03
                               end
```

Filtering Data

Filtering Data

- `Select Distinct` – query unique rows from a table using the DISTINCT clause.
- `Where` – filter rows of a result set using various conditions.
- `Limit` – constrain the number of rows returned by a query and how to get only the necessary data from a table.
- `Between` – test whether a value is in a range of values.
- `In` – check if a value matches any value in a list of values or subquery.
- `Like` – query data based on pattern matching using wildcard characters: percent sign (%) and underscore (_).
- `IS NULL` – check if a value is null or not.

Remaining Clauses

```
SELECT DISTINCT column_list
FROM table_list
  JOIN table ON join_condition
WHERE row_filter
ORDER BY column
LIMIT count OFFSET offset
GROUP BY column
HAVING group_filter;
```

Use INNER JOIN or LEFT JOIN to query data from multiple tables using join.

Use GROUP BY to get the group rows into groups and apply **aggregate function** for each group. Use HAVING clause to filter groups

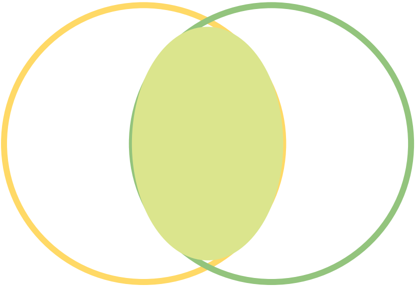
Joining Tables SQL

Join

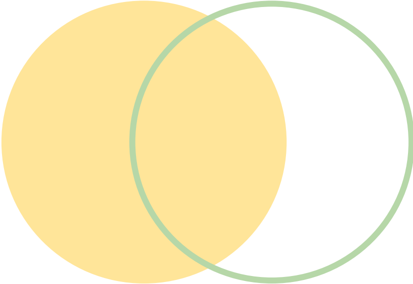
- A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
- INNER JOIN:
  - Returns records that have matching values in both tables
- LEFT JOIN:
  - Returns all records from the left table, and the matched records from the right table
- RIGHT JOIN:
  - Returns all records from the right table, and the matched records from the left table
- FULL JOIN:
  - Returns all records when there is a match in either left or right table

Join

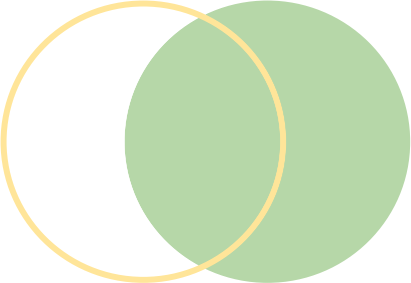
(INNER) JOIN



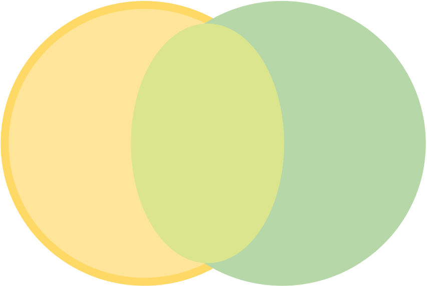
LEFT JOIN



RIGHT JOIN



FULL JOIN



LEFT JOIN

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R LEFT JOIN S

A	B	C
1	2	3
4	5	NULL

RIGHT JOIN

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R RIGHT JOIN S

A	B	C
1	2	3
NULL	6	7

FULL OUTER JOIN

R

A	B
1	2
4	5

S

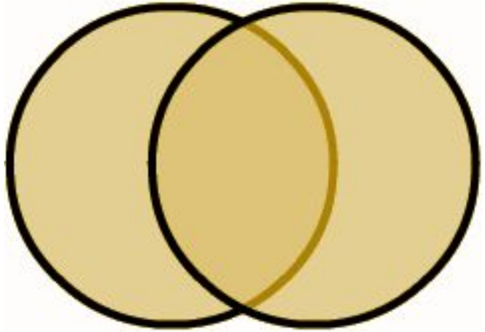
B	C
2	3
6	7

R FULL OUTER JOIN S

A	B	C
1	2	3
4	5	NULL
NULL	6	7

SQL FULL OUTER JOIN

- FULL OUTER JOIN is a combination of a LEFT JOIN and a RIGHT JOIN
- NULL values for every column of the table that does not have a matching row in the other table
- For the matching rows, the FULL OUTER JOIN produces a single row with values from columns of the rows in both tables.



SQL FULL OUTER JOIN - Example

```
-- create and insert data into the dogs table
CREATE TABLE dogs (
    type  TEXT,
    color TEXT
);

INSERT INTO dogs(type, color)
VALUES('Hunting','Black'), ('Guard','Brown');

-- create and insert data into the cats table
CREATE TABLE cats (
    type  TEXT,
    color TEXT
);

INSERT INTO cats(type,color)
VALUES('Indoor','White'),  ('Outdoor','Black');
```

dogs

type	color
Hunting	Black
Guard	Brown

cats

type	color
Indoor	White
Outdoor	Black

SQL FULL OUTER JOIN - Example

dogs

type	color
Hunting	Black
Guard	Brown

cats

type	color
Indoor	White
Outdoor	Black

```
SELECT *
FROM dogs
FULL OUTER JOIN cats
    ON dogs.color = cats.color;
```

type	Color	type	Color
Hunting	Black	Outdoor	Black
Guard	Brown	NULL	NULL
NULL	NULL	Indoor	White

Joining Tables SQLite

SQLite Join

INNER JOIN… ON
JOIN … USING
NATURAL JOIN
LEFT JOIN
CROSS JOIN

SQLite INNER JOIN

- Returns only the rows that match the join condition and eliminate all other rows that don't match the join condition.

```
SELECT
  name,
  student.dept_name
FROM Student
INNER JOIN Department ON Student.dept_name = Department.dept_name;
```

SQLite JOIN Using

- Returns only the rows that match the join condition and eliminate all other rows that don't match the join condition.

```
SELECT
  name,
  student.dept_name
FROM Student
    INNER JOIN Department USING(dept_name);
```

SQLite Natural JOIN

- A NATURAL JOIN automatically tests for equality between the values of every column that exists in both tables.

```
SELECT
  name,
  student.dept_name
FROM Student NATURAL JOIN Department;
```

SQLite LEFT JOIN

- All the values of the columns you select from the left table will be included in the result of the query

```
SELECT
  name,
  student.dept_name
FROM Student
LEFT JOIN Department ON Student.dept_name = Department.dept_name;
```

SQLite CROSS JOIN

- Cartesian product for the selected columns of the two joined tables, by matching all the values from the first table with all the values from the second table.

```
SELECT
  name,
  student.dept_name
FROM Student CROSS JOIN Department;
```

SQLite FULL OUTER JOIN

dogs

type	color
Hunting	Black
Guard	Brown

cats

type	color
Indoor	White
Outdoor	Black

```
SELECT d.type,
         d.color,
         c.type,
         c.color
FROM dogs d
LEFT JOIN cats c USING(color)
UNION ALL
SELECT d.type,
         d.color,
         c.type,
         c.color
FROM cats c
LEFT JOIN dogs d USING(color)
WHERE d.color IS NULL;
```

Aggregate Functions

avg, min, max, sum, count

Basic Aggregation

- Find the average salary of instructors in the Computer Science department

```
SELECT avg(salary) AS avg_salary
FROM instructor
WHERE dept_name = 'Comp. Sci.';
```

Basic Aggregation contd..

- Sometimes we must eliminate duplicates
    - Find the total number of instructors who teach in the Spring 2018 semester:

      ```
      select count (distinct ID)
      from teaches
      where semester = 'Spring' and year = 2018;
      ```

- Frequent use of count:
    - Counting tuples

      ```
      select count (*)
      from course;
      ```

Aggregation with Grouping

- Find the average salary of instructors in each department

```
SELECT dept_name, avg(salary) AS avg_salary
FROM instructor
GROUP BY dept_name;
```

- As opposed to: Find the average salary of all instructors

```
SELECT avg(salary)
FROM instructor;
```

Aggregation with Grouping

- Find the number of instructors in each department who teach a course in the Spring 2018 semester.

```
SELECT dept_name, count(DISTINCT Instructor.Id) as instr_count
FROM instructor, teaches
WHERE instructor.Id = teaches.Id and semester = 'Spring' and year = 2018
GROUP BY dept_name;
```

- important to ensure that the only attributes that appear in the select statement without being aggregated are those that are present in the group by clause.

- /* erroneous query */

```
SELECT dept_name, Id, avg(salary)
FROM instructor
GROUP BY dept_name;
```

Grouping Data

GROUP BY clause

- GROUP BY clause to make a set of summary rows from a set of rows.
- Returns one row for each group.
- For each group, you can apply an aggregate function such as MIN, MAX, SUM, COUNT, or AVG to provide more information about each group

```
SELECT
    column_1,
    aggregate_function(column_2)
FROM
    table
GROUP BY
    column_1,
    column_2;
```

GROUP BY Example

Total number of students present in each department

```
SELECT d.dept_name, COUNT(s.Id) AS StudentsCount
FROM Student AS s
INNER JOIN Department AS d ON s.dept_name = d.dept_name
GROUP BY d.dept_name
ORDER BY d.dept_name, s.tot_cred;
```

HAVING clause

- **Filter** the groups returned by the GROUP BY clause

```
SELECT
    column_1,
    aggregate_function(column_2)
FROM
    table
GROUP BY
    column_1,
    column_2;
HAVING
search_condition;
```

HAVING Example

- Find departments where exactly two students are present.

```
SELECT d.dept_name, COUNT(s.Id) AS StudentsCount
FROM Student AS s
INNER JOIN Department AS d ON s.dept_name = d.dept_name
GROUP BY d.dept_name
HAVING COUNT(s.Id) = 2;
```

Views

What is a View?

- A view is a virtual relation.
- A view is a result set of a stored query.
  - way to pack a query into a named object stored in the database.
  - access the data of the underlying tables through a view
- Usage
  - Provide an abstraction layer over tables.
  - encapsulate complex queries with joins to simplify the data access.

Creating Views

- Syntax:

  -
    ```
    CREATE VIEW view_name AS
                SELECT STATEMENT;
    ```

  -
    ```
    CREATE VIEW view_name(col_name1, col_name2, …, col_namek) AS
                                           SELECT STATEMENT;
    ```
- Example:

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

- We can now use view faculty as we would a table.
- Every time the view is used, it is reconstructed.

Why use Views?

- Allow us to break down a large query.
- Make available specific category of data a particular user.
- Gives another way to think about the data.

Q. Why is it good that views are virtual?

A. If a table is changed the corresponding view is changed appropriately.

SQLite Views Example

- Step 1: Create View

```
CREATE VIEW AllStudentsView AS
  SELECT
    s.StudentId,
    s.StudentName,
    s.DateOfBirth,
    d.DepartmentName
  FROM Students AS s
    INNER JOIN Departments AS d ON s.DepartmentId = d.DepartmentId;
```

- Step 2: Visualize it as any other relation

```
SELECT * FROM AllStudentsView;
```

**example not from univdb-sqlite.db

SQLite Views

- Temporary Views:

  `CREATE TEMP VIEW`, or

  `CREATE TEMPORARY VIEW`
- View only
  - You cannot use the statements INSERT, DELETE, or UPDATE with views.
- To delete a VIEW, you can use the "DROP VIEW" statement:

  ```
  DROP VIEW AllStudentsView;
  ```

Example of View

- Consider a clerk who needs to access all data in the instructor relation, except salary. The clerk should not be authorized to access the instructor relation.
- Instead, a view relation faculty can be made available to the clerk, with the view defined as follows:

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

Example of View

- Create a view that lists all course sections offered by the Physics department in the Fall 2017 semester with the building and room number of each section,

```
CREATE VIEW physics_fall_2017 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 = 2017;
```

- Using view:

```
select course_id
from physics_fall_2017 where building = 'Watson';
```

Example of View

- For each department find the sum of the salaries of all the instructors at that department.

```
CREATE VIEW dept_total_salary(dept_name, tot_salary) AS
SELECT dept_name, sum(salary)
FROM Instructor
GROUP BY dept_name;
```

- Using view:

```
select course_id
from physics_fall_2017 where building = 'Watson';
```

Example of View in other view

- Define a view that lists the course_id and room number of Physics courses offered in Fall 2017 in Watson building.

```
CREATE VIEW physics_fall_2017_Watson AS
SELECT course_id, room_number
FROM physics_fall_2017
WHERE building = 'Watson';
```

Next week

More SQL and Introduction to Web Development..