Week 4

More SQL Queries

Topics covered Last week

- Intro to SQL
- Mapping Relational Algebra to SQL queries
- Focused on queries to start – assumed tables and database exist.

Topics covered this week

- Creating tables, setting constraints…
- Inserting and updating tables
- More query commands

- Creating views
- Joins
  - Left
  - Right
  - Full
- More on NULL values

Data Definition

- Create Table – show you how to create a new table in the database.
- 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.

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(valu1, value2, ….);

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

Constraints

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

Drop, Alter Table

- **Drop Table**

```
DROP TABLE students;
```

- **Alter Table to rename:**

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

- **Alter Table to add or rename column:**

```
ALTER TABLE students ADD COLUMN age INT;
```

```
ALTER TABLE students
RENAME COLUMN Address to Streets;
```

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;
```

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;
```

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;
```

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;
```

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.

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.

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;
```

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

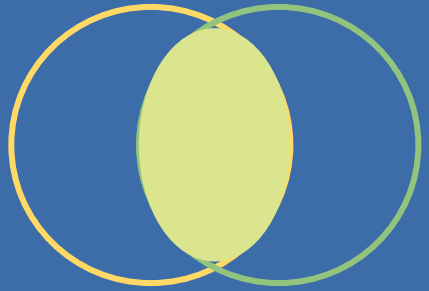
Joining Tables SQL

Join

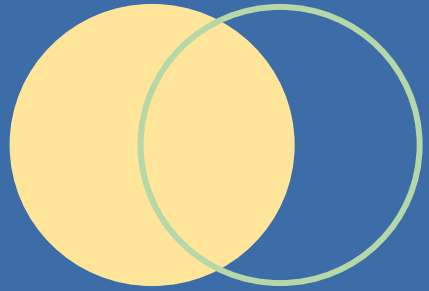
- A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
- INNER JOIN: Returns records that have matching values in both tables
- LEFT JOIN: Returns all records from the left table, and the matched records from the right table
- RIGHT JOIN: Returns all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

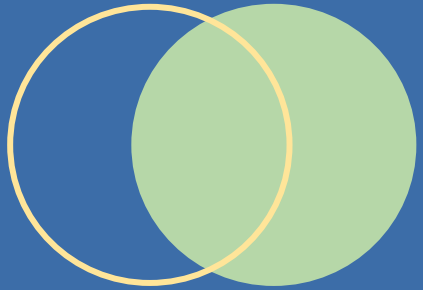
(INNER) JOIN



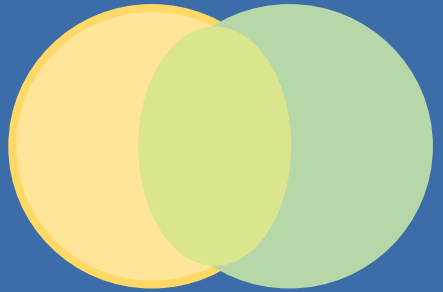
LEFT JOIN



RIGHT JOIN



FULL JOIN



INNER JOIN

- INNER JOIN clause returns rows from the A table that has the corresponding row in B table.
- selects records that have matching values in both tables.

```
SELECT *
FROM student
INNER JOIN takes
ON student.id = takes.id;
```

Q. What about students who have not yet taken any courses?

A. They are left out.

OUTER JOIN

- Use OUTER JOINS to prevent this loss of information.
- The LEFT OUTER JOIN preserves tuples only in the relation to the left of the JOIN.
- The RIGHT OUTER JOIN preserves tuples only in the relation to the right of the JOIN.
- The FULL OUTER JOIN preserves tuples in both relations.

****Note****

Note that SQLite doesn’t directly support the RIGHT JOIN and FULL OUTER JOIN

You can emulate SQLite full outer join using the UNION and LEFT JOIN clauses.

Natural JOIN

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL JOIN S

A	B	C
1	2	3

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

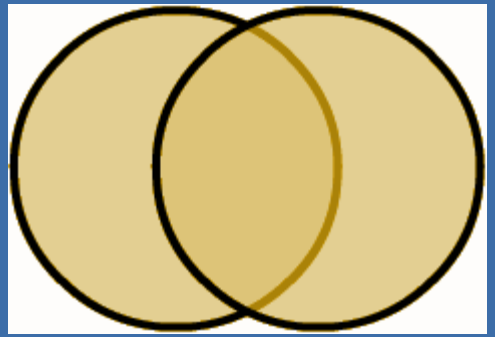
B	C
2	3
6	7

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
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
  Students.StudentName,
  Departments.DepartmentName
FROM Students
INNER JOIN Departments ON Students.DepartmentId = Departments.DepartmentId;
```

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
  Students.StudentName,
  Departments.DepartmentName
FROM Students
INNER JOIN Departments USING(DepartmentId);
```

SQLite Natural JOIN

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

```
SELECT
  Students.StudentName,
  Departments.DepartmentName
FROM Students
Natural JOIN Departments;
```

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
  Students.StudentName,
  Departments.DepartmentName
FROM Students          -- this is the left table
LEFT JOIN Departments ON Students.DepartmentId = Departments.DepartmentId;
```

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
  Students.StudentName,
  Departments.DepartmentName
FROM Students
CROSS JOIN Departments;
```

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;
```

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.DepartmentName, COUNT(s.StudentId) AS StudentsCount
FROM Students AS s
INNER JOIN Departments AS d ON s.DepartmentId = d.DepartmentId
GROUP BY d.DepartmentName;
```

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

Total number of students present in each department

```
SELECT d.DepartmentName, COUNT(s.StudentId) AS StudentsCount
FROM Students AS s
INNER JOIN Departments AS d ON s.DepartmentId = d.DepartmentId
GROUP BY d. DepartmentName;
HAVING COUNT(s.StudentId) = 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

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

- 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;
```

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;

Set Operators:
Union, Except, Intersect

UNION

To combine rows from two or more queries into a single result set, you use SQLite UNION operator.

```
query_1
UNION [ALL]
query_2
UNION [ALL]
query_3
...;
```

EXCEPT

EXCEPT operator compares the result sets of two queries and returns distinct rows from the left query that are not output by the right query.

The following shows the syntax of the EXCEPT operator:

```
SELECT select_list1
FROM table1
EXCEPT
SELECT select_list2
FROM table2
```

This query must conform to the following rules:

First, the number of columns in the select lists of both queries must be the same.
Second, the order of the columns and their types must be comparable.

INTERSECT

INTERSECT operator compares the result sets of two queries and returns distinct rows that are output by both queries.

The following illustrates the syntax of the INTERSECT operator:

```
SELECT select_list1
FROM table1
INTERSECT
SELECT select_list2
FROM table2
```

The basic rules for combining the result sets of two queries are as follows:

First, the number and the order of the columns in all queries must be the same.
Second, the data types must be comparable.