Spring 2024 CSE 380

1-30-2024

Quiz Info

- Quiz 2 is moved to Feb 6th, the syllabus is updated
- Quiz 1 Stats:
  - – Average: 50/65
  - – High: 64/65 (2)
    - 63/64 (4)
  - – Modes: 56/65, 63/65
- Remember there is a 50% requirement on quiz total

Review

When is a Left Outer Join the Same as an Inner Join?

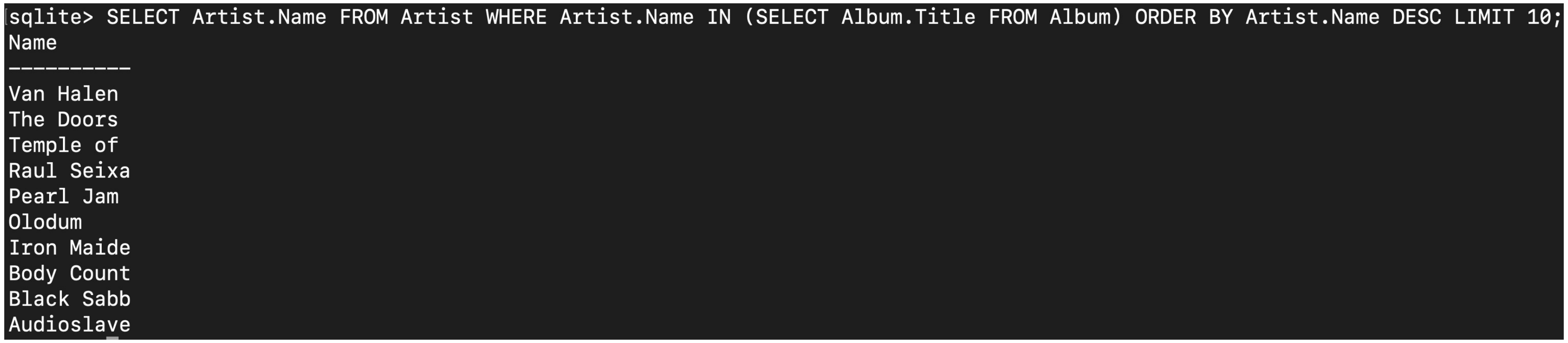
- Never
- When the left rows always have a match
- When the left rows don't have NULLs
- Always

SQL Subqueries

- A subquery is a SQL query within a query.
- Subqueries are nested queries that provide data to the enclosing query.
- Subqueries can return individual values or a list of records
- Subqueries must be enclosed with parenthesis
- If I wanted all the Artists who released an Album with just their own name:

```
SELECT Artist.Name
FROM Artist
WHERE Artist.Name IN (SELECT Album.Title FROM
Album);
```

SQL Subqueries



```
sqlite> SELECT Artist.Name FROM Artist WHERE Artist.Name IN (SELECT Album.Title FROM Album) ORDER BY Artist.Name DESC LIMIT 10;
Name
----------
Van Halen
The Doors
Temple of
Raul Seixa
Pearl Jam
Olodum
Iron Maide
Body Count
Black Sabb
Audioslave
```

```
SELECT Artist.Name
FROM Artist
WHERE Artist.Name IN (SELECT Album.Title FROM
Album);
```

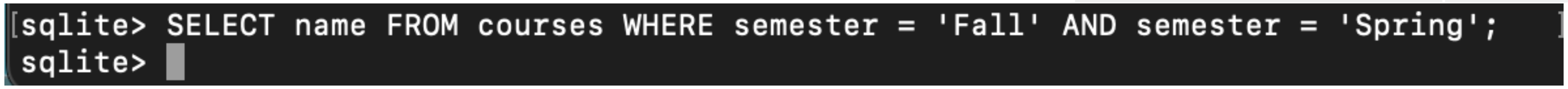
SQL Subqueries

- Subqueries often used to perform tests for set membership, make set comparisons, and determine set cardinality

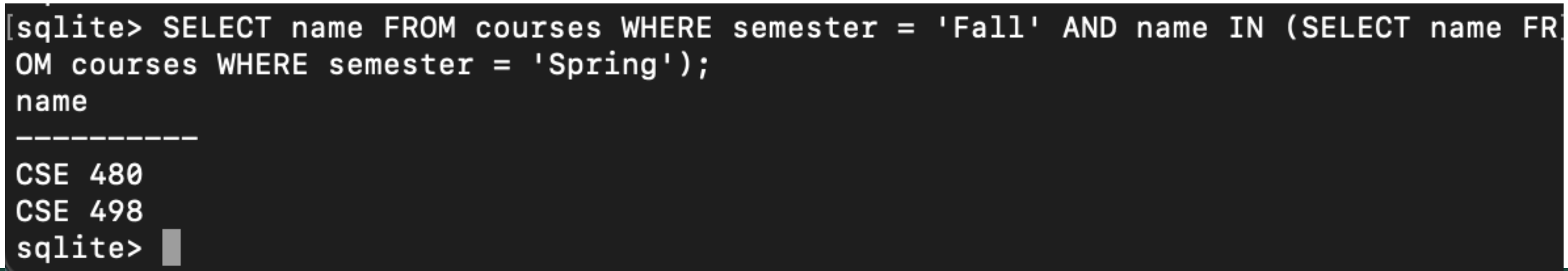
SELECT name FROM courses WHERE
  semester = 'Fall' AND semester = 'Spring';

SELECT name FROM courses WHERE semester
  = 'Fall' AND name in (SELECT name FROM
  courses WHERE semester = 'Spring');

name	semester
CSE 480	Fall
CSE 480	Spring
CSE 498	Fall
CSE 498	Spring
CSE 422	Fall
CSE 476	Fall



```
sqlite> SELECT name FROM courses WHERE semester = 'Fall' AND semester = 'Spring';
sqlite>
```

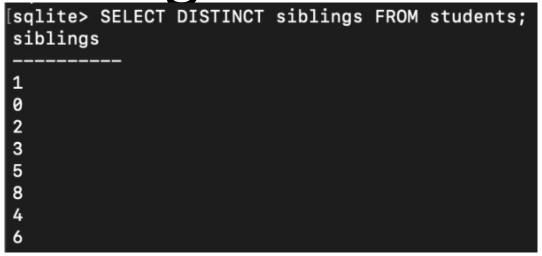


```
sqlite> SELECT name FROM courses WHERE semester = 'Fall' AND name IN (SELECT name FR
OM courses WHERE semester = 'Spring');
name
----------
CSE 480
CSE 498
sqlite>
```

DISTINCT

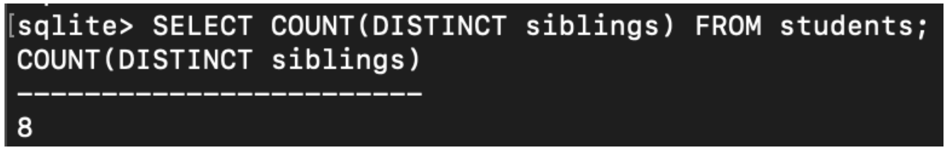
- SELECT DISTINCT returns only distinct (different) values.
- SELECT DISTINCT eliminates duplicate records from the results.
- DISTINCT can be used with aggregates: COUNT, AVG, MAX, etc.
- DISTINCT operates on a single column. DISTINCT for multiple columns is not supported.

SELECT DISTINCT siblings FROM students;



```
sqlite> SELECT DISTINCT siblings FROM students;
siblings
----------
1
0
2
3
5
8
4
6
```

SELECT count(DISTINCT siblings) FROM students;

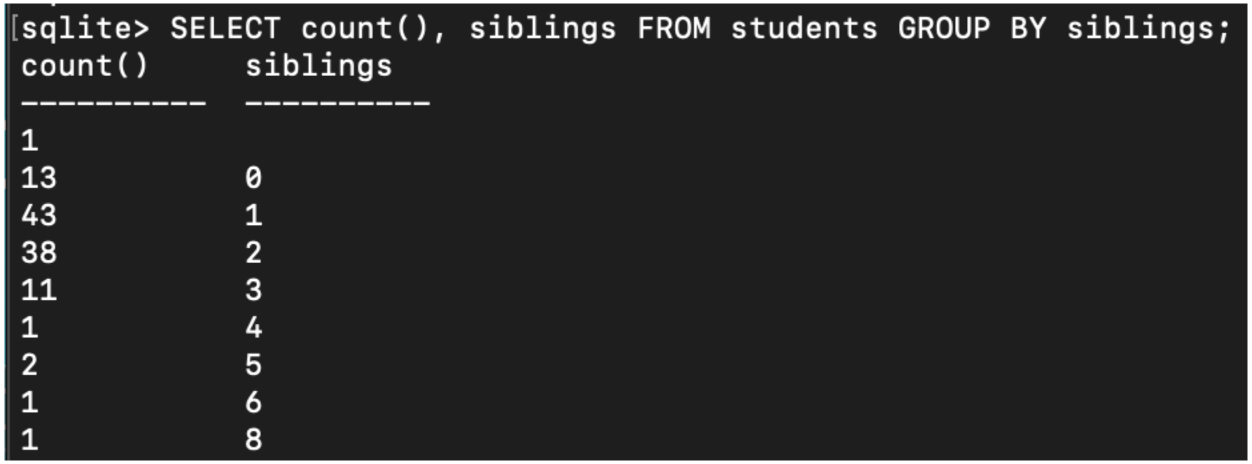


```
sqlite> SELECT COUNT(DISTINCT siblings) FROM students;
COUNT(DISTINCT siblings)
------------------------
8
```

GROUP BY

- The GROUP BY clause groups records into summary rows.
- GROUP BY returns one records for each group.
- GROUP BY typically also involves aggregates: COUNT, MAX, SUM, AVG, etc.
- GROUP BY can group by one or more columns.

SELECT count(), siblings FROM students GROUP BY siblings;



```
[sqlite> SELECT count(), siblings FROM students GROUP BY siblings;
count()     siblings
----------  ----------
1
13          0
43          1
38          2
11          3
1           4
2           5
1           6
1           8
```

All the parts of a SELECT query

- SELECT column-names
- FROM table-name
- WHERE condition
- GROUP BY column-names
- HAVING condition
- ORDER BY column-names
- LIMIT max-rows
- ;

New Material

CRUD

- Four fundamental operations that apply to any database are:
  - Read the data -- *SELECT*
  - Insert new data -- *INSERT*
  - Update existing data -- *UPDATE*
  - Remove data -- *DELETE*
- Collectively these are referred to as **CRUD** (Create, Read, Update, Delete).

INSERT INTO

- The INSERT INTO statement is used to add new data to a database.
- The INSERT INTO statement adds a new record to a table.
- INSERT INTO can contain values for some or all of its columns.
- INSERT INTO can be combined with a SELECT to insert records.

```
INSERT INTO spelling_team, (first_name, spelling)
  VALUES ('James', 10);
```

INSERT Multiple Values

- You can insert multiple rows with a single INSERT statement:

```
INSERT INTO spelling_team (first_name, spellings)
VALUES ('James', 10),
           (Abigail', 9);
```

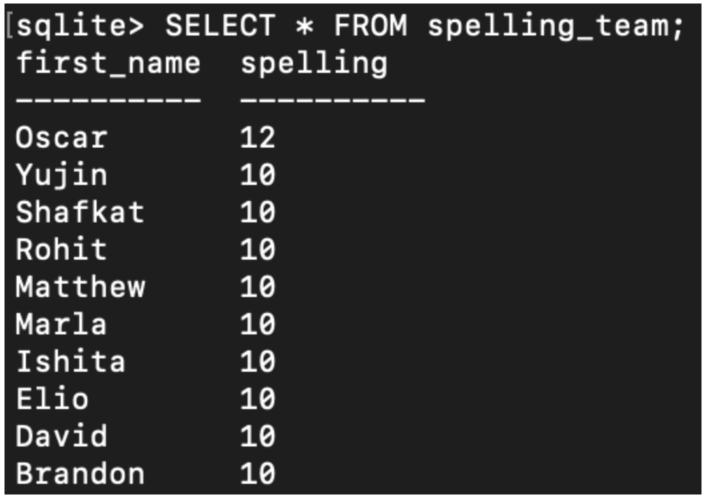
INSERT INTO SELECT

- You can also use a SELECT clause to generate the needed rows, but you need to return the correct column types and order.

INSERT INTO spelling_team (first_name, spelling)

SELECT students.first_name, students.spelling

FROM students ORDER BY spelling DESC LIMIT 10;

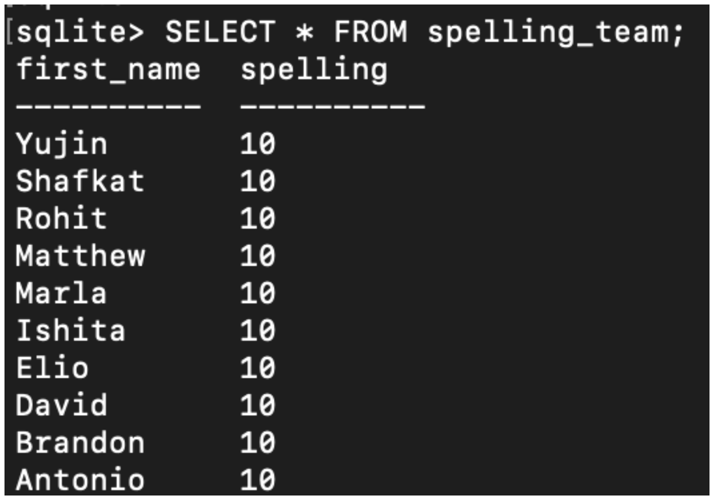


```
sqlite> SELECT * FROM spelling_team;
first_name  spelling
----------  ----------
Oscar       12
Yujin       10
Shafkat     10
Rohit       10
Matthew     10
Marla       10
Ishita      10
Elio        10
David       10
Brandon     10
```

UPDATE

- The UPDATE statement updates data values in a database.
- UPDATE can update one or more records in a table.
- Use the WHERE clause to UPDATE only specific records.

```
UPDATE students SET spelling = 0 WHERE spelling >10;
```



```
sqlite> SELECT * FROM spelling_team;
first_name  spelling
----------  ----------
Yujin       10
Shafkat     10
Rohit       10
Matthew     10
Marla       10
Ishita      10
Elio        10
David       10
Brandon     10
Antonio     10
```

UPDATE

- The UPDATE statement updates data values in a database.
- UPDATE can update one or more records in a table.
- Use the WHERE clause to UPDATE only specific records.

```
UPDATE students SET spelling = 0 WHERE spelling >10;
UPDATE students SET fav_word = fav_word || '!';

|| is concatenate in SQL
```

DELETE

- DELETE permanently removes records from a table.
- DELETE can delete one or more records in a table.
- Use the WHERE clause to DELETE only specific records.

```
DELETE FROM spelling_team; -- removes all rows

DELETE FROM spelling_team WHERE got_answer_right = 'False';
--removes some rows
```

AUTOINCREMENT

- SQLite AUTOINCREMENT is a keyword used for auto incrementing a value of a field in the table. We can auto increment a field value by using AUTOINCREMENT keyword when creating a table with specific column name to auto incrementing it.
- The keyword AUTOINCREMENT can be used with a INTEGER PRIMARY KEY field only.
- CREATE TABLE table_name (column1 INTEGER PRIMARY KEY AUTOINCREMENT, column2 datatype, column3 datatype, ... columnN datatype);

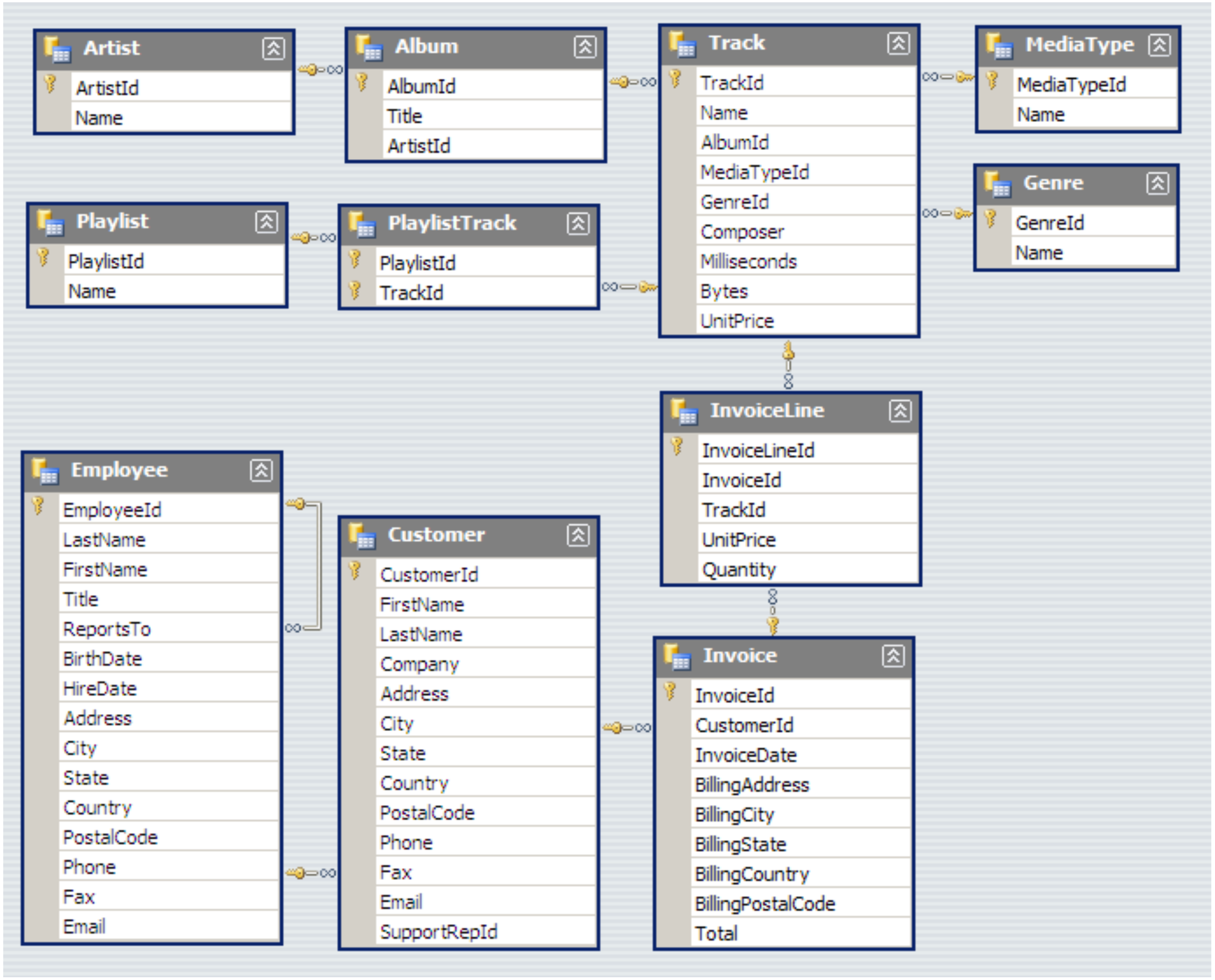
Should you use it?

- From https://www.sqlite.org/autoinc.html:

  "The AUTOINCREMENT keyword imposes extra CPU, memory, disk space, and disk I/O overhead and should be avoided if not strictly needed. It is usually not needed."
- If a column is INTEGER PRIMARY KEY, it already will autofill a unique value if a value isn't provided.
  - The only difference is AUTOINCREMENT guarantees monotonically increasing values, instead of just unique.

Practice

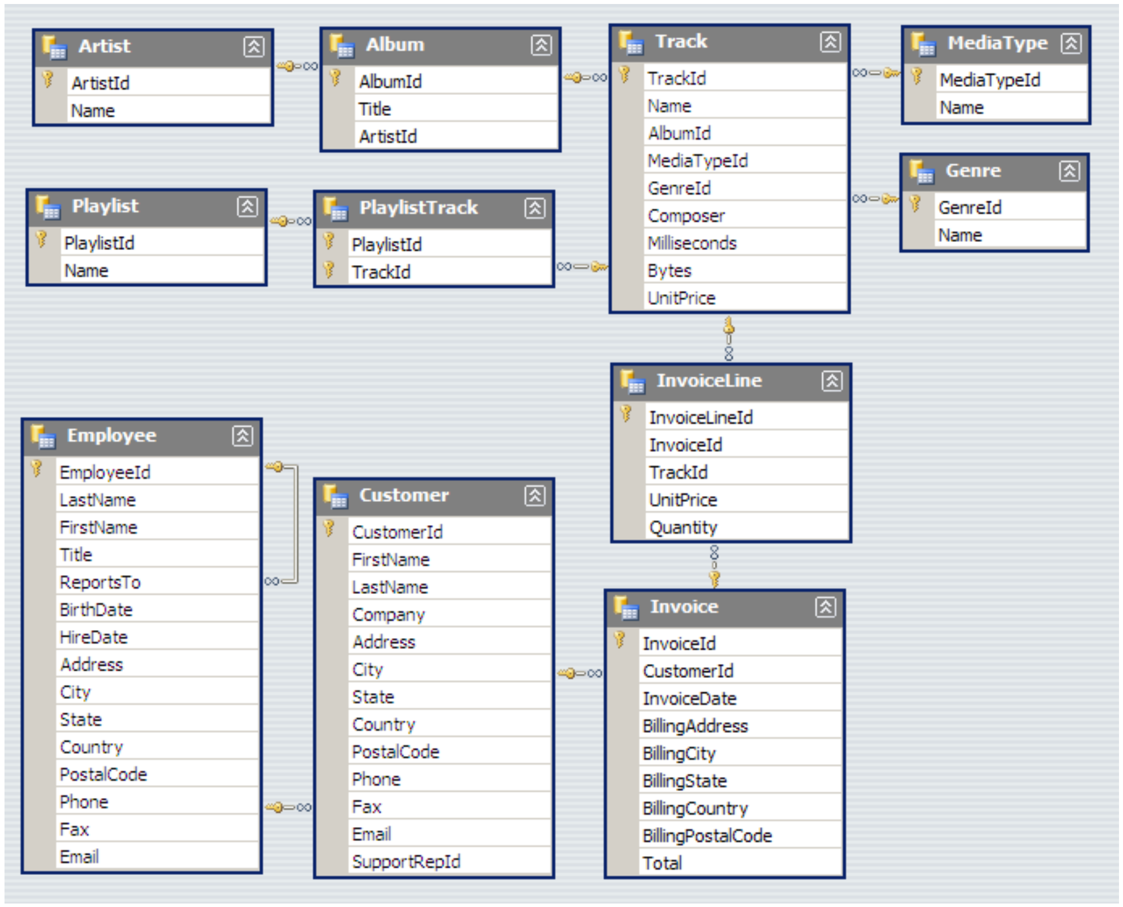
Chinook Dataset



Test Databases

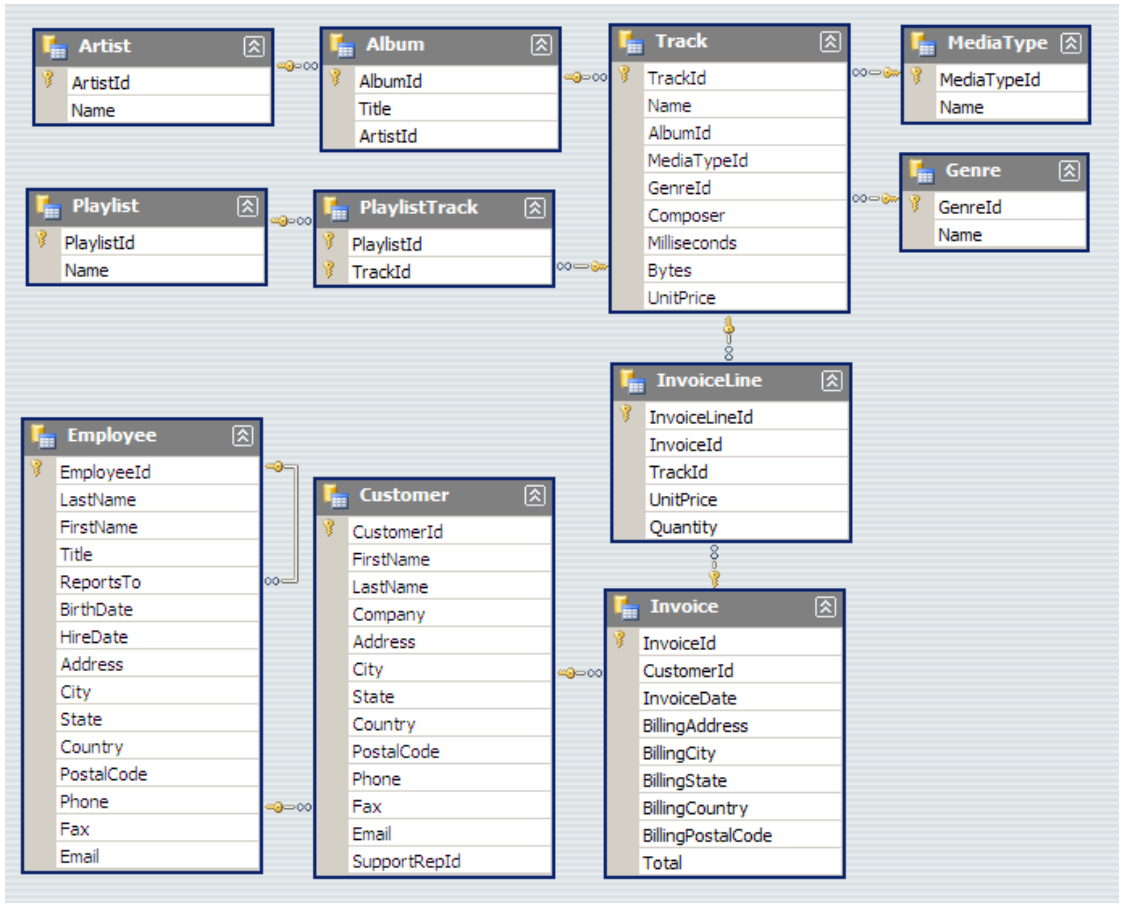
- Chinook Music Database
  - Used for examples
- Create chinook DB instance - .sql file on D2L

```
sqlite3 chinook.db < Chinook_Sqlite.sql
```



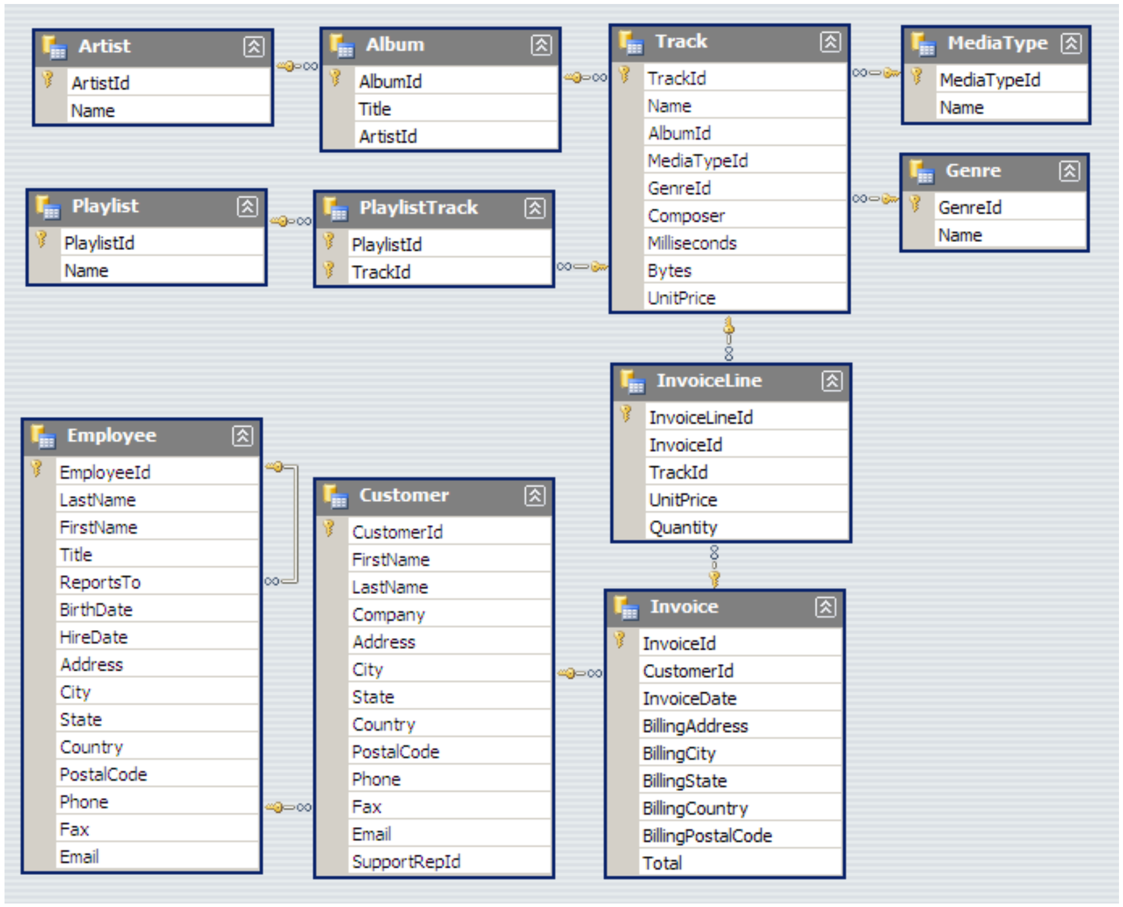
Practice

Provide a query that shows the total sales per country in order from most to least sales



Practice

Provide a query that shows the number of customers assigned to each sales agent



Practice

Provide a query that shows the total number of tracks in each playlist, and includes the playlist name in the result

Misc. SQL Table Operations

- CREATE TABLE IF NOT EXISTS <table>;
  - Do not throw an error if you try to create a table that already exists
- DROP TABLE <table name>;
  - Remove the entire table and all records of the table
- DROP TABLE IF EXISTS <table name>;
  - Same, but don’t throw an error if the table doesn’t exist

More Table Constraints

tracks

id	name	artist_id
1	Let it be	1
2	Penny Lane	1
3	Yellow Submarine	1
4	Hedwig's Theme	2
5	Jingle Bell Rock	NULL
6	The Devil Is A Patient Man	3

tracks.artist_id is a foreign key referencing artists.id

artists

id	name
1	The Beatles
2	John Williams
3	CRUD

Foreign Keys

- We discussed primary keys, which are columns that uniquely identify each row.
- However, often our tables will have columns that are meant to match up with columns in a different table.
- We want to add a constraint on those columns that there must be an associated row in a foreign (other) table.
  - NULLs are okay

Foreign Key Syntax

```
CREATE TABLE artists (id INTEGER, name
TEXT);
CREATE TABLE tracks (
    id INTEGER,
    name TEXT,
    artist_id INTEGER,
    FOREIGN KEY (artist_id) REFERENCES
artists(id)
);
```

Foreign Key Efficiency

- With a foreign key, it is an error to change the database in a way which makes a row not match with a foreign row.
  - This means insert, update, and delete statements must all be checked.
- This adds a heavy cost to changing the database.
- SQLite by default doesn't check foreign keys for correctness.
- You need to turn on this functionality with:
  - `PRAGMA foreign_keys = ON;`

Example

```
PRAGMA foreign_keys = ON;
INSERT INTO artists (id, name) VALUES (1, 'Beatles');
INSERT INTO tracks (id, name, artist_id) VALUES
      (1, 'Jingle Bell Rock', NULL), -- OKAY
      (2, 'Let it be', 1), -- Okay
      (3, 'Jurassic Park', 2); -- ERROR: no matching key
```

When should you use a foreign key constraint?

- When you need to enforce matching values
- When you refer to values in a different table
- Never

MISC SQL Stuff

SELECT * and Qualified names

- SELECT * is shorthand for all the columns (in order) in the table.
- * can be qualified (i.e. students.*) to make it unambiguous which table's columns are being included.

Parameterized Queries

```
conn = sqlite3.connect(":memory:")
conn.execute("CREATE TABLE students (name TEXT, age INTEGER);")
conn.execute("INSERT INTO students VALUES ('James’, 30);")
```

What if we wanted to add a python integer?

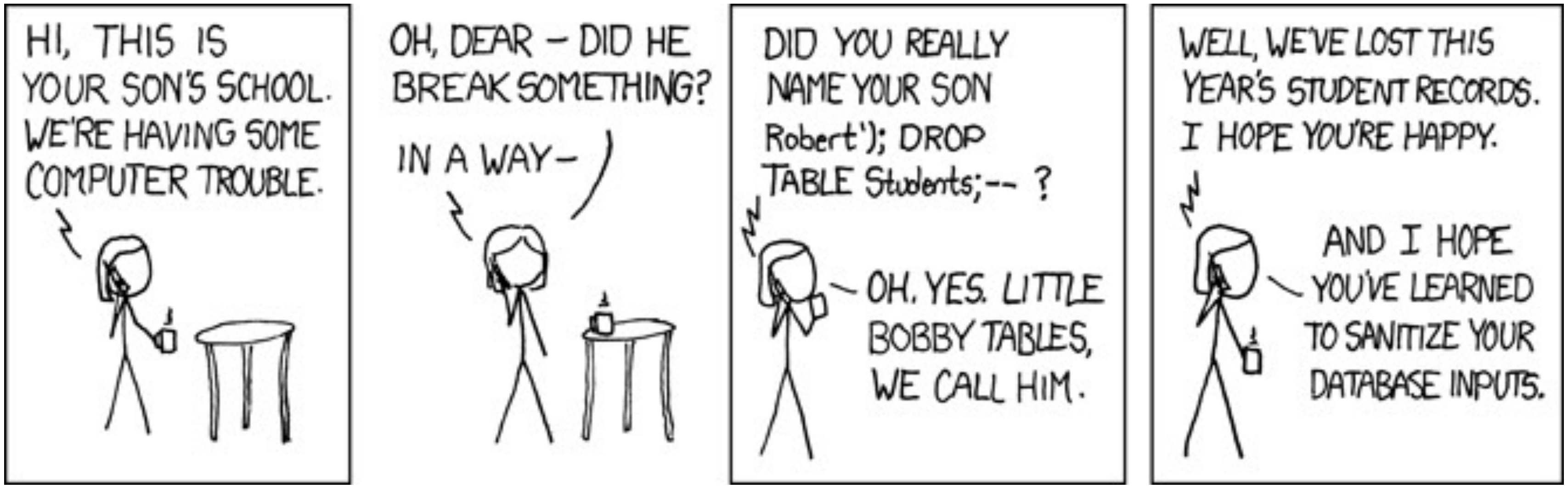
```
Steve_age = 23
conn.execute( "INSERT INTO students VALUES ('Steve', " + str(steve_age) + ");" )
```

Parameterized Queries

- Used to pass python objects into queries without needing to manually convert to strings.

```
Steve_age = 23
conn.execute("INSERT INTO students VALUES ('Steve', ?);", (steve_age,))
row = ('Tim', 45)
conn.execute("INSERT INTO students VALUES (?, ?);", row)
```

Why do we care?


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

https://xkcd.com/327/

Explanation

- The problem is if we wrote code like this:

```
name = "Robert'); DROP TABLE students; --"
conn.execute("INSERT INTO students VALUES ('"
            + name + "');")
# INSERT INTO students VALUES ('Robert');
# DROP TABLE students; --');
```

The input would be allowed to execute arbitrary queries against the database.

Protection by Parameterized Query

- Parameterized queries will automatically escape the input and ensure that the value passed in is store as that value.

```
name = "Robert'); DROP TABLE students; --"
conn.execute("INSERT INTO students VALUES (?);", (name,))
```

- The string name will be stored, in its entirety, and the single quote will be escaped to stop it from harming the database.

Python and SQLite

SQLite to Python

SQLite type	Python type
NULL	None
INTEGER	int
REAL	float
TEXT	str
~~BLOB~~	~~bytes~~

Connect

- connect is a module function that takes a filename and results a connection object
  - .connect(“test.db”)
  - .connect(:memory:)

Cursor

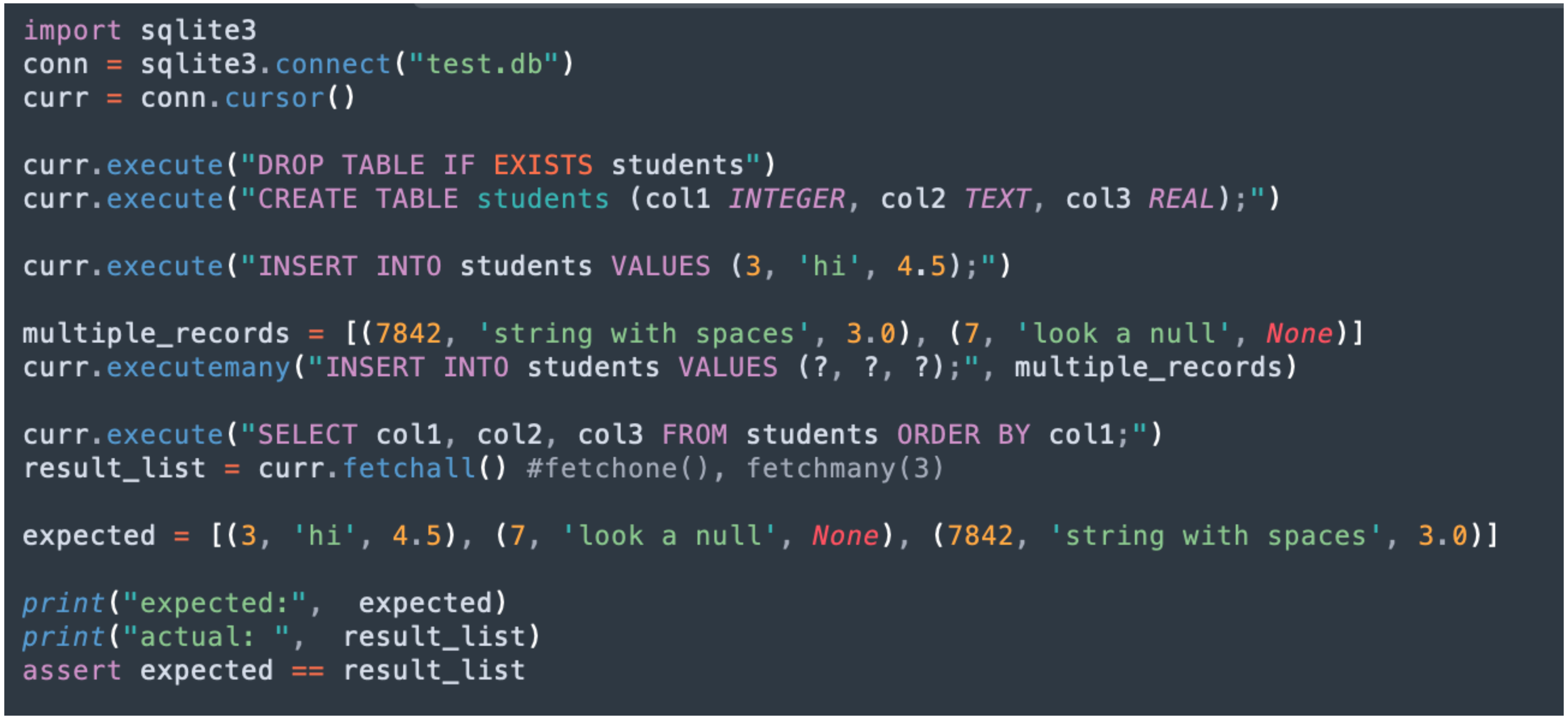
- A Cursor object represents a database cursor (not important what that means), it is where we’ll execute SQL statements

Execute

- The execute method on the Cursor object takes a string (query) and returns an iterable object or None, depending on if the query returns rows from the database.

Example Code

- This will be posted online



```python
import sqlite3
conn = sqlite3.connect("test.db")
curr = conn.cursor()

curr.execute("DROP TABLE IF EXISTS students")
curr.execute("CREATE TABLE students (col1 INTEGER, col2 TEXT, col3 REAL);")

curr.execute("INSERT INTO students VALUES (3, 'hi', 4.5);")

multiple_records = [(7842, 'string with spaces', 3.0), (7, 'look a null', None)]
curr.executemany("INSERT INTO students VALUES (?, ?, ?);", multiple_records)

curr.execute("SELECT col1, col2, col3 FROM students ORDER BY col1;")
result_list = curr.fetchall() #fetchone(), fetchmany(3)

expected = [(3, 'hi', 4.5), (7, 'look a null', None), (7842, 'string with spaces', 3.0)]

print("expected:",  expected)
print("actual: ",  result_list)
assert expected == result_list
```

Online Tutorial

- I haven't watched all of this, but the general idea seems to be good
  - You can watch this or other online tutorials for more help
- https://www.youtube.com/watch?v=byHcYRpMgI4&ab_channel=freeCodeCamp.org

That’s it for today

- Questions?