Structured English Query Language

Basics

- **Creating a table and inserting data**

/** Grocery list:
Bananas (4)
Peanut Butter (1)
Dark Chocolate Bars (2)
**/

CREATE TABLE groceries (id INTEGER PRIMARY KEY, name TEXT, quantity INTEGER);

INSERT INTO groceries VALUES (1, "Bananas", 4);
INSERT INTO groceries VALUES (2, "Peanut Butter", 1);
INSERT INTO groceries VALUES (3, "Dark chocolate bars", 2);

- **Querying the table**

SELECT *class name(* if all)* FROM *table name* WHERE *condition(ex: class name>5)* ORDER BY *class name* ASC/DESC;

- **Aggregating data**

Total number: SELECT SUM(*class name*) FROM *table name*;
Max number: SELECT MAX(*class name*) FROM *table name*;
Total number grouped by different classes:
SELECT *class name1*, SUM(*class name 2*) FROM *table name* GROUP BY *class name 1*;

More complex SQL

- **More complex queries with AND/OR**

CREATE TABLE exercise_logs
(id INTEGER PRIMARY KEY AUTOINCREMENT, (so don't need to insert id)
type TEXT,
minutes INTEGER,
calories INTEGER,
heart_rate INTEGER);

INSERT INTO exercise_logs(type, minutes, calories, heart_rate) VALUES ("biking", 30, 100, 110);
INSERT INTO exercise_logs(type, minutes, calories, heart_rate) VALUES ("biking", 10, 30, 105);
INSERT INTO exercise_logs(type, minutes, calories, heart_rate) VALUES ("dancing", 15, 200, 120);

/* AND */
SELECT * FROM exercise_logs WHERE calories > 50 AND minutes < 30;

```
/* OR */
SELECT * FROM exercise_logs WHERE calories > 50 OR heart_rate > 100;
```

- **Querying IN subqueries**

```
SELECT * FROM exercise_logs WHERE type (NOT) IN ("biking", "hiking", "tree climbing",
"rowing");


CREATE TABLE drs_favorites
   (id INTEGER PRIMARY KEY,
   type TEXT,
   reason TEXT);

INSERT INTO drs_favorites(type, reason) VALUES ("biking", "Improves endurance and
flexibility.");
INSERT INTO drs_favorites(type, reason) VALUES ("hiking", "Increases cardiovascular
health.");
```

Show the type in table exercise_logs that matches with type in drs_favorites:

```
SELECT * FROM exercise_logs WHERE type IN (
SELECT type FROM drs_favorites);
```

Show the activity with reason:

```
SELECT * FROM exercise_logs WHERE type IN (
   SELECT type FROM drs_favorites WHERE reason = "Increases cardiovascular health.");
```

Don’t have to copy the reason, but contain the keyword: LIKE”%%”

```
SELECT * FROM exercise_logs WHERE type IN (
   SELECT type FROM drs_favorites WHERE reason LIKE "%cardiovascular%");
```

- **Restricting grouped results with HAVING**

Get the sum of calories for each exercise and give this column a new name:

```
SELECT type, SUM(calories) AS total_calories FROM exercise_logs GROUP BY type;
```

Get exercise where sum of calories more than 150: Having

```
SELECT type, SUM(calories) AS total_calories FROM exercise_logs
   GROUP BY type
   HAVING total_calories > 150;
```

(When using HAVING, we’re applying the conditions to the grouped values, not the individual ones)

Get exercise appeared equal or more than 2 times in the table: COUNT()

```
SELECT type FROM exercise_logs GROUP BY type HAVING COUNT(*) >= 2;
```

- **Calculating results with CASE**

See the number of exercise times where I have a heart rate of more than 220-30:

```
SELECT COUNT(*) FROM exercise_logs WHERE heart_rate > 220 - 30;
```

Add a case to the table: CASE WHEN THEN ELSE END as

```
SELECT type, heart_rate,
   CASE
      WHEN heart_rate > 220-30 THEN "above max"
      WHEN heart_rate > ROUND(0.90 * (220-30)) THEN "above target"
      WHEN heart_rate > ROUND(0.50 * (220-30)) THEN "within target"
      ELSE "below target"
   END as "hr_zone"
FROM exercise_logs;
```

Count the number of times according to case condition:

```
SELECT COUNT(*),
   CASE
      WHEN heart_rate > 220-30 THEN "above max"
      WHEN heart_rate > ROUND(0.90 * (220-30)) THEN "above target"
      WHEN heart_rate > ROUND(0.50 * (220-30)) THEN "within target"
      ELSE "below target"
   END as "hr_zone"
FROM exercise_logs
GROUP BY hr_zone;
```

If only need to calculate values based on the existing ones:

```
   SELECT name, number_grade,ROUND(fraction_completed*100) AS percent_completed
   FROM student_grades;
```

Splitting data into related tables

- **JOINing related tables**

```
/* cross join */
SELECT * FROM student_grades, students;
/* implicit inner join */
SELECT * FROM student_grades, students
   WHERE student_grades.student_id = students.id;
/* explicit inner join - JOIN */
SELECT students.first_name, students.last_name, students.email, student_grades.test,
student_grades.grade FROM students
  JOIN student_grades
  ON students.id = student_grades.student_id;
```

- **Joining related tables with left outer joins**

```
/* outer join */
SELECT students.first_name, students.last_name, student_projects.title
   FROM students
   LEFT OUTER JOIN student_projects
   ON students.id = student_projects.student_id;
```

Include everything from the left table and join everything into the right table even if there're no items on the right (also RIGHT OUTER JOIN, FULL OUTER JOIN)

- **Joining tables to themselves with self-joins**

```
/* self join */
SELECT students.first_name, students.last_name, buddies.email as buddy_email
  FROM students
  JOIN students buddies
  ON students.buddy_id = buddies.id;
```

- **Combining multiple joins**

```
SELECT a.title, b.title FROM project_pairs
  JOIN student_projects a
  ON project_pairs.project1_id = a.id
  JOIN student_projects b
  ON project_pairs.project2_id = b.id;
```

Show friends of each other:

```
SELECT a.fullname, b.fullname FROM friends
   JOIN persons a
   ON a.id=person1_id
   JOIN persons b
   ON b.id=person2_id;
```

- **query tuning**

identify what queries you want to tune
understand how a particular SQL engine is executing a query--EXPLAIN QUERY PLAN
manual optimization to improve that execution plan (ex. creating indexes)

Modifying databases with SQL

- **Changing rows with UPDATE and DELETE**

```
UPDATE diary_logs SET content = "I had a horrible fight with OhNoesGuy" WHERE id = 1;
DELETE FROM diary_logs WHERE id = 1;
```

- **Altering tables after creation**

```
ALTER TABLE diary_logs ADD emotion TEXT default "unknown";
DROP TABLE diary_logs;
```

- **Make your SQL safer**

Avoiding bad updates/deletes

Before you issue an UPDATE, run a SELECT with the same WHERE to make sure you're updating the right column and row.

For example, before running:

```
UPDATE users SET deleted = true WHERE id = 1;
```

You could run:

```
SELECT id, deleted FROM users WHERE id = 1;
```

Once you decide to run the update, you can use the LIMIT operator to make sure you don't accidentally update too many rows:

UPDATE users SET deleted = true WHERE id = 1 LIMIT 1;
Or if you were deleting:
DELETE users WHERE id = 1 LIMIT 1;

Using transactions
Whenever we issue a command like CREATE, UPDATE, INSERT, or DELETE, we are automatically starting a transaction.
an UPDATE to go through if another UPDATE goes through as well (also used to guarantee that no other transactions happen in the middle):

```
BEGIN TRANSACTION;
UPDATE people SET husband = "Winston" WHERE user_id = 1;
UPDATE people SET wife = "Winnefer" WHERE user_id = 2;
COMMIT;
```

Making backups/replication

Granting privileges
As a general rule, there should be only a few users that have full access to the database (like backend engineers)
GRANT FULL ON TABLE users TO super_admin;
GRANT SELECT ON TABLE users TO analyzing_user;