**SQL**

**Creating Table**

**CREATE TABLE** Name of table (idname INTEGER PRIMARY KEY, name TEXT, quantity INTEGER);

**INSERT INTO** Name of table **VALUES**( values above);

**Querying the table**

**\* = select all**

SELECT \* FROM name of table WHERE \_\_ > # ORDER BY

**Aggregating functions**

SUM, MAX, MIN, etc

**GROUP BY:** SELECT \_\_\_, SUM(\_\_\_) FROM \_\_\_\_ GROUP BY \_\_\_\_

Example Table:

CREATE TABLE superstore (id INTEGER PRIMARY KEY, name TEXT, inventory INTEGER, price INTEGER, aisle INTEGER);

INSERT into superstore VALUES (1, “Grapes”, 200, 2.50, 1);

INSERT into superstore VALUES (2, “Halos”, 170, 4.50, 1);

INSERT into superstore VALUES (3, “Broccoli”, 56, 2.65, 2);

INSERT into superstore VALUES (4, “Spices”, 400, 1.50, 4);

INSERT into superstore VALUES (5, “Shrimp”, 50, 12.50, 7);

INSERT into superstore VALUES (6, “Pancakes”, 350, 6.00, 6);

INSERT into superstore VALUES (7, “Oatmeal”, 300, 7.00, 6);

INSERT into superstore VALUES (8, “Juice”, 700, 6.50, 5);

INSERT into superstore VALUES (9, “Eggs”, 425, 3.50, 3);

INSERT into superstore VALUES (10, “Chicken”, 275, 10.50, 7);

INSERT into superstore VALUES (11, “Cards”, 400, 1.50, 1);

INSERT into superstore VALUES (12, “Poker Set”, 200, 20.50, 1);

INSERT into superstore VALUES (13, “Milk”, 320, 4.50, 3);

INSERT into superstore VALUES (14, “Coffee”, 800, 2.50, 4);

INSERT into superstore VALUES (15, “Tea”, 700, 2.50, 2);

SELECT \* FROM superstore WHERE inventory > 300 ORDER BY price;

**Complex Query tools**

**Specifying what data to enter**

**AUTOINCREMENT: places value in table automatically**

**CREAT TABLE** exercise\_logs

(id INTEGER PRIMARY KEY **AUTOINCREMENT**,

type TEXT,

minutes INTEGER,

calories INTEGER,

heart\_rate INTEGER)

**INSERT** INTOexercise\_logs(type, minutes, calories, heart\_rate) VALUES ("biking", 30, 100, 110);

**/\* 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** = “biking” **OR type** = **“**hiking”;

**Same As: \*NOT does all except IN values\***

SELECT \* FROM exercise\_logs WHERE type (NOT) **IN** ("biking", "hiking", "tree climbing", "rowing");

**/\* LIKE \*/ Used for inexact match queries (missing a value such as a period or space)**

**%= non exact value**

SELECT \* FROM exercise\_logs WHERE type IN (

SELECT type FROM drs\_favorites WHERE reason LIKE "%cardiovascular%")

**Restricting Group results**

**AS**= renaming what comes before as **HAVING**= adding criteria to query **COUNT**= constraints to how often data needs to occur

SELECT type, SUM(calories) **AS** total\_calories FROM exercise\_logs GROUP BY type;

SELECT type, SUM(calories) **AS** total\_calories FROM exercise\_logs

**GROUP BY** type

**HAVING** total\_calories > 150

;

SELECT type FROM exercise\_logs GROUP BY type **HAVING COUNT**(\*) >= 2;

**Calculating results with CASE**

SELECT COUNT(\*) FROM exercise\_logs WHERE heart\_rate > 220 - 30;

**/\* 50-90% of max\*/**

SELECT COUNT(\*) FROM exercise\_logs WHERE

heart\_rate >= ROUND(0.50 \* (220-30))

AND heart\_rate <= ROUND(0.90 \* (220-30));

**/\* CASE \*/ CASE:** begins a multiple criteria query. **THEN:** gives criteria an output for the graph **ELSE**: if any other option outside criteria use this output END:

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

GROUP BY hr\_zone;

**Gradebook Challenge**

SELECT COUNT(\*) name,

**CASE**

WHEN number\_grade > 90 THEN "A"

WHEN number\_grade > 80 THEN "B"

WHEN number\_grade > 70 THEN "C"

ELSE "F"

END as "letter\_grade”

FROM student\_grades

GROUP BY letter\_grade;

**JOINing related tables**

**/\* cross join \*/ rarely used**

SELECT \* FROM student\_grades, students;

**/\* implicit inner join \*/ choose databases you want to open. WHERE =** database name.column = database2\_name.column

SELECT \* FROM student\_grades, students

WHERE student\_grades.student\_id = students.id;

**/\* explicit inner join - JOIN \*/ best practice. JOIN=** choosing databases to select data from**. ON=** criteria from database.column that relate and sync with another database.column

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

**Joining related tables with left outer joins**

**/\* outer join \*/ LEFT**clude all data from the left table (table after the FROM) **OUTER**= retain the rows even if there is no match

SELECT students.first\_name, students.last\_name, student\_projects.title

FROM students

**LEFT OUTER** JOIN student\_projects

ON students.id = student\_projects.student\_id;

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

**SELECT** id, first\_name, last\_name, buddy\_id FROM students;

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

**Example**

SELECT movies.title, sequel.title FROM movies

LEFT OUTER JOIN movies sequel

ON movies.sequel\_id = sequel.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

**Lifecycle of a SQL query**

1. The **query parser** makes sure that the query is syntactically correct (e.g. commas out of place) and semantically correct (i.e. the tables exist), and returns errors if not. If it's correct, then it turns it into an algebraic expression and passes it to the next step.
2. The **query planner and optimizer** does the hard thinking work. It first performs straightforward optimizations (improvements that always result in better performance, like simplifying 5\*10 into 50). It then considers different "query plans" which may have different optimizations, estimates the cost (CPU and time) of each query plan based on the number of rows in the relevant tables, then it picks the optimal plan and passes it on to the next step.
3. The **query executor** takes the plan and turns it into operations for the database, returning the results back to us if there are any.

optimize a query, and that's known as **"query tuning"**.

**Changing rows with UPDATE and DELETE**

SELECT \* FROM diary\_logs;

UPDATE diary\_logs SET content = "I had a horrible fight with OhNoesGuy" WHERE id = 1;

SELECT \* FROM diary\_logs;

DELETE FROM diary\_logs WHERE id = 1;

SELECT \* FROM diary\_logs;

**Altering tables after creation**

**default**=output for null responses. **ALTER Table** = adds a new column to the table

**ALTER TABLE** diary\_logs **ADD** emotion **TEXT** default "unknown";

**INSERT INTO** diary\_logs (user\_id, date, content, emotion) VALUES (1, "2015-04-03",

"We went to Disneyland!", "happy");

SELECT \* FROM diary\_log

**DROP TABLES** = deletes table