README

- Open tinyurl.com/select-org to code along and save as `select.org`
- Download the foods database as an SQL file from tinyurl.com/foods-sql and save it as foods.sql
- Either import the `foods` database from the commandline:

  ```
  sqlite3 foods.db < foods.sql
  ```

- Or import the database in a code block:

  ```
  .read ../data/foods.sql
  .database
  .tables
  ```

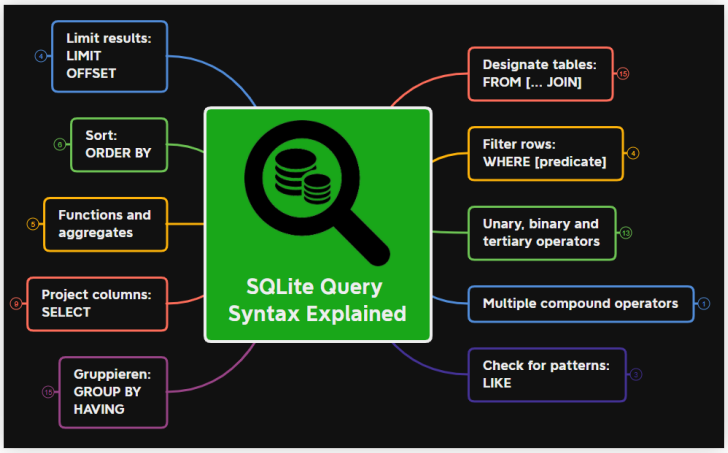
```
ls -l foods.db   # DIR foods.db
```

- If we don't finish, you'll get the updated file at the start

Content:

- Overview of the `SELECT` pipeline
- Projection and restriction/filtering
- Anatomy of the `WHERE` clause
- Values and data types
- Unary, binary, and ternary operators

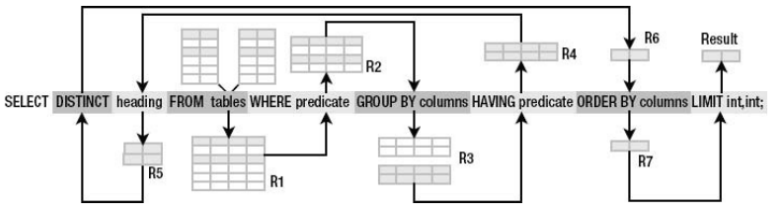
SELECT command overview



- The mindmap contains 91 examples but you need to open XMind (xmind.app) to see it, alas (GitHub copy).

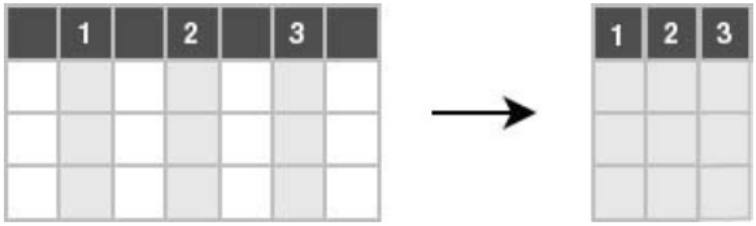
SELECT pipeline overview

- Phases of `SELECT`



SELECT column restriction (projection)

- `SELECT` is the most complex command in SQL.
- `SELECT` projects columns into a new working table.



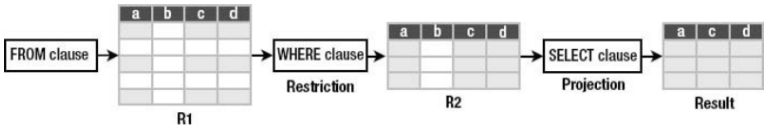

```
SELECT name FROM food_types;
```

But how many columns does this table have?

```
.schema food_types
```

`WHERE` row restriction (filtering)

- The `WHERE` clause is the most complex clause in `SELECT`.
- `SELECT` projects, and `WHERE` restricts the number of rows. In between, virtual tables R1 and R2 are created.




```
SELECT name FROM foods WHERE (id=1 OR id=2); /* restriction with Boolean expressio
```

- The argument of `WHERE` is a logical *predicate*.
- A predicate is an *assertion* about something:

  "The dog is purple and has a toothy grin."

  Here, `dog` is the **subject**, and `"color is purple"` and `"grin is toothy"` is the **predicate**.
- The logical **subject** in the `WHERE` clause is a row.
- The `WHERE` clause is the logical **predicate**.

- How would the sentence above be translated into SQL?

  SELECT * FROM dogs WHERE color='purple' AND grin='toothy';

  1. `dogs` is the table with all data about dogs
  2. `color` and `grin` are two column in `dogs`

- This statement would filter those rows that satisfy the logical proposition:

  This row has color='purple' and grin='toothy'

Values and data types

- *Values* represent data in in the real world.
- Values are classified by their type (numeric, string, date etc.)
- Values can be expressed as one of:

  1. **literals** (unchangeable) - like `1`, `2`, `3`, `"Batesville"` etc.

     ```
     SELECT "Batesville" AS 'City of';

     SELECT '1,2,3' AS "numbers";
     ```

  2. **variables** (changeable) - e.g. column names like `foods.name`

     ```
     SELECT foods.name FROM foods LIMIT 2;
     ```

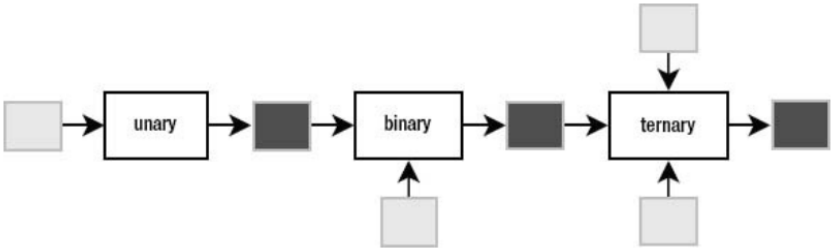
  3. **expressions** (reducible) - e.g. `3 + 2/5`

     ```
     SELECT 3+2/5;  /* without decimal point, rounding takes place */
     SELECT 3.+2./17.;
     ```

  4. **function results** - e.g. `COUNT(foods.name))`

     ```
     SELECT COUNT(foods.name) AS "No. of foods" FROM foods;
     ```

Operators



- An operator takes one or more values as input and produces a value as output.
- Unary/Binary/Ternary operators take 1/2/3 input values (operands).
- Operators can be *pipelined* and strung together forming value expressions:

```
x = count(episodes.name)
y = count(foods.name)
z = y/x * 11
```

- In `foods`, this could look like this (max 1000 subqueries in SQLite)[1]:

```
SELECT
(SELECT COUNT(name) FROM foods)/
(SELECT COUNT(name) FROM episodes) * 11;
```

- Redo this with `REAL` numbers:

```
SELECT
((SELECT COUNT(name) FROM foods) * 1.0 /
(SELECT COUNT(name) FROM episodes) ) * 11.0;
```

- Or you could create a temporary table `TempResult` to hold the result:

[1] In SQLite, the set parameter `SQLITE_MAX_EXPR_DEPTH` is 1000, not 9, and it's not just for nested SELECT queries but for the whole expression tree (source). For other flavors, the values are all over the place: SQL Server = 32 nested subqueries, Oracle = 255, and another source said "There is no limit to the number of subqueries you can use in an SQL query." MySQL and PostgreSQL have a `thread_stack` parameter that can be set freely. Either way, these set values can be changed at runtime.

```
CREATE TEMP TABLE TempResult AS
SELECT
((SELECT COUNT(name) * 1.0 FROM foods) /
(SELECT COUNT(name) FROM episodes)) * 11.0 AS result;
.tables
SELECT * FROM temp.TempResult;
```

- As a Common Table Expression (CTE):

```
WITH ComputedResult AS (
 SELECT
 ((SELECT COUNT(name) * 1.0 FROM foods) /
  (SELECT COUNT(name) FROM episodes)) * 11.0 AS result
  )
SELECT result FROM ComputedResult;
```

- SQLite has a ternary operator-like structure in the `CASE` statement:

```
SELECT
  CASE
    WHEN condition THEN true_value
    ELSE false_value
  END AS ResultColumn
FROM TableName;
```

- This SQL snippet demonstrates how to use the `CASE` statement in SQLite to mimic a ternary operator. It evaluates a condition for each row in the table `TableName`. If the condition is true, it returns `true_value`; otherwise, it returns `false_value` in the `ResultColumn`.

- For `foods`, we could for example check if a food type is sweet and print that information as a new (virtual) column `Sweet`:

```
SELECT name,
  CASE
   WHEN name='Bakery' OR name='Cereal' OR name='Fruit' THEN 'yes'
   ELSE 'no'
  END AS 'Sweet'
FROM food_types;
```

- This is *ternary* (three-way) in the sense that each row in the `name` column is input, with two output possibilities depending on the `CASE`.

Binary operators

- Binary operators (operators with two operands) are the most common ones. The table lists them by precedence from highest to lowest.

Operator	Type	Action
||	String Concatenation	
*	Arithmetic Multiply	
/	Arithmetic Divi	de
%	Arithmetic Modul	us
+	Arithmetic	Add
-	Arithmetic Subtra	ct
<<	Bitwise Right	shift
>>	Bitwise Left	shift
&	Logical	And
|	Logical Or	
<	Relational Les	sthan
<=	Relational	Less than or equal to
>	Relational Grea	ter than
>=	Relational	Greater than or equal to
=	Relational Equa	l to
==	Relational Equa	l to
<>	Relational Not	equal to

String concatenation

- String concatenation can be used to combine field values:

```
SELECT name || '(' || season || ')' AS 'Episode (#)' FROM episodes LIMIT 1,3;
```

- Is `"Episode (#)` a new table?

`.schema`

Relational operators

- These operators work even without reference to any table:

```
SELECT 1 > 2-2; /* order or precedence is observed */
SELECT 1 < 2;
SELECT 1 == 2; /* Same result as 1 = 2 */
SELECT 1 = 2; /* Same result as 1 == 2 */
```

Logical operators

- Logical operators (`AND`, `OR`, `NOT`, `IN`) are binary operators that operate no truth values or logical expressions. Here, like in C, any non-zero value is `TRUE`.
- What do you think `SELECT -1 AND 1;` will yield?

```
SELECT -1 AND 1; /* same as TRUE AND TRUE */
```

- Try `SELECT 0 AND 1` and `0 OR 1`:

```
SELECT 0 AND 1;
SELECT 0 OR 1;
```

- Bonus assignment: prove De Morgan's laws[2] using `SELECT`:

```
SELECT NOT(1 OR 0);
SELECT (NOT 1 AND NOT 0);
```

- What if you wanted values in a particular range, e.g. `id \in [3,8)`

```
SELECT id, name FROM foods WHERE 3 <= id AND id < 8; /* id in [3,8) */
```

- What if you wanted values outside of the range `id \in [3,8)`

```
SELECT id, name FROM foods WHERE  id < 3 OR 8 <= id LIMIT 5; /* id NOT in [3,8) */
```

- What if you wanted to know if there is a `Junkfood` (`type_id = 9`) named `JujyFruit`?

```
SELECT * FROM foods WHERE name='JujyFruit' AND type_id=9;
```

- The `IN` operator is used to check if a value matches a value in a list or a subquery result. For example: how many `foods` are chicken (`type_id=3`) or meat (`type_id=10`)?

```
SELECT COUNT() AS "Chicken or meat" FROM foods WHERE type_id IN (3,10);
```

[2] The negation of a disjunction is the conjunction of the negations, and the negation of a conjunction is the disjunction of the negations.

The LIKE and GLOB operators

- The LIKE operator is used to match string values against patterns (like grep).
- Say you wanted to know all foods whose name begins with a J:

```
SELECT id, name FROM foods WHERE name LIKE 'J%';
```

- Here, a percent symbol % matches any sequence of zero or more characters in the string.
- An underscore symbol _ matches any single character in the string.
- Example: find all foods that have ac and P somewhere in the name.

```
SELECT id, name FROM foods WHERE name LIKE '%ac%P%';
```

- If you remove the last % sign, you're only looking for foods that also end in p or P.
- LIKE is not case-sensitive in all SQL flavors. In SQLite, you can switch case sensitivity on (PRAGMA case_sensitive_like=ON):

```
PRAGMA case_sensitive_like=OFF;  /* Values are ON or OFF */
SELECT id, name FROM foods WHERE name LIKE '%ac%P%';
```

- A useful trick is NOT to negate a pattern: for example, if you did not want foods with Sch in the name (excludes German/Dutch foods):

```
SELECT id, name FROM foods
WHERE name LIKE '%ac%P%' AND name NOT LIKE '%Sch%';
```

- In Unix/Linux, *globbing* refers to auto-completion for example when searching for files beginning with hello using a wildcard: ls -l hello*.

```
ls -l foo*
```

- The GLOB operator uses wildcards like * and _ and matching is cas sensitive:

```
/* match all rows whose ‘name‘ begins with ‘Pine‘ */
SELECT id, name FROM foods WHERE name GLOB 'Pine*';
```

- You can get very creative if you know, like and use regular expressions. SQLite does not provide native implementations but you can develop your own using the sqlite_create_function() API call (read more).

Limiting and ordering

- You can limit size and range of the result with `LIMIT` and `OFFSET`:

```
SELECT * FROM food_types ORDER BY id LIMIT 1 OFFSET 1;
```

- The `OFFSET` clause skips one row (`Bakery`), and the `LIMIT` clause returns a maximum of one row (`Cereal`). They come last in the pipeline.
- The `ORDER BY` clause sorts the result by a column or columns before it is returned.
- This is essential because the rows returned from `SELECT` are never guaranteed to be in a specific order (this is part of the SQL standard).
- In other words: you need `ORDER BY` if you need to count on the result being in any specific order.
- THe `ORDER BY` clause is similar to `SELECT`: it takes an ordered, comma-separated list of column. After each column name, you can specify if you want ascending (`ASC` default) or descending (`DESC`) order.
- Example: extract all rows whose `food` names start with a `B`, and then order them in descending food `type_id` order. Return only 10 results.

```
SELECT * FROM foods WHERE NAME LIKE 'B%'
       ORDER BY type_id DESC, name LIMIT 10;
```

- You can see that the `type_id` column is order in descending order. Within each group of identical food types, the food `name` is ordered in ascending order (by default).
- Challenge: show the `foods` that start with C and order them in ascending order by type, but in descending order by name.

```
SELECT * FROM foods WHERE NAME LIKE 'C%'
       ORDER BY type_id, name DESC LIMIT 10;
```

- If you use both `LIMIT` and `OFFSET` together, you can use a comma notation in place of the `OFFSET` keyword:

```
SELECT * FROM foods WHERE NAME LIKE 'B%'
       ORDER BY type_id DESC, name LIMIT 2,1; /* display 2nd row of result only */
```

Functions

- SQL comes with various built-in functions and aggregetas that can be used in clauses.
- Function types include: mathematical like `ABS()`, and string-formatting like `UPPER()` and `LOWER()`, which convert text to upper- and lowercase, respectively.
- Example: Guess what the output of this command is before running it

```
SELECT UPPER('hello there'), LENGTH('hello there'), ABS(-12);
```

- Built-in functions are case-insensitive: `abs` is the same as `ABS` or `Abs`.
- Functions can accept column values as their arguments:

```
SELECT id, UPPER(name), LENGTH(name) FROM foods
       WHERE type_id=1 LIMIT 10;  /* foods with type_id = 1 (Bakery) */
```

- Since functions can be part of any expression, they can also be part of a `WHERE` clause:

```
SELECT id, UPPER(name), LENGTH(name) FROM foods
       WHERE LENGTH(name) < 5 LIMIT 5;  /* foods with names of fewer than 5 charac
```

Aggregates

- Aggregates are functions that calculate a composite (or aggregate) value over a group of rows (or relation) - statistical functions are useful aggregates.
- Within the pipeline, aggregates follow the `WHERE` clause: they compute their values on the rows filtered by `WHERE`. `SELECT` filters first, and then aggregates values.
- The SQLite C API allows you to create fast custom functions and aggregates using C.
- Reasons to perform calculations in SQL rather than in R or Python:

1. **Efficiency:** SQL databases are optimized for operations on large datasets. By using the `AVG()` function directly in SQL, the calculation is done on the database server, which can handle the computation more efficiently than fetching all the data into an external program and then computing the mean. This is especially important with very large datasets.
2. **Network Overhead:** Calculating the mean directly in the database reduces the amount of data that needs to be transferred over the network. If you were to calculate the mean in R or Python, you would first need to transfer all the relevant data from the database to the application, which can be slow and resource-intensive for large datasets.
3. **Simplicity:** Writing a query to calculate the average in SQL is straightforward and keeps the data manipulation logic within the database, which can make the code easier to manage and understand. It avoids the complexity of fetching the data and then using a separate tool for the calculation.
4. **Scalability:** Databases are designed to handle queries on large datasets efficiently, and they often include optimizations for aggregation functions like `AVG()`. These optimizations can include parallel processing and indexing strategies that are not as easily implemented in client-side languages.
5. **Real-time data processing**: When working with real-time data, it might be necessary to get the average value updated dynamically as the data changes. Performing this operation directly in SQL ensures that the most current data is used for the calculation without the need for repeated data transfers.
6. **Consistency**: When multiple analyses are being performed on the same data, performing calculations directly in the database ensures that all calculations are based on the same data state, which helps in maintaining consistency across different reports or analyses.

- Example: How many `foods` are `Bakery` (`type_id=1`) goods? (`type_id=1`):

```
SELECT COUNT(*) AS "Baked goods" FROM foods WHERE type_id=1;
```

- When you see an aggregate, you should think "For each row in the table, do something".

- Aggregates can aggregate any expression including functions.
- Example: what is the average `LENGTH` of all names in `foods`?

  ```
  SELECT AVG(LENGTH(name)) AS "Average length of food names" FROM foods;
  ```

- You might not care for the decimal places:

  ```
  SELECT ROUND(AVG(LENGTH(name))) AS "Average length of food names" FROM foods;
  ```

Grouping

- You can compute aggregates over an entire query result (subset of rows). You can also split that result into groups of rows with like values and compute aggregates on each group - all in one step.
- Example: group the types of food by type.

  ```
  SELECT type_id FROM foods GROUP BY type_id;
  ```

- `GROUP BY` takes the output from `WHERE` and splits it into groups of rows that share a common value (or values) for a specific column (or columns) - see image at GitHub:

  In the example, `GROUP BY` organizes all rows in `foods` in 15 (`DISTINCT type_ID`) groups varying by `type_id`.

- When `GROUP BY` is used, `SELECT` applies aggregates to each group separately rather than the entire filtered result as a whole.
- Example: count the number of records per group, for each type:

  ```
  SELECT type_id AS "Food group", COUNT(*) AS "Foods by group"
  FROM foods GROUP BY type_id;
  ```

- Here, `COUNT` is applied 15 times, once for each group:

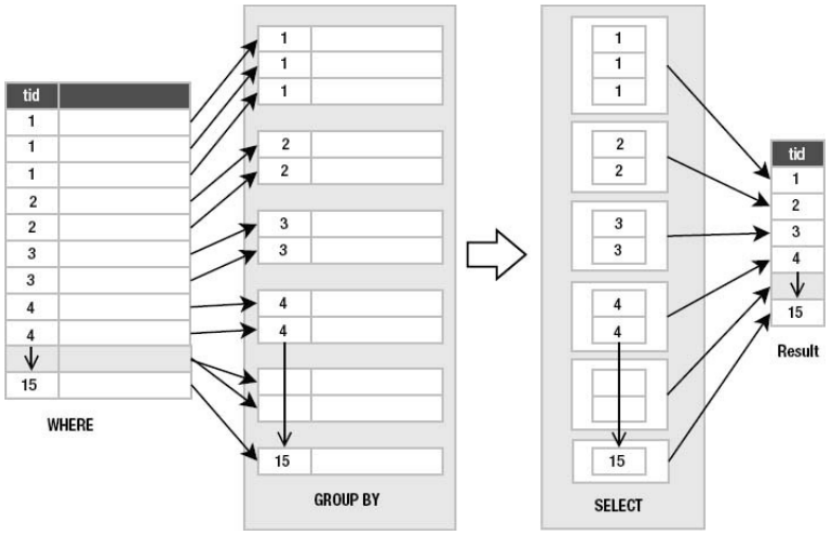
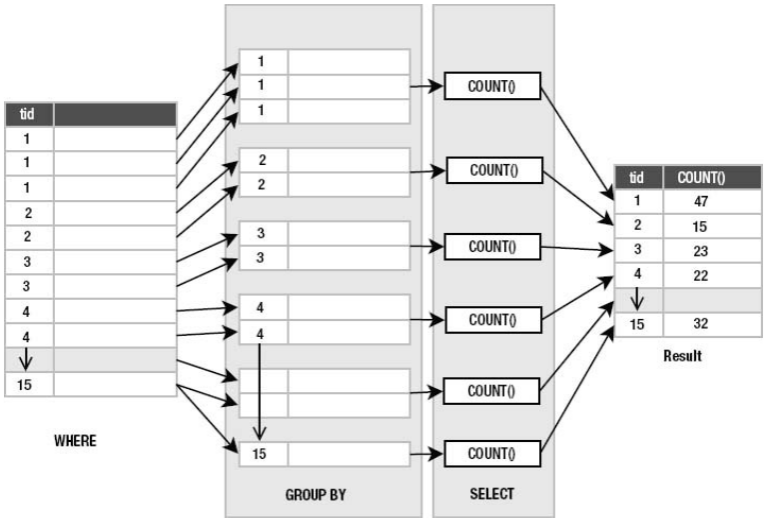


Figure 1: Source: Allen/Owens (Apress, 2010)



- The number of `Bakery` foods is 47, the number of `Cereal` foods is 15, etc. To get the same information, you could also run individual queries:

```
SELECT COUNT(*) AS "Bakery" FROM foods WHERE type_id=1;
```

```
SELECT COUNT(*) AS "Cereal" FROM foods WHERE type_id=2;
SELECT COUNT(*) AS "Chicken/Fowl" FROM foods WHERE type_id=3;
.header OFF
SELECT "               (...) ";
.header ON
SELECT COUNT(*) AS "Vegetables" FROM foods WHERE type_id=15;
```

TODO `HAVING` patterns

TODO Removing duplicates

TODO SUMMARY

- Table joins are a big topic and we'll postpone their discussion until after we talked about database design some more.