SELECT ROUND-UP

Table of Contents

README

This notebook contains a round-up of the entire SELECT command pipeline in SQL. The code blocks run with SQLite, and we use a simple database of three different tables to explore SELECT.

To run this notebook, which resides in GitHub and GDrive (in the *practice directory), you need to have SQLite installed and in the PATH so that Emacs can find it, and you need to have the* ~.emacs~ file also available from GDrive, installed in Emacs' HOME directory.

Download

- Download `xyz.zip` from GDrive.
- This includes the SQLite file `xyz.sql` and a directory with image files.

Preliminaries

- For the presentation, I'm going to use the XMind-map from GitHub.

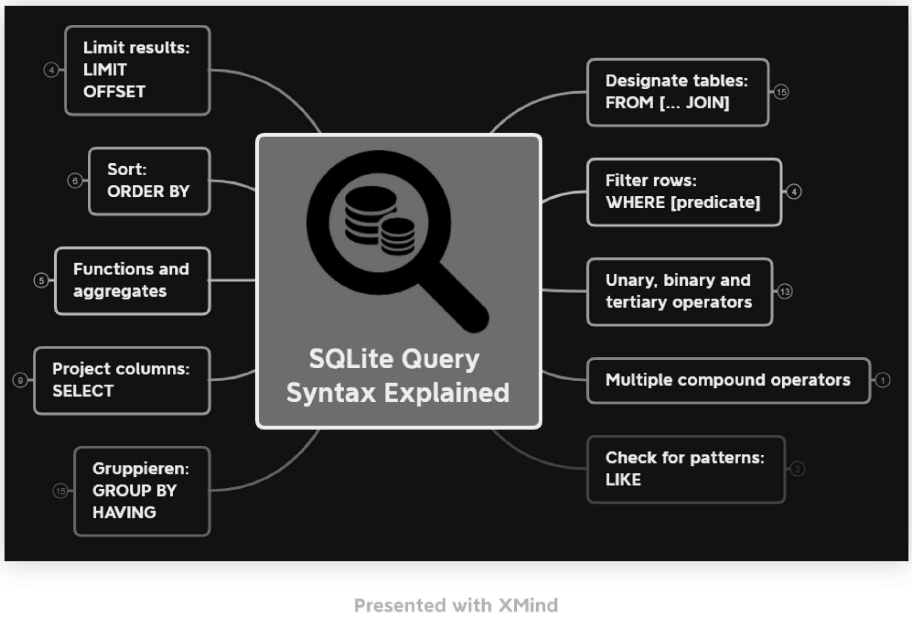

Figure 1: SELECT round-up

- `[X]`

  How are the commands in the mindmap ordered (from right to left)?

  > Answer: this is the pipeline order for `SELECT`

Create some tables

- `[X]`

  Let's load some tables to play with. The file `xyz.sql` is available from GDrive's notebook directory. It contains a complete database dump. We load it with the SQLite command `.read`, then check database and tables. There should be three, named `x`, `y`, and `z`.

```
.read xyz.sql
.database
.tables
```

```
main: c:\Users\birkenkrahe\Documents\GitHub\db330\practice\xyz.db r/w
x  y  z
```

- `[ ]`

  Challenge: if the tables already exist in `xyz.db` (and you don't have the `IF NOT EXISTS` clause in the `CREATE TABLE` statement), you need to delete the database before loading the tables. But even if you do that, the `INSERT` commands will add to the existing tables. So how can you limit the import or delete duplicate entries?

  > ANSWER: You have to select the unique entries with `DISTINCT`.

- We use `SELECT` with the wildcard `*` for the column selection, and without conditions (i.e. conditions on row selections) to look at each table.
- [ ]

  Print table `x` - one integer column and one column of strings.

  ```
  SELECT * FROM x;
  ```

a	b
1	Alice
2	Bob
3	Charlie

- [ ]

  Print table `y` - one integer column and one column of floats.

  ```
  SELECT * FROM y;
  ```

c	d
1	3.14159
1	2.71828
2	1.61803

- [ ]

  Print table `y` - two integer columns

  ```
  SELECT * FROM z;
  ```

a	e
1	100
1	150
3	300
9	900

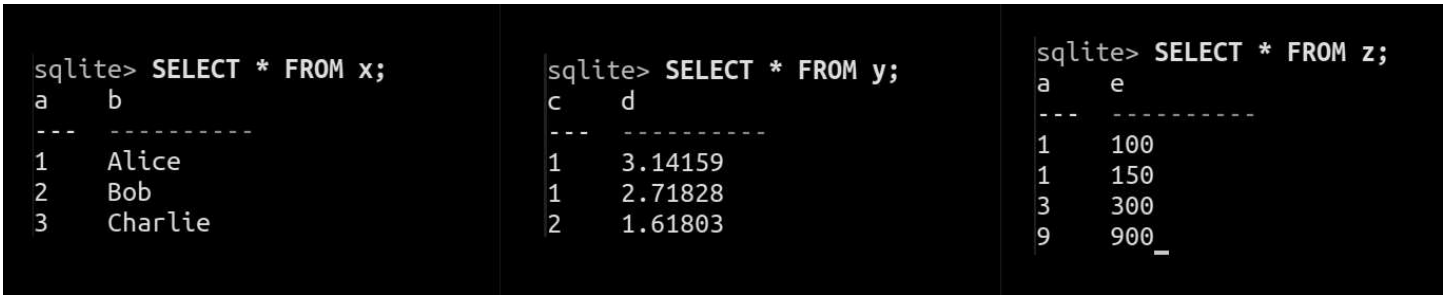

Figure 2: Sample tables x,y,z

Simple JOINs

- [ ]

  CROSS JOIN: Since both tables had 3 rows and 2 colums, the result set has 9=3*3 rows and 4=2*2 columns.

```
SELECT * FROM x JOIN y;
```

a	b	c	d
1	Alice	1	3.14159
1	Alice	1	2.71828
1	Alice	2	1.61803
2	Bob	1	3.14159
2	Bob	1	2.71828
2	Bob	2	1.61803
3	Charlie	1	3.14159
3	Charlie	1	2.71828
3	Charlie	2	1.61803

```
SELECT * FROM x CROSS JOIN y;
```

a	b	c	d
1	Alice	1	3.14159
1	Alice	1	2.71828
1	Alice	2	1.61803
2	Bob	1	3.14159
2	Bob	1	2.71828

2	Bob	2	1.61803
3	Charlie	1	3.14159
3	Charlie	1	2.71828
3	Charlie	2	1.61803

```
SELECT * FROM x,y;
```

a	b	c	d
1	Alice	1	3.14159
1	Alice	1	2.71828
1	Alice	2	1.61803
2	Bob	1	3.14159
2	Bob	1	2.71828
2	Bob	2	1.61803
3	Charlie	1	3.14159
3	Charlie	1	2.71828
3	Charlie	2	1.61803

JOIN … ON

- [ ]

  INNER JOIN: Remember the zipper principle - identify 2 columns to zip xtogether. This identification follows after the `ON` keyword.

  ```
  SELECT * FROM x JOIN y ON a = c;
  ```

a	b	c	d
1	Alice	1	2.71828
1	Alice	1	3.14159
2	Bob	2	1.61803

  - Compare with the CROSS JOIN before. This time, Only those columns that satisfy the condition `a=c` are included in 1
- [ ]

  What if we want to `JOIN` tables `x` and `z`? They both have a column named `a`. We now need to qualify the selection with `.`

  ```
  SELECT * FROM x JOIN z ON x.a = z.a;
  ```

a	b	a	e

-	-------	-	---
1	Alice	1	100
1	Alice	1	150
3	Charlie	3	300

- [ ]

You can resolve column name confusions (same column name in different tables) by using aliases. Run the previous command 1 again, but alias `x` as `t_x` and `z` as `t_z`.

```
SELECT * FROM x AS t_x JOIN z AS t_z ON t_x.a = t_z.a;
```

a	b	a	e
-	-------	-	---
1	Alice	1	100
1	Alice	1	150
3	Charlie	3	300

- [ ]

There are five (!) more `JOIN` commands:

  - `LEFT OUTER JOIN` will also include not matched items
  - `COMPOUND JOIN` joins multiple tables

```
SELECT * FROM x JOIN y ON x.a=y.c LEFT OUTER JOIN z ON y.c=z.a;
```

a	b	c	d	a	e
-	-----	-	-------	------	------
1	Alice	1	2.71828	1	100
1	Alice	1	2.71828	1	150
1	Alice	1	3.14159	1	100
1	Alice	1	3.14159	1	150
2	Bob	2	1.61803	[NULL]	[NULL]

  - Work through this example until you reall understand what this multiple join = INNER JOIN + LEFT JOIN does!

WHERE examples

- [ ]

Print a row: the value `Alice` for the attribute `x.b`.

```
SELECT * FROM x WHERE x.b = 'Alice';
```

a	b
-	-----

1	Alice

- [ ]

Print a range of values, for 1.0 < d < 3.0.

```
SELECT * FROM y WHERE y.d BETWEEN 1.0 AND 3.0;
```

c	d
1	2.71828
2	1.61803

- [ ]

Print columns c, d and a column for the sum of c+d with the condition that the sum is smaller than 4.

```
SELECT c, d, c+d AS sum FROM y WHERE sum < 4.0
```

c	d	sum
1	2.71828	3.71828
2	1.61803	3.61803

- [ ]

The next block 1 uses foods.db to select a range of values with wildcards. Here, * instead of % would also work (try it).

```
Select name from foods where name between 'Ta%' AND 'Ti%';
```

Tarragon
Tea
Three Musketeers
Tamale
Tamales

GROUP BY Examples

- [ ]

Group table z by the column z.a. Can you guess how many rows are going to be printed?

```
SELECT a FROM z GROUP BY z.a;
```

a
1

3
9

- [ ]

Print the number of rows next to every value of z.a. Call this new column 'count'n.

```
SELECT a, COUNT(a) AS count FROM z GROUP BY z.a;
```

a	count
-	-----
1	2
3	1
9	1

- [ ]

Run 1 again (group by z.a) but now also print out the sum of all the z.e values in each group. Call the new column 'TOTAL'.

```
SELECT a, sum(e) AS TOTAL FROM z GROUP BY z.a;
```

a	TOTAL
-	-----
1	250
3	300
9	900

- [ ]

Run 1 again (group by z.a) but now also compute

- the sum(e) as SUM
- the count(e) as TOTAL
- the average as AGG computed with sum and count
- the average as AVG computed with the aggregate function

```
SELECT a, sum(e) as SUM, count(e) as TOTAL, sum(e)/count(e) AS AGG, avg(e) AS AVG FROM z GROUP BY z.a;
```

a	SUM	TOTAL	AGG	AVG
-	---	-----	---	-----
1	250	2	125	125.0
3	300	1	300	300.0
9	900	1	900	900.0

- [ ]

A HAVING clause can be used to filter rows based off the results of the sum() aggregation. Run the block 1.

```
SELECT a, sum(e) AS TOTAL FROM z GROUP BY z.a HAVING total > 500;
```

a	TOTAL
-	-----
9	900

- [ ]

An example with the foods database, and the table `foods`; print the food type ID and the total number of food types per food type group, and print those IDs whose group has less than 20 foods in it.

```
SELECT type_id, COUNT(*)
FROM foods
GROUP BY type_id
HAVING COUNT(*) < 20;
```

type_id	COUNT(*)
-------	--------
2	15
5	17
6	4
11	16
13	14
14	19

type_id	COUNT(*)
-------	--------
2	15
5	17
6	4
11	16
13	14
14	19

ORDER BY examples

- [ ]

Order table y by the numbers in y.d, and print all columns.

```
SELECT * FROM y ORDER BY d;
```

c	d

```
-    -------
2   1.61803
1   2.71828
1   3.14159
```

```
c         d
-    -------
2   1.61803
1   2.71828
1   3.14159
```

- [ ]

An example from the foods database. Display all columns from the table `foods`, and filter those rows whose `name` begins with a `B`. Print only 10 lines.

```
SELECT * FROM foods WHERE name LIKE 'B%' LIMIT 10;
```

```
id  type_id  name
--  -------  ----------------------
1   1        Bagels
2   1        Bagels, raisin
3   1        Bavarian Cream Pie
4   1        Bear Claws
5   1        Black and White cookies
6   1        Bread (with nuts)
7   1        Butterfingers
48  2        Bran
63  3        Broiled Chicken
87  4        Barbeque Sauce
```

- [ ]

Now take the command from 1 and order by food type ID in descending order. To do this, put `DESC` after the column name.

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

```
id   type_id  name
---  -------  -------------------------
382  15       Baked Beans
383  15       Baked Potato w/Sour Cream
384  15       Big Salad
385  15       Brocolli
362  14       Bouillabaisse
326  12       Bologna
327  12       Bacon Club (no turkey)
328  12       BLT
329  12       Brisket Sandwich
274  10       Bacon
```

- [ ]

You can order by any number of columns. Modify the command in 1: order in descending order on `type_id`, and then order the result in ascending order alphabetically on `name`.

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

```
id   type_id  name
---  -------  ------------------------
382  15       Baked Beans
383  15       Baked Potato w/Sour Cream
384  15       Big Salad
385  15       Brocolli
362  14       Bouillabaisse
328  12       BLT
327  12       Bacon Club (no turkey)
326  12       Bologna
329  12       Brisket Sandwich
274  10       Bacon
```

LIMIT and OFFSET examples

- [ ]

  From foods, print the first 5 lines of all columns.

```
SELECT * FROM foods LIMIT 5;
```

```
id  type_id  name
--  -------  ----------------------
1   1        Bagels
2   1        Bagels, raisin
3   1        Bavarian Cream Pie
4   1        Bear Claws
5   1        Black and White cookies
```

- [ ]

  Change the code in 1 only to skip the first 2 lines.

```
SELECT * FROM foods LIMIT 5 OFFSET 2;
```

```
id  type_id  name
--  -------  ----------------------
3   1        Bavarian Cream Pie
4   1        Bear Claws
5   1        Black and White cookies
6   1        Bread (with nuts)
7   1        Butterfingers
```

- [ ]

  Skip the first 2 lines without using the keyword `OFFSET`.

```
SELECT * FROM foods LIMIT 2, 5;
```

```
id  type_id  name
--  -------  ----------------------
3   1        Bavarian Cream Pie
4   1        Bear Claws
5   1        Black and White cookies
6   1        Bread (with nuts)
7   1        Butterfingers
```

Author: Marcus Birkenkrahe
Created: 2022-03-11 Fri 22:32
Validate