

SQL & Indexes

Lecture 17

Dr. Colin Rundel

SQL

Structures Query Language is a special purpose language for interacting with (querying and modifying) indexed tabular data.

- ANSI Standard but with dialect divergence (MySQL, Postgres, SQLite, etc.)
- This functionality maps very closely (but not exactly) with the data manipulation verbs present in dplyr.
- SQL is likely to be a foundational skill if you go into industry - learn it and put it on your CV

Connecting via CLI

```
1 cr173@trig2 [class_2023_10_27]$ sqlite3 employees.sqlite
2
3 SQLite version 3.36.0 2021-06-18 18:36:39
4 Enter ".help" for usage hints.
5 Connected to a transient in-memory database.
6 Use ".open FILENAME" to reopen on a persistent database.
7 sqlite>
```

Table information

The following is specific to SQLite

```
1 sqlite> .tables
2
3 ## employees
```

```
1 sqlite> .schema employees
2
3 ## CREATE TABLE `employees` (
4 ##   `name` TEXT,
5 ##   `email` TEXT,
6 ##   `salary` REAL,
7 ##   `dept` TEXT
8 ## );
```

```
1 sqlite> .indices employees
```

SELECT Statements

```
1  sqlite> SELECT * FROM employees;  
2  
3  ## Alice|alice@company.com|52000.0|Accounting  
4  ## Bob|bob@company.com|40000.0|Accounting  
5  ## Carol|carol@company.com|30000.0|Sales  
6  ## Dave|dave@company.com|33000.0|Accounting  
7  ## Eve|eve@company.com|44000.0|Sales  
8  ## Frank|frank@comany.com|37000.0|Sales
```

Pretty Output

We can make this table output a little nicer with some additional SQLite options:

```
1  sqlite> .mode column
2  sqlite> .headers on
3  sqlite> SELECT * FROM employees;
4
5  ## name          email                salary             dept
6  ## -----      -
7  ## Alice        alice@company.com   52000.0           Accounting
8  ## Bob          bob@company.com     40000.0           Accounting
9  ## Carol        carol@company.com   30000.0           Sales
10 ## Dave         dave@company.com    33000.0           Accounting
11 ## Eve          eve@company.com     44000.0           Sales
12 ## Frank       frank@comany.com    37000.0           Sales
```

select() using SELECT

We can subset for certain columns (and rename them) using `SELECT`

```
1  sqlite> SELECT name AS first_name, salary FROM employees;
2
3  ## first_name  salary
4  ## -----  -----
5  ## Alice      52000.0
6  ## Bob        40000.0
7  ## Carol      30000.0
8  ## Dave       33000.0
9  ## Eve        44000.0
10 ## Frank      37000.0
```

arrange() using ORDER BY

We can sort our results by adding **ORDER BY** to our **SELECT** statement

```
1 sqlite> SELECT name AS first_name, salary FROM employees ORDER BY salary;
2
3 ## first_name  salary
4 ## -----  -----
5 ## Carol      30000.0
6 ## Dave       33000.0
7 ## Frank      37000.0
8 ## Bob        40000.0
9 ## Eve        44000.0
10 ## Alice     52000.0
```

We can sort in the opposite order by adding **DESC**

```
1 SELECT name AS first_name, salary FROM employees ORDER BY salary DESC;
2
3 ## first_name  salary
4 ## -----  -----
5 ## Alice      52000.0
6 ## Eve        44000.0
7 ## Bob        40000.0
8 ## Frank      37000.0
9 ## Dave       33000.0
10 ## Carol     30000.0
```


filter() using WHERE

We can filter rows by adding **WHERE** to our statements

```
1 sqlite> SELECT * FROM employees WHERE salary < 40000;
2
3 ## name          email                salary      dept
4 ## -----      -
```

##	name	email	salary	dept
5	## Carol	carol@company.com	30000.0	Sales
6	## Dave	dave@company.com	33000.0	Accounting
7	## Frank	frank@comany.com	37000.0	Sales

```
1 sqlite> SELECT * FROM employees WHERE salary < 40000 AND dept = "Sales";
2
3 ## name          email                salary      dept
4 ## -----      -
```

##	name	email	salary	dept
5	## Carol	carol@company.com	30000.0	Sales
6	## Frank	frank@comany.com	37000.0	Sales

group_by() using GROUP BY

We can create groups for the purpose of summarizing using `GROUP BY`. As with `dplyr` it is not terribly useful by itself.

```
1 sqlite> SELECT * FROM employees GROUP BY dept;
2
3 ## name          email          salary      dept
4 ## -----          -----          -----          -----
5 ## Dave          dave@company.com 33000.0      Accounting
6 ## Frank         frank@comany.com 37000.0      Sales
```

```
1 sqlite> SELECT dept, COUNT(*) AS n FROM employees GROUP BY dept;
2
3 ## dept          n
4 ## -----          -----
5 ## Accounting    3
6 ## Sales         3
```

head() using LIMIT

We can limit the number of rows we get by using `LIMIT` and order results with `ORDER BY` with or without `DESC`

```
1 sqlite> SELECT * FROM employees LIMIT 3;
2
3 ## name          email          salary         dept
4 ## -----          -----          -----          -----
5 ## Alice        alice@company.com  52000.0        Accounting
6 ## Bob          bob@company.com   40000.0        Accounting
7 ## Carol        carol@company.com  30000.0        Sales
```

```
1 sqlite> SELECT * FROM employees ORDER BY name DESC LIMIT 3;
2
3 ## name          email          salary         dept
4 ## -----          -----          -----          -----
5 ## Frank         frank@comany.com  37000.0        Sales
6 ## Eve           eve@company.com   44000.0        Sales
7 ## Dave          dave@company.com  33000.0        Accounting
```

Exercise 1

Using sqlite calculate the following quantities,

1. The total costs in payroll for this company
2. The average salary within each department

Import CSV files

```
1  sqlite> .mode csv
2  sqlite> .import phone.csv phone
3  sqlite> .tables
4
5  ## employees  phone
6
7  sqlite> .mode column
8  sqlite> SELECT * FROM phone;
9
10 ## name          phone
11 ## -----
12 ## Bob           919 555-1111
13 ## Carol         919 555-2222
14 ## Eve           919 555-3333
15 ## Frank         919 555-4444
```

Joins - Default

By default SQLite uses a `CROSS JOIN` which is not terribly useful most of the time (similar to R's `expand.grid()`)

```
1 sqlite> SELECT * FROM employees JOIN phone;
2
3 ## name          email          salary      dept         name         phone
4 ## -----      -
```

##	name	email	salary	dept	name	phone
5	Alice	alice@company.com	52000.0	Accounting	Bob	919 555-1111
6	Alice	alice@company.com	52000.0	Accounting	Carol	919 555-2222
7	Alice	alice@company.com	52000.0	Accounting	Eve	919 555-3333
8	Alice	alice@company.com	52000.0	Accounting	Frank	919 555-4444
9	Bob	bob@company.com	40000.0	Accounting	Bob	919 555-1111
10	Bob	bob@company.com	40000.0	Accounting	Carol	919 555-2222
11	Bob	bob@company.com	40000.0	Accounting	Eve	919 555-3333
12	Bob	bob@company.com	40000.0	Accounting	Frank	919 555-4444
13	Carol	carol@company.com	30000.0	Sales	Bob	919 555-1111
14	Carol	carol@company.com	30000.0	Sales	Carol	919 555-2222
15	Carol	carol@company.com	30000.0	Sales	Eve	919 555-3333
16	Carol	carol@company.com	30000.0	Sales	Frank	919 555-4444
17	Dave	dave@company.com	33000.0	Accounting	Bob	919 555-1111
18	Dave	dave@company.com	33000.0	Accounting	Carol	919 555-2222
19	Dave	dave@company.com	33000.0	Accounting	Eve	919 555-3333
20	Dave	dave@company.com	33000.0	Accounting	Frank	919 555-4444
21	Eve	eve@company.com	44000.0	Sales	Bob	919 555-1111
22	Eve	eve@company.com	44000.0	Sales	Carol	919 555-2222
23	Eve	eve@company.com	44000.0	Sales	Eve	919 555-3333

Inner Join

If you want SQLite to find the columns to merge on automatically then we prefix the join with **NATURAL**.

```
1  sqlite> SELECT * FROM employees NATURAL JOIN phone;
2
3  ## name          email          salary      dept        phone
4  ## -----      -
```

##	name	email	salary	dept	phone
##	Bob	bob@company.com	40000.0	Accounting	919 555-1111
##	Carol	carol@company.c	30000.0	Sales	919 555-2222
##	Eve	eve@company.com	44000.0	Sales	919 555-3333
##	Frank	frank@comany.co	37000.0	Sales	919 555-4444

Inner Join - Explicit

```
1 sqlite> SELECT * FROM employees JOIN phone ON employees.name = phone.name;
2
3 ## name          email          salary      dept          name          phone
4 ## -----          -----          -----          -----          -----          -----
5 ## Bob           bob@company.com 40000.0      Accounting    Bob           919 555-1111
6 ## Carol         carol@company.c 30000.0      Sales         Carol          919 555-2222
7 ## Eve           eve@company.com 44000.0      Sales         Eve            919 555-3333
8 ## Frank         frank@comany.co 37000.0      Sales         Frank          919 555-4444
```

to avoid the duplicate `name` column we can use `USING` instead of `ON`

```
1 sqlite> SELECT * FROM employees JOIN phone USING(name);
2
3 ## name  email          salary      dept          phone
4 ## -----          -----          -----          -----          -----
5 ## Bob   bob@company.com 40000.0      Accounting    919 555-1111
6 ## Carol carol@company.com 30000.0      Sales         919 555-2222
7 ## Eve   eve@company.com 44000.0      Sales         919 555-3333
8 ## Frank frank@comany.com 37000.0      Sales         919 555-4444
```

As a rule, the `USING` (or `NATURAL`) clause is used if the column names match between tables, otherwise `ON` is needed.

Left Join - Natural

```
1  sqlite> SELECT * FROM employees NATURAL LEFT JOIN phone;
2
3  ## name          email          salary      dept          phone
4  ## -----      -
5  ## Alice         alice@company.com 52000.0     Accounting
6  ## Bob           bob@company.com  40000.0     Accounting   919 555-11
7  ## Carol         carol@company.com 30000.0     Sales        919 555-22
8  ## Dave          dave@company.com 33000.0     Accounting
9  ## Eve           eve@company.com  44000.0     Sales        919 555-33
10 ## Frank         frank@comany.com 37000.0     Sales        919 555-44
```

Left Join - Explicit

```
1 sqlite> SELECT * FROM employees LEFT JOIN phone ON employees.name = phone.name;
2
3 ## name          email          salary      dept          name          phone
4 ## -----          -----          -----          -----          -----          -----
5 ## Alice         alice@company.com 52000.0     Accounting
6 ## Bob           bob@company.com  40000.0     Accounting    Bob           919 555-11
7 ## Carol         carol@company.com 30000.0     Sales         Carol         919 555-22
8 ## Dave          dave@company.com 33000.0     Accounting
9 ## Eve           eve@company.com  44000.0     Sales         Eve           919 555-33
10 ## Frank        frank@comany.com 37000.0     Sales         Frank         919 555-44
```

As above to avoid the duplicate `name` column we can use `USING`, or can be more selective about our returned columns,

```
1 sqlite> SELECT employees.*, phone FROM employees LEFT JOIN phone ON employees.name = phone.name;
2
3 ## name  email          salary      dept          phone
4 ## -----          -----          -----          -----          -----
5 ## Alice alice@company.com 52000.0     Accounting
6 ## Bob   bob@company.com  40000.0     Accounting    919 555-1111
7 ## Carol carol@company.com 30000.0     Sales         919 555-2222
8 ## Dave  dave@company.com 33000.0     Accounting
9 ## Eve   eve@company.com  44000.0     Sales         919 555-3333
10 ## Frank frank@comany.com 37000.0     Sales         919 555-4444
```

Other Joins

Note that SQLite does not support directly support an **OUTER JOIN** (e.g a full join in dplyr) or a **RIGHT JOIN**.

- A **RIGHT JOIN** can be achieved by swapping the two tables (i.e. A right join B is equivalent to B left join A)
- An **OUTER JOIN** can be achieved via using **UNION ALL** with both left joins (A on B and B on A)

Subqueries

We can nest tables within tables for the purpose of queries.

```
1 SELECT * FROM (SELECT * FROM employees NATURAL LEFT JOIN phone) WHERE phone IS NULL;
2
3 ## name          email          salary      dept        phone
4 ## -----      -
```

##	name	email	salary	dept	phone
5 ##	Alice	alice@company.com	52000.0	Accounting	
6 ##	Dave	dave@company.com	33000.0	Accounting	

```
1 sqlite> SELECT * FROM (SELECT * FROM employees NATURAL LEFT JOIN phone) WHERE phone IS
2
3 ## name          email          salary      dept        phone
4 ## -----      -
```

##	name	email	salary	dept	phone
5 ##	Bob	bob@company.com	40000.0	Accounting	919 555-1111
6 ##	Carol	carol@company.c	30000.0	Sales	919 555-2222
7 ##	Eve	eve@company.com	44000.0	Sales	919 555-3333
8 ##	Frank	frank@comany.co	37000.0	Sales	919 555-4444

Exercise 2

Lets try to create a table that has a new column - `abv_avg` which contains how much more (or less) than the average, for their department, each person is paid.

Hint - This will require joining a subquery.

`employees.sqlite` is available in the exercises repo.

Creating an index

```
1  sqlite> CREATE INDEX index_name ON employees (name);
2  sqlite> .indices
3
4  index_name
5
6  sqlite> CREATE INDEX index_name_email ON employees (name,email);
7  sqlite> .indices
8
9  index_name
10 index_name_email
```

Query performance

Setup

To give us a bit more variety, we have created another SQLite database `flights.sqlite` that contains both `nycflights13::flights` and `nycflights13::planes`, the latter of which has details on the characteristics of the planes in the dataset as identified by their tail numbers.

```
1 db = DBI::dbConnect(RSQLite::SQLite(), "flights.sqlite")
2 dplyr::copy_to(db, nycflights13::flights, name = "flights", temporary = FALSE)
3 dplyr::copy_to(db, nycflights13::planes, name = "planes", temporary = FALSE,
4 DBI::dbDisconnect(db)
```

All of the following code will be run in the SQLite command line interface, make sure you've created the database and copied both the flights and planes tables into the db.

Opening `flights.sqlite`

The database can then be opened from the terminal tab using,

```
1 > sqlite3 flights.sqlite
```

As before we should set a couple of configuration options so that our output is readable, we include `.timer on` so that we get time our queries.

```
1 sqlite> .headers on
2 sqlite> .mode column
3 sqlite> .timer on
```

flights

```
1 sqlite> SELECT * FROM flights LIMIT 10;
2
3 ## year  month  day  dep_time  sched_dep_time  dep_delay  arr_time  sched_arr_time  arr_delay  carrier
4 ## ----  -
5 ## 2013  1      1    517      515             2.0       830         819           11.0       UA
6 ## 2013  1      1    533      529             4.0       850         830           20.0       UA
7 ## 2013  1      1    542      540             2.0       923         850           33.0       AA
8 ## 2013  1      1    544      545            -1.0      1004        1022          -18.0      B6
9 ## 2013  1      1    554      600            -6.0       812         837          -25.0      DL
10 ## 2013  1      1    554      558            -4.0       740         728           12.0       UA
11 ## 2013  1      1    555      600            -5.0       913         854           19.0       B6
12 ## 2013  1      1    557      600            -3.0       709         723          -14.0      EV
13 ## 2013  1      1    557      600            -3.0       838         846           -8.0       B6
14 ## 2013  1      1    558      600            -2.0       753         745            8.0       AA
15 ##
16 ## Run Time: real 0.051 user 0.000258 sys 0.000126
```

planes

```
1 sqlite> SELECT * FROM planes LIMIT 10;
2
3 ## tailnum  year  type                manufacturer      model      engines  seats  speed  engine
4 ## -----  ----  -
5 ## N10156   2004  Fixed wing multi engine  EMBRAER        EMB-145XR   2        55
6 ## N102UW   1998  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182
7 ## N103US   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182
8 ## N104UW   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182
9 ## N10575   2002  Fixed wing multi engine  EMBRAER        EMB-145LR   2        55
10 ## N105UW   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182
11 ## N107US   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182
12 ## N108UW   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182
13 ## N109UW   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182
14 ## N110UW   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182
15 ##
16 ## Run Time: real 0.001 user 0.000159 sys 0.000106
```

Exercise 3

Write a query that determines the total number of seats available on all of the planes that flew out of New York in 2013.

Incorrect

```
1 sqlite> SELECT sum(seats) FROM flights NATURAL LEFT JOIN planes;  
2  
3 ## sum(seats)  
4 ## -----  
5 ## 614366  
6 ##  
7 ## Run Time: real 0.148 user 0.139176 sys 0.007804
```

Why?

Correct

Join and select:

```
1 sqlite> SELECT sum(seats) FROM flights LEFT JOIN planes USING (tailnum);
2
3 ## sum(seats)
4 ## -----
5 ## 38851317
6 ##
7 ## Run Time: real 0.176 user 0.167993 sys 0.007354
```

...

Select then join:

```
1 sqlite> SELECT sum(seats) FROM (SELECT tailnum FROM flights) LEFT JOIN (SELECT tailnum, seats FROM plan
2
3 ## sum(seats)
4 ## -----
5 ## 38851317
6 ##
7 ## Run Time: real 0.174 user 0.166085 sys 0.007122
```

EXPLAIN QUERY PLAN

```
1 sqlite> EXPLAIN QUERY PLAN SELECT sum(seats) FROM flights
2   LEFT JOIN planes USING (tailnum);
3
4 ## QUERY PLAN
5 ## |--SCAN flights
6 ## `--SEARCH planes USING AUTOMATIC COVERING INDEX (tailnum=?)
```

```
1 sqlite> EXPLAIN QUERY PLAN SELECT sum(seats) FROM (SELECT tailnum FROM flights)
2   LFET JOIN (SELECT tailnum, seats FROM planes) USING (tailnum);
3
4 ## QUERY PLAN
5 ## |--MATERIALIZED SUBQUERY 2
6 ## |   `--SCAN planes
7 ## |--SCAN flights
8 ## `--SEARCH SUBQUERY 2 USING AUTOMATIC COVERING INDEX (tailnum=?)
```

Key things to look for:

- **SCAN** - indicates that a full table scan is occurring
- **SEARCH** - indicates that only a subset of the table rows are visited
- **AUTOMATIC COVERING INDEX** - indicates that a temporary index has been created for this query

Adding indexes

```
1 sqlite> CREATE INDEX flight_tailnum ON flights (tailnum);  
2  
3 ## Run Time: real 0.241 user 0.210099 sys 0.027611
```

```
1 sqlite> CREATE INDEX plane_tailnum ON planes (tailnum);  
2  
3 ## Run Time: real 0.003 user 0.001407 sys 0.001442
```

```
1 sqlite> .indexes  
2  
3 ## flight_tailnum  plane_tailnum
```


Improvements?

```
1 sqlite> SELECT sum(seats) FROM flights LEFT JOIN planes USING (tailnum);
2
3 ## sum(seats)
4 ## -----
5 ## 38851317
6 ##
7 ## Run Time: real 0.118 user 0.115899 sys 0.001952
```

```
1 sqlite> SELECT sum(seats) FROM (SELECT tailnum FROM flights)
2     LEFT JOIN (SELECT tailnum, seats FROM planes) USING (tailnum);
3
4 ## sum(seats)
5 ## -----
6 ## 38851317
7 ##
8 ## Run Time: real 0.131 user 0.129165 sys 0.001214
```

```
1 sqlite> EXPLAIN QUERY PLAN SELECT sum(seats) FROM flights
2   LEFT JOIN planes USING (tailnum);
3
4 ## QUERY PLAN
5 ## |--SCAN flights USING COVERING INDEX flight_tailnum
6 ## `--SEARCH planes USING INDEX plane_tailnum (tailnum=?)
```

```
1 sqlite> EXPLAIN QUERY PLAN SELECT sum(seats) FROM (SELECT tailnum FROM flights)
2   LEFT JOIN (SELECT tailnum, seats FROM planes) USING (tailnum);
3
4 ## QUERY PLAN
5 ## |--MATERIALIZED SUBQUERY 2
6 ## |  `--SCAN planes
7 ## |--SCAN flights USING COVERING INDEX flight_tailnum
8 ## `--SEARCH SUBQUERY 2 USING AUTOMATIC COVERING INDEX (tailnum=?)
```

Filtering

```
1 sqlite> SELECT origin, count(*) FROM flights WHERE origin = "EWR";
2
3 ## origin  count(*)
4 ## -----  -
5 ## EWR      120835
6 ##
7 ## Run Time: real 0.034 user 0.028124 sys 0.005847
```

```
1 sqlite> EXPLAIN QUERY PLAN  SELECT origin, count(*) FROM flights WHERE origin = "EWR";
2
3 ## QUERY PLAN
4 ## `--SCAN flights
```

```
1 sqlite> SELECT origin, count(*) FROM flights WHERE origin != "EWR";
2
3 ## origin  count(*)
4 ## -----  -
5 ## LGA      215941
6 ##
7 ## Run Time: real 0.036 user 0.029798 sys 0.006171
```

```
1 sqlite> EXPLAIN QUERY PLAN SELECT origin, count(*) FROM flights WHERE origin != "EWR";
2
3 ## QUERY PLAN
4 ## `--SCAN flights
```

Nested indexes

An index can be created on more than one column at a time. This is useful for queries that filter on multiple columns, but note that the order of the columns in the index matters.

```
1 sqlite> CREATE INDEX flights_orig_dest ON flights (origin, dest);  
2  
3 ## Run Time: real 0.267 user 0.232886 sys 0.030270
```

Filtering w/ indexes

```
1 sqlite> SELECT origin, count(*) FROM flights WHERE origin = "EWR";
2
3 ## origin  count(*)
4 ## -----  -----
5 ## EWR      120835
6 ##
7 ## Run Time: real 0.007 user 0.006419 sys 0.000159
```

```
1 sqlite> EXPLAIN QUERY PLAN  SELECT origin, count(*) FROM flights WHERE origin = "EWR";
2
3 ## QUERY PLAN
4 ## `--SEARCH flights USING COVERING INDEX flights_orig_dest (origin=?)
```

```
1 sqlite> SELECT origin, count(*) FROM flights WHERE origin != "EWR";
2
3 ## origin  count(*)
4 ## -----  -----
5 ## JFK      215941
6 ##
7 ## Run Time: real 0.028 user 0.019203 sys 0.000497
```

```
1 sqlite> EXPLAIN QUERY PLAN  SELECT origin, count(*) FROM flights WHERE origin != "EWR";
2
3 ## QUERY PLAN
4 ## `--SCAN flights USING COVERING INDEX flights_orig_dest
```

!= alternative

```
1 sqlite> SELECT origin, count(*) FROM flights
2   WHERE origin > "EWR" OR origin < "EWR";
3
4 ## origin  count(*)
5 ## -----  -----
6 ## JFK      215941
7 ##
8 ## Run Time: real 0.020 user 0.021148 sys 0.001290
```

```
1 sqlite> EXPLAIN QUERY PLAN SELECT origin, count(*) FROM flights
2   WHERE origin > "EWR" OR origin < "EWR";
3
4 ## QUERY PLAN
5 ##  |--MULTI-INDEX OR
6 ##   |--INDEX 1
7 ##   |  |--SEARCH flights USING COVERING INDEX flights_orig_dest (origin>?)
8 ##   |--INDEX 2
9 ##   |  |--SEARCH flights USING COVERING INDEX flights_orig_dest (origin<?)
```

What about dest?

```
1 sqlite> SELECT dest, count(*) FROM flights WHERE dest = "LAX";
2
3 ## dest  count(*)
4 ## ----  -
5 ## LAX   16174
6 ##
7 ## Run Time: real 0.027 user 0.016513 sys 0.000237
```

```
1 sqlite> EXPLAIN QUERY PLAN SELECT dest, count(*) FROM flights WHERE dest = "LAX";
2
3 ## QUERY PLAN
4 ## `--SCAN flights USING COVERING INDEX flights_orig_dest
```

```
1 sqlite> SELECT dest, count(*) FROM flights WHERE dest = "LAX" AND origin = "EWR";
2
3 ## dest  count(*)
4 ## ----  -
5 ## LAX   4912
6 ##
7 ## Run Time: real 0.001 user 0.000729 sys 0.000208
```

```
1 sqlite> EXPLAIN QUERY PLAN SELECT dest, count(*) FROM flights WHERE dest = "LAX" AND origin = "EWR";
2
3 ## QUERY PLAN
4 ## `--SEARCH flights USING COVERING INDEX flights_orig_dest (origin=? AND dest=?)
```

Group by

```
1  sqlite> SELECT carrier, count(*) FROM flights
2  GROUP BY carrier;
3
4  ## carrier  count(*)
5  ## -----  -----
6  ## 9E      18460
7  ## AA      32729
8  ## AS      714
9  ## B6      54635
10 ## DL      48110
11 ## EV      54173
12 ## F9      685
13 ## FL      3260
14 ## HA      342
15 ## MQ      26397
16 ## OO      32
17 ## UA      58665
18 ## US      20536
19 ## VX      5162
20 ## WN      12275
21 ## YV      601
22 ##
23 ## Run Time: real 0.172 user 0.114274 sys 0.0189
```

```
1  sqlite> EXPLAIN QUERY PLAN SELECT carrier, count
2
3  ## QUERY PLAN
4  ## |--SCAN flights
5  ## `--USE TEMP B-TREE FOR GROUP BY
```


GROUP with index

```
1 sqlite> CREATE INDEX flight_carrier ON flights (carrier);  
2  
3 ##  
4 ## Run Time: real 0.131 user 0.113260 sys 0.014691
```

Why not index all the things?

- As mentioned before, creating an index requires additional storage (memory or disk)
- Additionally, when adding or updating data - indexes also need to be updated, making these processes slower (read vs. write tradeoffs)
- Index order matters - `flights (origin, dest)`, `flights (dest, origin)` are not the same and similarly are not the same as separate indexes on `dest` and `origin`.
 - common access patterns will determine what of the above will perform better

