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

```
cr173@trig2 [class_2022_10_28]$ sqlite3 employees.sqlite

SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

Table information

The following is specific to SQLite

```
sqlite> .tables

employees
```

```
sqlite> .schema employees

CREATE TABLE `employees` (
  `name` TEXT,
  `email` TEXT,
  `salary` REAL,
  `dept` TEXT
);
```

```
sqlite> .indices employees

```

SELECT Statements

```
sqlite> SELECT * FROM employees;

Alice|alice@company.com|52000.0|Accounting
Bob|bob@company.com|40000.0|Accounting
Carol|carol@company.com|30000.0|Sales
Dave|dave@company.com|33000.0|Accounting
Eve|eve@company.com|44000.0|Sales
Frank|frank@comany.com|37000.0|Sales
```

Pretty Output

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

```
sqlite> .mode column
sqlite> .headers on
sqlite> SELECT * FROM employees;

name        email                salary      dept
----------  -------------------  ----------  ----------
Alice       alice@company.com    52000.0     Accounting
Bob         bob@company.com      40000.0     Accounting
Carol       carol@company.com    30000.0     Sales
Dave        dave@company.com     33000.0     Accounting
Eve         eve@company.com      44000.0     Sales
Frank       frank@comany.com     37000.0     Sales
```

select() using SELECT

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

```
sqlite> SELECT name AS first_name, salary FROM employees;

first_name  salary
----------  ----------
Alice       52000.0
Bob         40000.0
Carol       30000.0
Dave        33000.0
Eve         44000.0
Frank       37000.0
```

arrange() using ORDER BY

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

```
sqlite> SELECT name AS first_name, salary FROM employees ORDER BY salary;

first_name  salary
----------  ----------
Carol       30000.0
Dave        33000.0
Frank       37000.0
Bob         40000.0
Eve         44000.0
Alice       52000.0
```

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

```
SELECT name AS first_name, salary FROM employees ORDER BY salary DESC;

first_name  salary
----------  ----------
Alice       52000.0
Eve         44000.0
Bob         40000.0
Frank       37000.0
Dave        33000.0
Carol       30000.0
```

filter() using WHERE

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

```
sqlite> SELECT * FROM employees WHERE salary < 40000;

name        email                salary      dept
----------  -------------------  ----------  ----------
Carol       carol@company.com    30000.0     Sales
Dave        dave@company.com     33000.0     Accounting
Frank       frank@comany.com     37000.0     Sales
```

```
sqlite> SELECT * FROM employees WHERE salary < 40000 AND dept = "Sales";

name        email                salary      dept
----------  -------------------  ----------  ----------
Carol       carol@company.com    30000.0     Sales
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.

```
sqlite> SELECT * FROM employees GROUP BY dept;

name        email              salary      dept
----------  -----------------  ----------  ----------
Dave        dave@company.com   33000.0     Accounting
Frank       frank@comany.com   37000.0     Sales
```

```
sqlite> SELECT dept, COUNT(*) AS n FROM employees GROUP BY dept;

dept        n
----------  ----------
Accounting  3
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`

```
sqlite> SELECT * FROM employees LIMIT 3;

name        email              salary      dept
----------  -----------------  ----------  ----------
Alice       alice@company.com  52000.0     Accounting
Bob         bob@company.com    40000.0     Accounting
Carol       carol@company.com  30000.0     Sales
```

```
sqlite> SELECT * FROM employees ORDER BY name DESC LIMIT 3;

name        email              salary      dept
----------  -----------------  ----------  ----------
Frank       frank@comany.com   37000.0     Sales
Eve         eve@company.com    44000.0     Sales
Dave        dave@company.com   33000.0     Accounting
```

Exercise 1

Using sqlite calculate the following quantities,

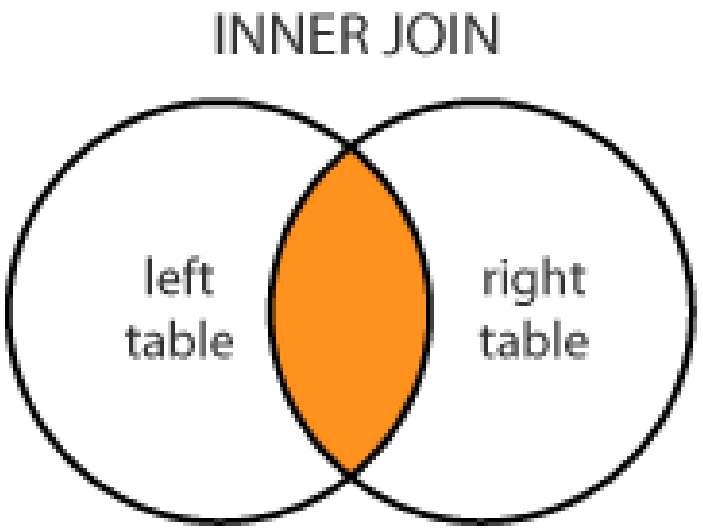
1. The total costs in payroll for this company
2. The average salary within each department

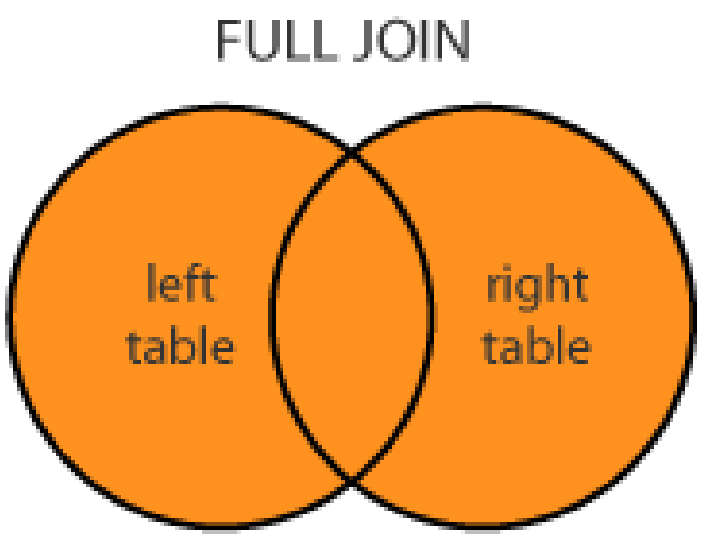
SQLite's aggregation function documentation

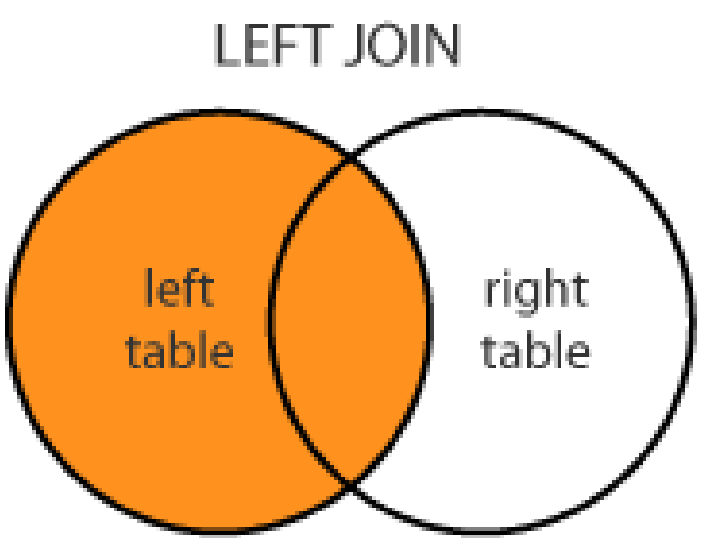
Import CSV files

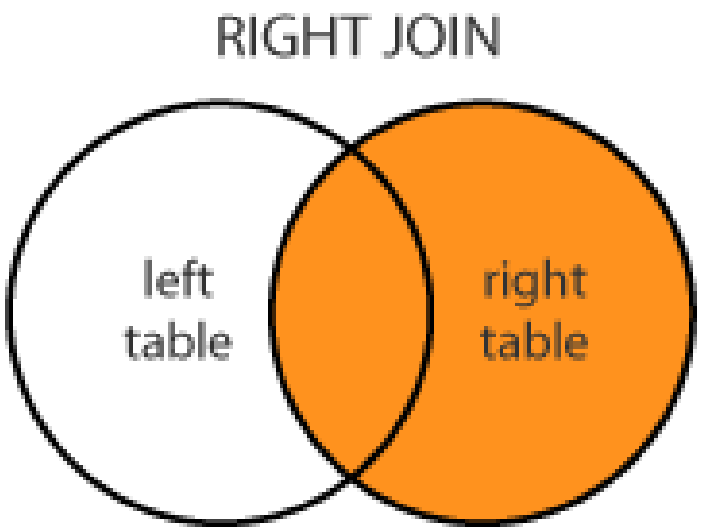
```
sqlite> .mode csv
sqlite> .import phone.csv phone
sqlite> .tables

employees  phone

sqlite> .mode column
sqlite> SELECT * FROM phone;

name        phone
----------  ------------
Bob         919 555-1111
Carol       919 555-2222
Eve         919 555-3333
Frank       919 555-4444
```

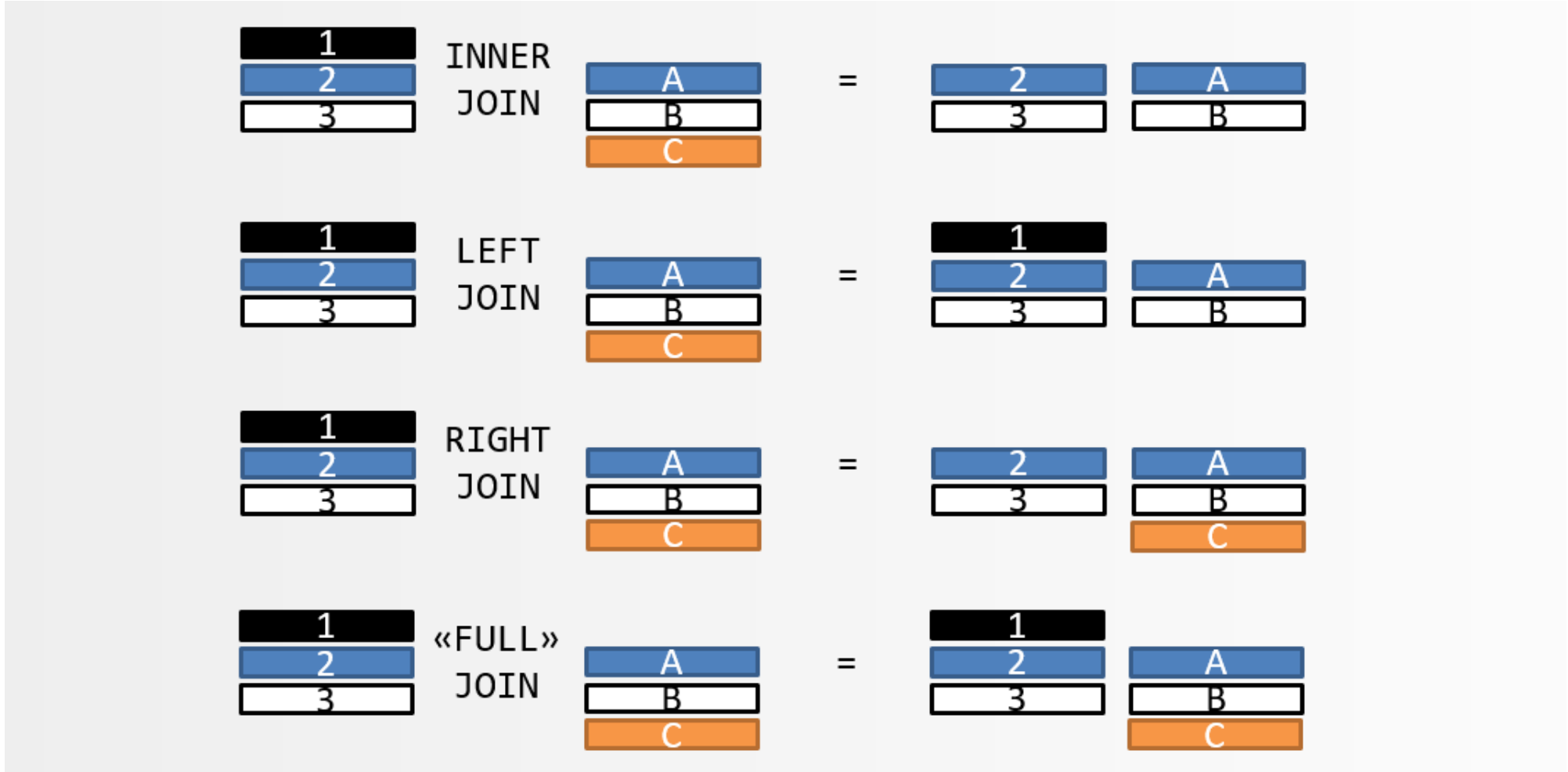
SQL Joins

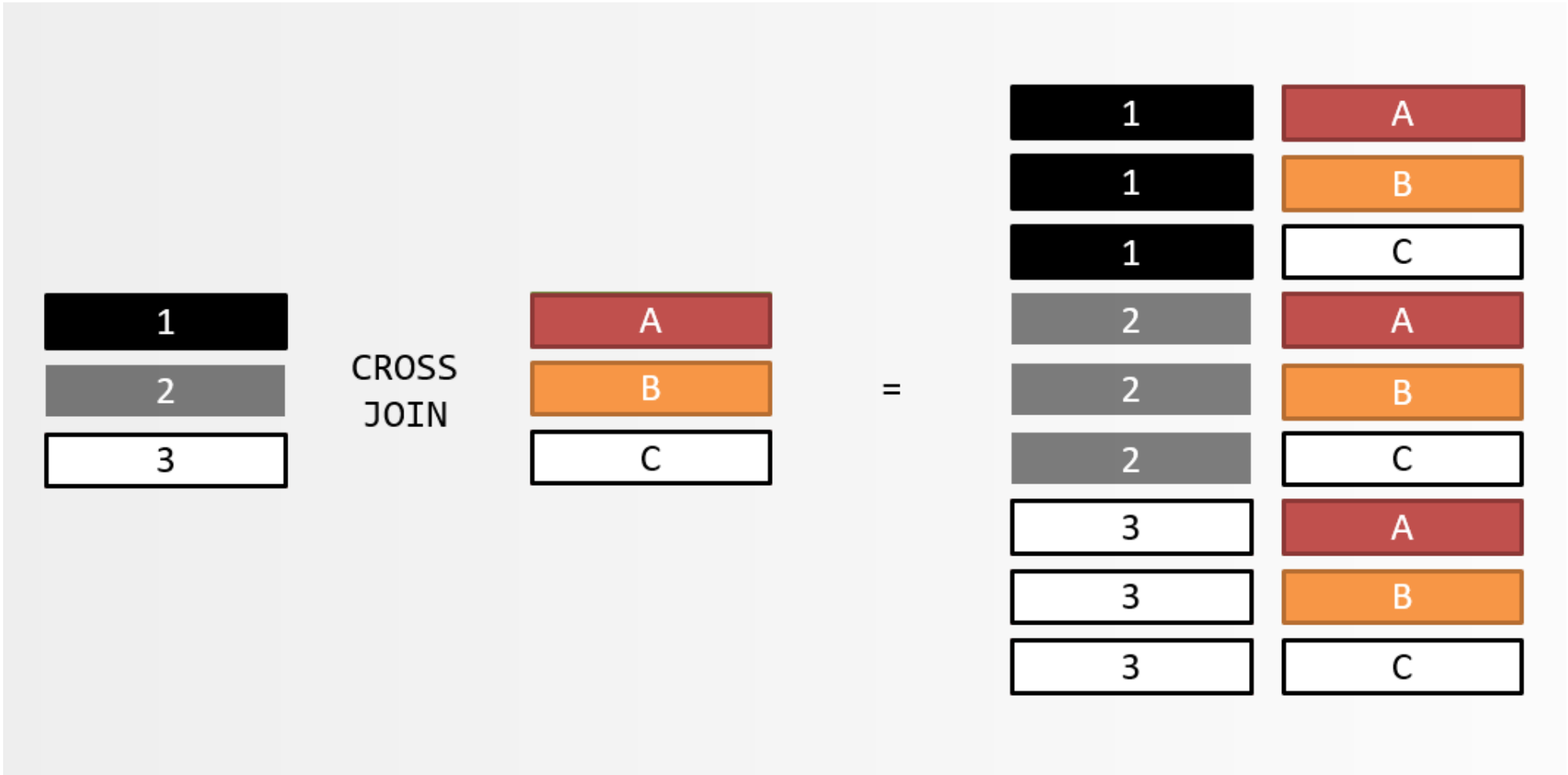

INNER JOIN
left table
right table


FULL JOIN
left table
right table


LEFT JOIN
left table
right table


RIGHT JOIN
left table
right table


1
2
3
INNER
JOIN
A
B
C
=
2
3
A
B
1
2
3
LEFT
JOIN
A
B
C
=
1
2
3
A
B
1
2
3
RIGHT
JOIN
A
B
C
=
2
3
A
B
C
1
2
3
«FULL»
JOIN
A
B
C
=
1
2
3
A
B
C


1
2
3
CROSS JOIN
A
B
C
=
1 A
1 B
1 C
2 A
2 B
2 C
3 A
3 B
3 C

Joins - Default

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

```
sqlite> SELECT * FROM employees JOIN phone;

name        email              salary      dept        name        phone
----------  -----------------  ----------  ----------  ----------  ------------
Alice       alice@company.com  52000.0     Accounting  Bob         919 555-1111
Alice       alice@company.com  52000.0     Accounting  Carol       919 555-2222
Alice       alice@company.com  52000.0     Accounting  Eve         919 555-3333
Alice       alice@company.com  52000.0     Accounting  Frank       919 555-4444
Bob         bob@company.com    40000.0     Accounting  Bob         919 555-1111
Bob         bob@company.com    40000.0     Accounting  Carol       919 555-2222
Bob         bob@company.com    40000.0     Accounting  Eve         919 555-3333
Bob         bob@company.com    40000.0     Accounting  Frank       919 555-4444
Carol       carol@company.com  30000.0     Sales       Bob         919 555-1111
Carol       carol@company.com  30000.0     Sales       Carol       919 555-2222
Carol       carol@company.com  30000.0     Sales       Eve         919 555-3333
Carol       carol@company.com  30000.0     Sales       Frank       919 555-4444
Dave        dave@company.com   33000.0     Accounting  Bob         919 555-1111
Dave        dave@company.com   33000.0     Accounting  Carol       919 555-2222
Dave        dave@company.com   33000.0     Accounting  Eve         919 555-3333
Dave        dave@company.com   33000.0     Accounting  Frank       919 555-4444
Eve         eve@company.com    44000.0     Sales       Bob         919 555-1111
Eve         eve@company.com    44000.0     Sales       Carol       919 555-2222
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`.

```
sqlite> SELECT * FROM employees NATURAL JOIN phone;

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

```
sqlite> SELECT * FROM employees JOIN phone ON employees.name = phone.name;

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

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

```
sqlite> SELECT * FROM employees JOIN phone USING(name);

name   email              salary   dept        phone
-----  -----------------  -------  ----------  ------------
Bob    bob@company.com    40000.0  Accounting  919 555-1111
Carol  carol@company.com  30000.0  Sales       919 555-2222
Eve    eve@company.com    44000.0  Sales       919 555-3333
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

```
sqlite> SELECT * FROM employees NATURAL LEFT JOIN phone;

name        email              salary      dept        phone
----------  -----------------  ----------  ----------  ----------
Alice       alice@company.com  52000.0     Accounting
Bob         bob@company.com    40000.0     Accounting  919 555-11
Carol       carol@company.com  30000.0     Sales       919 555-22
Dave        dave@company.com   33000.0     Accounting
Eve         eve@company.com    44000.0     Sales       919 555-33
Frank       frank@comany.com   37000.0     Sales       919 555-44
```

Left Join - Explicit

```
sqlite> SELECT * FROM employees LEFT JOIN phone ON employees.name = phone.name;

name        email              salary      dept        name        phone
----------  -----------------  ----------  ----------  ----------  ----------
Alice       alice@company.com  52000.0     Accounting
Bob         bob@company.com    40000.0     Accounting  Bob         919 555-11
Carol       carol@company.com  30000.0     Sales       Carol       919 555-22
Dave        dave@company.com   33000.0     Accounting
Eve         eve@company.com    44000.0     Sales       Eve         919 555-33
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,

```
sqlite> SELECT employees.*, phone FROM employees LEFT JOIN phone ON employees.name = phone.name;

name   email              salary   dept        phone
-----  -----------------  -------  ----------  ------------
Alice  alice@company.com  52000.0  Accounting
Bob    bob@company.com    40000.0  Accounting  919 555-1111
Carol  carol@company.com  30000.0  Sales       919 555-2222
Dave   dave@company.com   33000.0  Accounting
Eve    eve@company.com    44000.0  Sales       919 555-3333
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.

```
SELECT * FROM (SELECT * FROM employees NATURAL LEFT JOIN phone) WHERE phone IS NULL;

name        email                salary      dept        phone
----------  -----------------  ----------  ----------  ----------
Alice       alice@company.com  52000.0     Accounting
Dave        dave@company.com   33000.0     Accounting
```

```
sqlite> SELECT * FROM (SELECT * FROM employees NATURAL LEFT JOIN phone) WHERE phone IS

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
```

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

```
sqlite> CREATE INDEX index_name ON employees (name);
sqlite> .indices

index_name

sqlite> CREATE INDEX index_name_email ON employees (name,email);
sqlite> .indices

index_name
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.

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

```
> 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.

```
sqlite> .headers on
sqlite> .mode column
sqlite> .timer on
```

flights

```
sqlite> SELECT * FROM flights LIMIT 10;

## year  month  day  dep_time  sched_dep_time  dep_delay  arr_time  sched_arr_time  arr_delay  carrier
## ----  -----  ---  --------  --------------  ---------  --------  --------------  ---------  -------
## 2013  1      1    517       515             2.0        830       819             11.0       UA
## 2013  1      1    533       529             4.0        850       830             20.0       UA
## 2013  1      1    542       540             2.0        923       850             33.0       AA
## 2013  1      1    544       545             -1.0       1004      1022            -18.0      B6
## 2013  1      1    554       600             -6.0       812       837             -25.0      DL
## 2013  1      1    554       558             -4.0       740       728             12.0       UA
## 2013  1      1    555       600             -5.0       913       854             19.0       B6
## 2013  1      1    557       600             -3.0       709       723             -14.0      EV
## 2013  1      1    557       600             -3.0       838       846             -8.0       B6
## 2013  1      1    558       600             -2.0       753       745             8.0        AA
##
## Run Time: real 0.051 user 0.000258 sys 0.000126
```

planes

```
sqlite> SELECT * FROM planes LIMIT 10;

## tailnum  year  type                      manufacturer      model      engines  seats  speed  engine
## -------  ----  ------------------------  ----------------  ---------  -------  -----  -----  --------
## N10156   2004  Fixed wing multi engine  EMBRAER           EMB-145XR  2        55             Turbo-fa
## N102UW   1998  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182            Turbo-fa
## N103US   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182            Turbo-fa
## N104UW   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182            Turbo-fa
## N10575   2002  Fixed wing multi engine  EMBRAER           EMB-145LR  2        55             Turbo-fa
## N105UW   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182            Turbo-fa
## N107US   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182            Turbo-fa
## N108UW   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182            Turbo-fa
## N109UW   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182            Turbo-fa
## N110UW   1999  Fixed wing multi engine  AIRBUS INDUSTRIE  A320-214   2        182            Turbo-fa
##
## 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

```
sqlite> SELECT sum(seats) FROM flights NATURAL LEFT JOIN planes;

## sum(seats)
## ----------
## 614366
##
## Run Time: real 0.148 user 0.139176 sys 0.007804
```

Why?

Correct

Join and select:

```
sqlite> SELECT sum(seats) FROM flights LEFT JOIN planes USING (tailnum);

## sum(seats)
## ----------
## 38851317
##
## Run Time: real 0.176 user 0.167993 sys 0.007354
```

. . .

Select then join:

```
sqlite> SELECT sum(seats) FROM (SELECT tailnum FROM flights) LEFT JOIN (SELECT tailnum, seats FROM plan

## sum(seats)
## ----------
## 38851317
##
## Run Time: real 0.174 user 0.166085 sys 0.007122
```

EXPLAIN QUERY PLAN

```
sqlite> EXPLAIN QUERY PLAN SELECT sum(seats) FROM flights
  LEFT JOIN planes USING (tailnum);

## QUERY PLAN
## |--SCAN flights
## `--SEARCH planes USING AUTOMATIC COVERING INDEX (tailnum=?)
```

```
sqlite> EXPLAIN QUERY PLAN SELECT sum(seats) FROM (SELECT tailnum FROM flights)
  LFET JOIN (SELECT tailnum, seats FROM planes) USING (tailnum);

## QUERY PLAN
## |--MATERIALIZE SUBQUERY 2
## |  `--SCAN planes
## |--SCAN flights
## `--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

```
sqlite> CREATE INDEX flight_tailnum ON flights (tailnum);

## Run Time: real 0.241 user 0.210099 sys 0.027611
```

```
sqlite> CREATE INDEX plane_tailnum ON planes (tailnum);

## Run Time: real 0.003 user 0.001407 sys 0.001442
```

```
sqlite> .indexes

## flight_tailnum  plane_tailnum
```

Improvements?

```
sqlite> SELECT sum(seats) FROM flights LEFT JOIN planes USING (tailnum);

## sum(seats)
## ----------
## 38851317
##
## Run Time: real 0.118 user 0.115899 sys 0.001952
```

```
sqlite> SELECT sum(seats) FROM (SELECT tailnum FROM flights)
  LEFT JOIN (SELECT tailnum, seats FROM planes) USING (tailnum);

## sum(seats)
## ----------
## 38851317
##
## Run Time: real 0.131 user 0.129165 sys 0.001214
```

```
sqlite> EXPLAIN QUERY PLAN SELECT sum(seats) FROM flights
  LEFT JOIN planes USING (tailnum);

## QUERY PLAN
## |--SCAN flights USING COVERING INDEX flight_tailnum
## `--SEARCH planes USING INDEX plane_tailnum (tailnum=?)
```

```
sqlite> EXPLAIN QUERY PLAN SELECT sum(seats) FROM (SELECT tailnum FROM flights)
  LEFT JOIN (SELECT tailnum, seats FROM planes) USING (tailnum);

## QUERY PLAN
## |--MATERIALIZE SUBQUERY 2
## |  `--SCAN planes
## |--SCAN flights USING COVERING INDEX flight_tailnum
## `--SEARCH SUBQUERY 2 USING AUTOMATIC COVERING INDEX (tailnum=?)
```

Filtering

```
sqlite> SELECT origin, count(*) FROM flights WHERE origin = "EWR";

## origin  count(*)
## ------  --------
## EWR     120835
##
## Run Time: real 0.034 user 0.028124 sys 0.005847
```

```
sqlite> EXPLAIN QUERY PLAN  SELECT origin, count(*) FROM flights WHERE origin = "EWR";

## QUERY PLAN
## `--SCAN flights
```

```
sqlite> SELECT origin, count(*) FROM flights WHERE origin != "EWR";

## origin  count(*)
## ------  --------
## LGA     215941
##
## Run Time: real 0.036 user 0.029798 sys 0.006171
```

```
sqlite> EXPLAIN QUERY PLAN SELECT origin, count(*) FROM flights WHERE origin != "EWR";

## QUERY PLAN
## `--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.

```
sqlite> CREATE INDEX flights_orig_dest ON flights (origin, dest);

## Run Time: real 0.267 user 0.232886 sys 0.030270
```

Filtering w/ indexes

```
sqlite> SELECT origin, count(*) FROM flights WHERE origin = "EWR";

## origin  count(*)
## ------  --------
## EWR     120835
##
## Run Time: real 0.007 user 0.006419 sys 0.000159
```

```
sqlite> EXPLAIN QUERY PLAN  SELECT origin, count(*) FROM flights WHERE origin = "EWR";

## QUERY PLAN
## `--SEARCH flights USING COVERING INDEX flights_orig_dest (origin=?)
```

```
sqlite> SELECT origin, count(*) FROM flights WHERE origin != "EWR";

## origin  count(*)
## ------  --------
## JFK     215941
##
## Run Time: real 0.028 user 0.019203 sys 0.000497
```

```
sqlite> EXPLAIN QUERY PLAN  SELECT origin, count(*) FROM flights WHERE origin != "EWR";

## QUERY PLAN
## `--SCAN flights USING COVERING INDEX flights_orig_dest
```

!= alternative

```
sqlite> SELECT origin, count(*) FROM flights
  WHERE origin > "EWR" OR origin < "EWR";

## origin  count(*)
## ------  --------
## JFK     215941
##
## Run Time: real 0.020 user 0.021148 sys 0.001290
```

```
sqlite> EXPLAIN QUERY PLAN SELECT origin, count(*) FROM flights
  WHERE origin > "EWR" OR origin < "EWR";

## QUERY PLAN
## `--MULTI-INDEX OR
##    |--INDEX 1
##    |  `--SEARCH flights USING COVERING INDEX flights_orig_dest (origin>?)
##    `--INDEX 2
##       `--SEARCH flights USING COVERING INDEX flights_orig_dest (origin<?)
```

What about dest?

```
sqlite> SELECT dest, count(*) FROM flights WHERE dest = "LAX";

## dest  count(*)
## ----  --------
## LAX   16174
##
## Run Time: real 0.027 user 0.016513 sys 0.000237
```

```
sqlite> EXPLAIN QUERY PLAN SELECT dest, count(*) FROM flights WHERE dest = "LAX";

## QUERY PLAN
## `--SCAN flights USING COVERING INDEX flights_orig_dest
```

```
sqlite> SELECT dest, count(*) FROM flights WHERE dest = "LAX" AND origin = "EWR";

## dest  count(*)
## ----  --------
## LAX   4912
##
## Run Time: real 0.001 user 0.000729 sys 0.000208
```

```
sqlite> EXPLAIN QUERY PLAN  SELECT dest, count(*) FROM flights WHERE dest = "LAX" AND origin = "EWR";

## QUERY PLAN
## `--SEARCH flights USING COVERING INDEX flights_orig_dest (origin=? AND dest=?)
```

Group bys

```
sqlite> SELECT carrier, count(*) FROM flights
  GROUP BY carrier;

## carrier  count(*)
## -------  --------
## 9E       18460
## AA       32729
## AS       714
## B6       54635
## DL       48110
## EV       54173
## F9       685
## FL       3260
## HA       342
## MQ       26397
## OO       32
## UA       58665
## US       20536
## VX       5162
## WN       12275
## YV       601
##
## Run Time: real 0.172 user 0.114274 sys 0.0189
```

```
sqlite> EXPLAIN QUERY PLAN SELECT carrier, count

## QUERY PLAN
## |--SCAN flights
## `--USE TEMP B-TREE FOR GROUP BY
```

GROUP with index

```
sqlite> CREATE INDEX flight_carrier ON flights (carrier);

##
## 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