The why of databases

Numbers every programmer should know

Task	Timing (ns)	Timing (µs)
L1 cache reference	0.5	
L2 cache reference	7	
Main memory reference	100	0.1
Random seek SSD	150,000	150
Read 1 MB sequentially from memory	250,000	250
Read 1 MB sequentially from SSD	1,000,000	1,000
Disk seek	10,000,000	10,000
Read 1 MB sequentially from disk	20,000,000	20,000
Send packet CA->Netherlands->CA	150,000,000	150,000

From jboner/latency.txt & sirupsen/napkin-math
Jeff Dean's original talk

Implications for big data

Lets imagine we have a 10 GB flat data file and that we want to select certain rows based on a particular criteria. This requires a sequential read across the entire data set.

If we can store the file in memory:

- $10\ GB \times (250\ \mu s/1\ MB) = 2.5$ seconds

If we have to access the file from SSD:

- $10\ GB \times (1\ ms/1\ MB) = 10$ seconds

If we have to access the file from disk:

- $10\ GB \times (20\ ms/1\ MB) = 200$ seconds

This is just for reading sequential data, if we make any modifications (writing) or the data is

Blocks

Cost: Disk << SSD <<< Memory

Speed: Disk <<< SSD << Memory

So usually possible to grow our disk storage to accommodate our data. However, memory is usually the limiting resource, and if we can't fit everything into memory?

Create blocks - group related data (i.e. rows) and read in multiple rows at a time. Optimal size will depend on the task and the properties of the disk.

Linear vs Binary Search

Even with blocks, any kind of querying / subsetting of rows requires a linear search, which requires $\mathrm{O}(N)$ accesses where N is the number of blocks.

We can do much better if we are careful about how we structure our data, specifically sorting some or all of the columns.

- Sorting is expensive, $\mathrm{O}(N \log N)$, but it only needs to be done once.
- After sorting, we can use a binary search for any subsetting tasks ($\mathrm{O}(\log N)$).
- These "sorted" columns are known as indexes.
- Indexes require additional storage, but usually small enough to be kept in memory while blocks stay on disk.

Databases

SQL

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

- ANSI Standard but with dialect divergence (MySql, Postgre, sqlite, etc.)
- This functionality maps very closely (but not exactly) with the data manipulation verbs present in dplyr.
- We will see this mapping in more detail next time.
- SQL is likely to be a foundational skill if you go into industry - learn it and put it on your CV

R & databases - the DBI package

Low level package for interfacing R with Database management systems (DBMS) that provides a common interface to achieve the following functionality:

- connect/disconnect from DB
- create and execute statements in the DB
- extract results/output from statements
- error/exception handling
- information (meta-data) from database objects
- transaction management (optional)

See r-dbi.org for more details

RSQLite

Provides the implementation necessary to use DBI to interface with an SQLite database.

```
library(RSQLite)
```

this package also loads the necessary DBI functions as well.

Once loaded we can create a connection to our database,

```
con = dbConnect(RSQLite::SQLite(), ":memory:")
str(con)

## Formal class 'SQLiteConnection' [package "RSQLite"] with 5 slots
##   ..@ Id                 :<externalptr>
##   ..@ dbname             : chr ":memory:"
##   ..@ loadable.extensions: logi TRUE
##   ..@ flags              : int 6
##   ..@ vfs                : chr ""
```

Example Table

```
employees = data.frame(
  name   = c("Alice","Bob","Carol","Dave","Eve","Frank"),
  email  = c("alice@company.com", "bob@company.com",
             "carol@company.com", "dave@company.com",
             "eve@company.com",   "frank@comany.com"),
  salary = c(52000, 40000, 30000, 33000, 44000, 37000),
  dept   = c("Accounting", "Accounting","Sales",
             "Accounting","Sales","Sales"),
)
```

```
dbWriteTable(con, name = "employees", value = employees)
## [1] TRUE

dbListTables(con)
## [1] "employees"
```

Removing Tables

```
dbWriteTable(con, "employs", employees)
## [1] TRUE

dbListTables(con)
## [1] "employees" "employs"

dbRemoveTable(con,"employs")
## [1] TRUE

dbListTables(con)
## [1] "employees"
```

Querying Tables

```
(res = dbSendQuery(con, "SELECT * FROM employees"))
## <SQLiteResult>
##   SQL  SELECT * FROM employees
##   ROWS Fetched: 0 [incomplete]
##        Changed: 0

dbFetch(res)
##    name             email salary       dept
## 1 Alice alice@company.com  52000 Accounting
## 2   Bob   bob@company.com  40000 Accounting
## 3 Carol carol@company.com  30000      Sales
## 4  Dave  dave@company.com  33000 Accounting
## 5   Eve   eve@company.com  44000      Sales
## 6 Frank  frank@comany.com  37000      Sales

dbClearResult(res)
## [1] TRUE
```

Creating empty tables

```
dbCreateTable(con, "iris", iris)

(res = dbSendQuery(con, "select * from iris"))
## <SQLiteResult>
##   SQL  select * from iris
##   ROWS Fetched: 0 [complete]
##        Changed: 0

dbFetch(res)
## [1] Sepal.Length Sepal.Width  Petal.Length Petal.Width  Species
## <0 rows> (or 0-length row.names)
```

```
dbFetch(res) %>% as_tibble()
## # A tibble: 0 × 5
## # … with 5 variables: Sepal.Length <dbl>, Sepal.Width <dbl>, Petal.Length <dbl>,
## #   Petal.Width <dbl>, Species <chr>

dbClearResult(res)
```

Adding to tables

```
dbAppendTable(con, name = "iris", value = iris)
## [1] 150
## Warning message:
## Factors converted to character
```

```
(res = dbSendQuery(con, "select * from iris"))
## <SQLiteResult>
##   SQL  select * from iris
##   ROWS Fetched: 0 [incomplete]
##        Changed: 0

dbFetch(res) %>% as_tibble()
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <chr>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
```

Ephemeral results

```
res
## <SQLiteResult>
##   SQL  select * from iris
##   ROWS Fetched: 150 [complete]
##        Changed: 0

dbFetch(res) %>% as_tibble()
## # A tibble: 0 x 5
## # … with 5 variables: Sepal.Length <dbl>, Sepal.Width <dbl>, Petal.Length <dbl>, Petal.Width <dbl>,
## #   Species <chr>

dbClearResult(res)
```

SQL Queries

Connecting

```
cr173@trig2 [class_2021_10_19]$ sqlite3 employees.sqlite
SQLite version 3.34.1 2021-01-20 14:10:07
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 via 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 via 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 via 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,

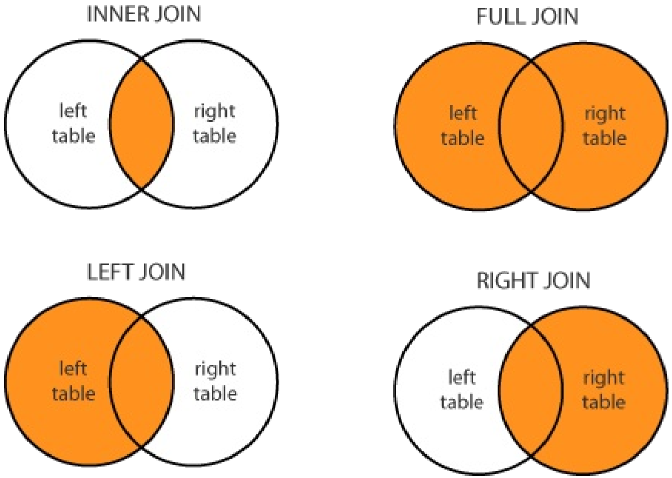
1. The total costs in payroll for this company
2. The average salary within each department

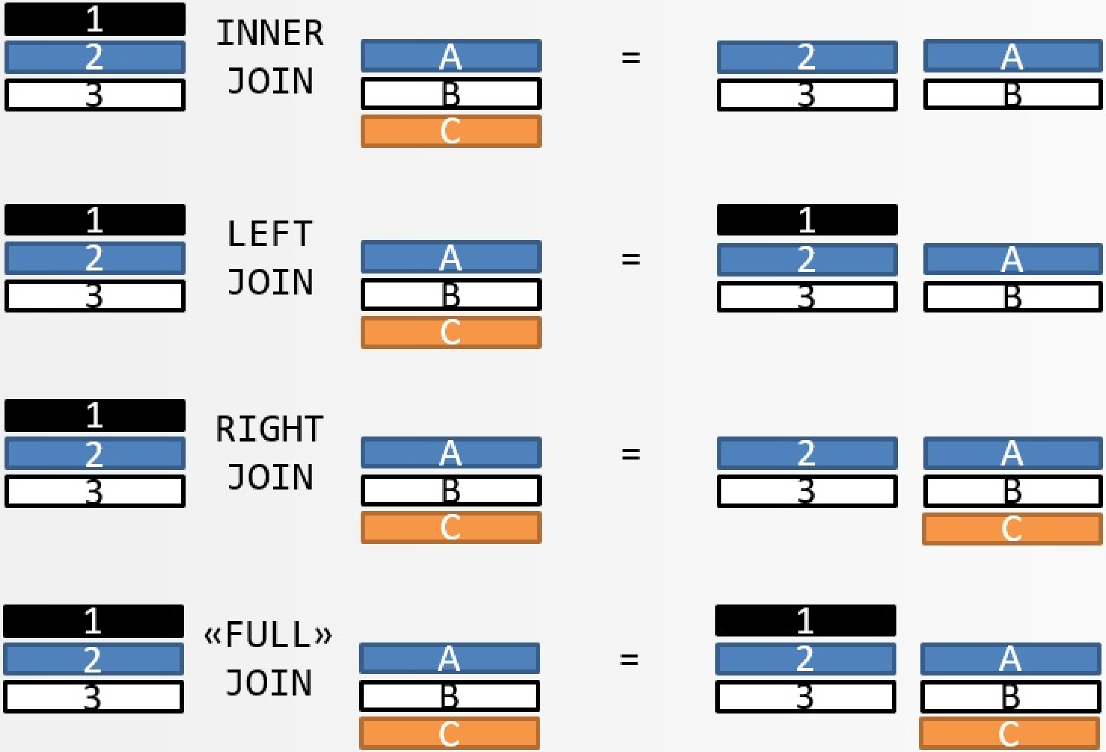
SQLite's aggregation function documentation

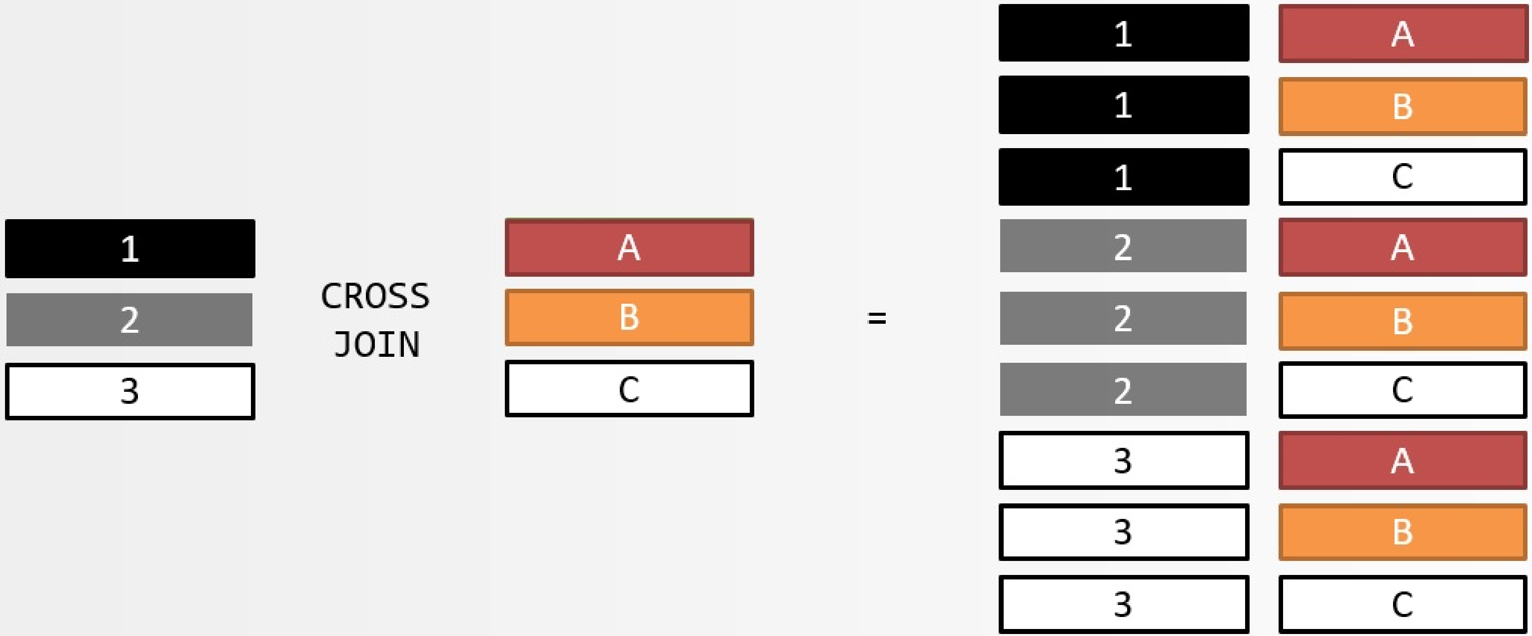
Import CSV files

```
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
```

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
```

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,

```
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
```

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 switch 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)

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
```

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 NOT NULL;

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
```

Excercise 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.