Data Engineering - Lecture 6

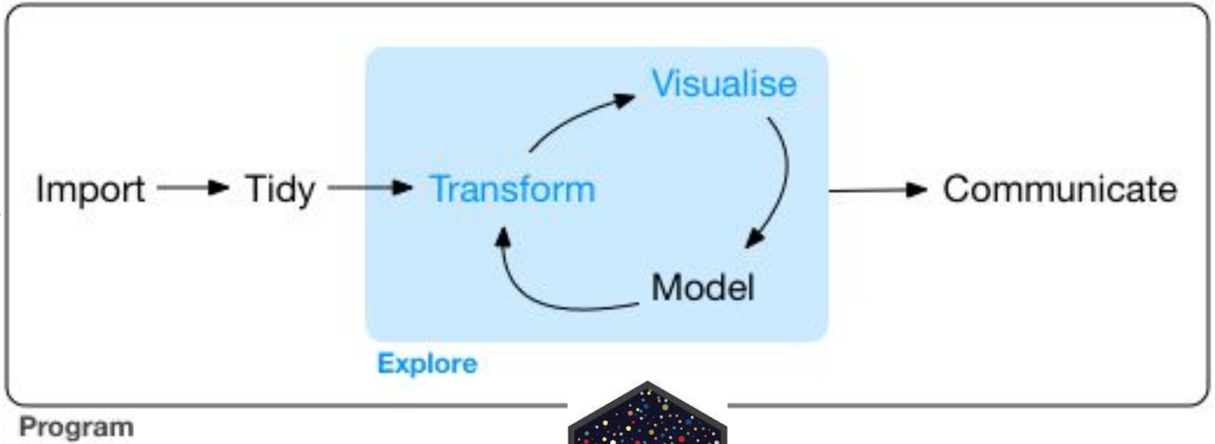
A practical approach to SQL - Part 2

Shamindra Shrotriya (CMU)

So what does a typical **data-driven** *workflow* look like?

Data-driven workflows adopt an interactive pipeline

Most data in industry
lives in **SQL databases**



We spend **most** of our time **here** in this course, e.g., **`dplyr`**, **`ggplot2`**, **`tidymodels`** etc.

Source: [R for Data Science]

Takeaway: being able to **efficiently extract SQL data** is vital for success

Aren't R/python/Julia **alone** sufficient for this purpose?

No - But they work brilliantly **with `SQL`**!

`SQL` databases allow you to **persistently store** and **organize** data

Support a streamlined **E**xtract-**T**ransform-**L**oad (**ETL**) process for streaming data

Provide **access management restrictions** to **specific data**, e.g., health records

Allow for **explicit linkages across tables** (primary and foreign keys)

Enable **indexes** to be defined on tables for efficiency, e.g., **date/time** fields

Takeaway: use R for **accessing subsets** of data from a **`SQL`** database for modeling

Key idea `query`: ***`table(s)`*** ➜ ***`table`***

***SQL** provides a consistent grammar (**S**tructured **L**anguage) for asking and answering questions (**Q**ueries) about your collected data*

SQL tables are nouns, on which you ask targeted queries

Columns (variables)

Observations (rows)

	dest	month	day	mnd	mxd	avd
1	ABQ	12	1	-36	-36	-36
2	ABQ	12	2	-17	-17	-17
3	ABQ	12	3	20	20	20
4	ABQ	12	4	27	27	27
5	ABQ	12	5	32	32	32
6	ABQ	12	6	46	46	46
7	ABQ	12	7	53	53	53
8	ABQ	12	8	114	114	114
9	ABQ	12	9	57	57	57
10	ABQ	12	10	108	108	108

Tables are just **2D representations** of data

A **collection** of **columns** and **observations**

These are similar to data **frames/tibbles** in R

“tibble” even phonetically **sounds like “table”**

You’re already used to them in R - yay!

Takeaway: `data frames` in **`R/Python`** are natural analogues of **`SQL`** tables

SQL grammar comes pre-built with common keywords

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time
1	2013	1	1	517	515	2	830
2	2013	1	1	533	529	4	850
3	2013	1	1	542	540	2	923
4	2013	1	1	544	545	-1	1004
5	2013	1	1	554	600	-6	812
6	2013	1	1	554	558	-4	740
7	2013	1	1	555	600	-5	913
8	2013	1	1	557	600	-3	709
9	2013	1	1	557	600	-3	838
10	2013	1	1	558	600	-2	753

Thousands **more observations**

SQL Code

```
SELECT dest, month, day,
  MIN(arr_delay) AS mnd,
  MAX(arr_delay) AS mxd,
  AVG(arr_delay) AS avd
FROM flights
GROUP BY dest, month, day
ORDER BY dest, month DESC,
         day
LIMIT 10;
```

	dest	month	day	mnd	mxd	avd
1	ABQ	12	1	-36	-36	-36
2	ABQ	12	2	-17	-17	-17
3	ABQ	12	3	20	20	20
4	ABQ	12	4	27	27	27
5	ABQ	12	5	32	32	32
6	ABQ	12	6	46	46	46
7	ABQ	12	7	53	53	53
8	ABQ	12	8	114	114	114
9	ABQ	12	9	57	57	57
10	ABQ	12	10	108	108	108

Takeaway: these keywords (verbs) allow you to systematically query tables (nouns)

`SQL` keywords have a direct **bidirectional** to `dplyr` verbs

`SELECT`	⟷	`select(), mutate(), summarize()`
`FROM`	⟷	specified **input** data frame/tibble
`WHERE`	⟷	`filter()`
`GROUP BY`	⟷	`group_by()`
`HAVING`	⟷	`group_by() %>% summarize() %>% filter()`
`ORDER BY`	⟷	`arrange()`
`LIMIT`	⟷	`“head()” or “tail()”`

Adapted from: Ian Cook

Takeaway: `dplyr` developed this precise relationship to **`SQL` by design** over time

A reminder as to why I use SQL

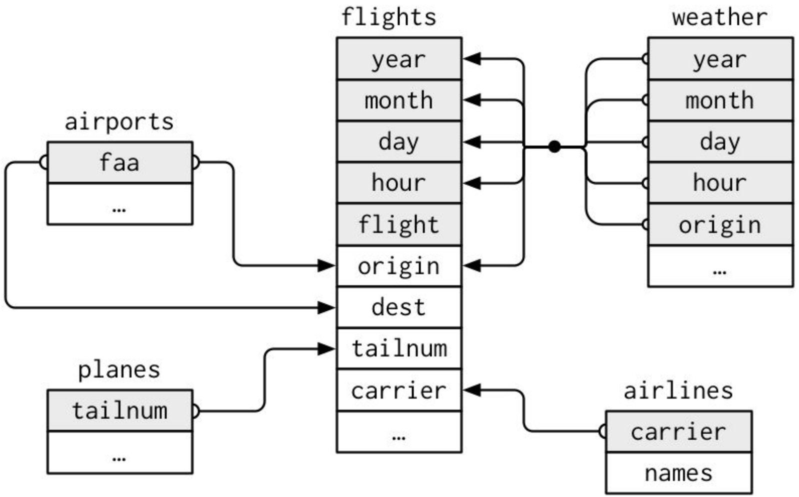
I like using **`SQL`** because it's ***fun*** and ***necessary***

Specifically **`SQL`** allows me to **ask** and **answer** precise **questions** on collected data, in a manner that is both easy to *communicate* and *scales* with data size.

Always first aim to visualize your database before using SQL

We'll use the `nycflights13` database for our analysis

What: Contains flight info for NYC departures to various US destinations in 2013



Source: nycflights13

`flights:` all NYC departures in 2013

`weather:` hourly data for each airport

`planes:` construction info for each plane

`airports:` airport names and locations

`airlines:` two letter carrier codes/names

Takeaway: building this **mental picture** *up front* gets us in the right **SQL mindset**

Let’s run sqlite3 queries **within** `R` for `nycflights13`

sqlite: “small, fast, self-contained, high-reliability, full-featured, SQL database engine”

```
> install.packages(c("dittodb", "RSQLite", "nycflights13"))

> NYC_CONN <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")

> dittodb::nycflights13_create_sql(NYC_CONN)

> fetch_query <- function(query, con = NYC_CONN) {

  return(DBI::dbGetQuery(con, query))

}

> fetch_query("SELECT * FROM flights LIMIT 11")
```

SELECT ⟷ dplyr::**select()**

We can `SELECT` any column we want from a table

Answer to: how can we select specific columns from a table

```
> SELECT <column_name> FROM <table_name>
```

Let's glimpse 10 rows and all variables from the flights data

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

The `*` means return all (any) columns

SQL will return any `10` rows, so the original `flights` order ***may not be*** preserved

Takeaway: don't **assume** that SQL results **implicitly preserve** original data **ordering**

We can `SELECT` any column we want from a table (cont'd)

```
> SELECT dep_time, arr_time, flight FROM flights LIMIT 10;
```

The equivalent `dplyr` code is

```
> flights %>% select(dep_time, arr_time) %>% head(10)
```

Note that **original** `flights` ordering ***is*** preserved in `dplyr`

`SQL` operates on **sets** of observations, which are an unordered collection

We'll later control ordering explicitly in `SQL` using `ORDER BY`

Takeaway: always add a `LIMIT` clause when you are **just** selecting from a table

SELECT ⟷ dplyr::**mutate()**

We can also use `SELECT` to create new variables

Answer to: how can add new columns to a table, e.g., from existing ones?

Let's get a measure of average speed (miles per hour) for each flight

```
> SELECT flight, distance/(air_time/60) AS speed FROM flights LIMIT 10;
```

We created the required column and named it `AS speed`

In `dplyr` we have the `mutate()` verb

```
> flights %>% mutate(speed = distance/(air_time/60)) %>% select(flight,
speed) %>% head(10)
```

Takeaway: `SELECT` serves to **pick existing** columns or to **create new** ones

SELECT ⟷ `dplyr::summarize()`

We can also aggregate on columns using `SELECT`

Answer to: how can create summary statistics across **all** rows?

SQL has built in aggregate functions: **`MIN`**, **`MAX`**, **`COUNT`**, **`SUM`**, **`AVG`**, …

```
> SELECT MIN(air_time) AS min_ar, MAX(air_time) AS max_ar from flights;
```

We didn't need **`LIMIT`** here, since we **returned a single** aggregate observation

We can get the **total number of observations** using **`COUNT(*)`** operator

```
> SELECT COUNT(*) AS num_obs from flights;
```

Takeaway: Aggregations are **most effective** when working across groups of data

WHERE ↔ `dplyr::filter()`

We can filter observations **WHERE** a criteria is met

Answer to: how can we subset observations which meet a given criteria?

Fetch all flights which departed from "JFK" (but limit to 10 observations)

```
> SELECT * FROM flights WHERE origin = "JFK" LIMIT 10;
```

Count flights which did not arrive at "JFK"

```
> SELECT COUNT(*) FROM flights WHERE dest != "JFK";
```

We can also use these comparison operators `=, !=, <, <=, >, >=`

Takeaway: Filtering operations in **SQL** are similar to **R**, except **==** is just **=** in SQL

How about **WHERE** a variable is IN or NOT IN a range?

Find 20 records which have a tail number matching either {“N593JB”, “N532UA”}

```
> SELECT * FROM flights WHERE origin IN (“N593JB”, “N532UA”) LIMIT 20;
```

Flights which did not depart in either {Dec, Jan} and had an arrival delay > 120 mins

```
> SELECT * FROM flights WHERE month NOT IN (1, 12) AND arr_delay > 120
LIMIT 10;
```

We could have written the following in dplyr

```
> flights %>% filter(!(month %in% c(1, 12)) & arr_delay > 120) %>% head(20)
```

Takeaway: It’s helpful to re-write queries in **R**, and pattern match to **SQL**

Missing values are `NULL` in `SQL` and dealt with differently

Get weather records where wind gust is not missing

```
> SELECT * FROM weather WHERE wind_gust IS NOT NULL LIMIT 20;
```

Note: `wind_gust != NULL` does **not work**, `NULL` values don't match this way

In `R`, missing values are `NA` so we could do either of the following in `dplyr`

```
> weather %>% filter(!is.na(wind_gust))

> weather %>% drop_na(wind_gust) %>% head(20)
```

Takeaway: Be **careful** when dealing with missing (**`NULL`**) values in **`SQL`**

`GROUP BY` ↔ `dplyr::group_by()`

We can `GROUP BY` variables and do aggregate calculations

Answer to: how can we compute aggregate summaries by groups across columns?

Get average arrival delay by flight origin

```
> SELECT origin, AVG(arr_delay) AS avd FROM flights GROUP BY origin;
```

Note that we renamed the average arrival delay column `AS avd`

In `dplyr` we could do the following

```
> flights %>% group_by(origin) %>% summarize(avd = mean(arr_delay, na.rm = TRUE))
```

Takeaway: similar verbs have slightly **different implementations** in **R** and **SQL**

We can also GROUP BY multiple variables

Get minimum, maximum, and average arrival delay by month day and destination

```
> SELECT dest, month, day,
         MIN(arr_delay) AS mnd,
         MAX(arr_delay) AS mxd,
         AVG(arr_delay) AS avd
  FROM flights
  GROUP BY dest, month, day
  LIMIT 10;
```

Takeaway: SQL handles the variable groups, you specify **which** variables to group

```
HAVING ⟷ dplyr::group_by() %>%
         dplyr::summarize() %>%
         dplyr::filter()
```

We can filter aggregated values **HAVING** met a condition

Answer to: how can filter on the aggregated values?

Given number of plane engines, how many had more less than 200 manufacturers?

```
> SELECT engines, COUNT(*) AS tot_num
  FROM planes
  GROUP BY engines
  HAVING tot_num < 200;
```

We could have done `HAVING COUNT(*) < 200;`

We can filter aggregated values **HAVING** met a condition

Given number of plane engines, how many had more less than 200 manufacturers?

In **dplyr** we could do

```
> planes %>% group_by(engines) %>%
  summarize(tot_num = n()) %>% filter(tot_num < 200)
```

Or we could use the nice **count** verb to avoid an explicit **group_by**/**filter**

```
> planes %>% count(engines, name = "tot_num") %>% filter(tot_num < 200)
```

`ORDER BY` ↔ `dplyr::arrange()`

We can ORDER BY many columns for displaying output

Answer to: how to **display** tables **sorted** by one or more columns?

Get minimum, maximum, and average arrival delay by month day and destination

```
> SELECT dest, month, day,
           MIN(arr_delay) AS mnd,  MAX(arr_delay) AS mxd,
           AVG(arr_delay) AS avd
  FROM flights
  GROUP BY dest, month, day
  ORDER BY dest, month DESC, day
  LIMIT 10;
```

Takeaway: ordering is by **default ascending**, unless you specify descending

So ***what's*** next...?

So much more - but we’ll aim for the following

Table aliases: shorthand ways to reference specific tables in your queries

Subqueries: queries within queries

JOINS: how to connect information across tables

WINDOW functions: how to run non-aggregated operations across groups

References

Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund. *R for data science*. " O'Reilly Media, Inc.", 2023. [Link]

Wickham H (2022). *nycflights13: Flights that Departed NYC in 2013. R package* version 1.0.2, [Link]

Cook, Ian. *tidyquery and queryparser: Translating SQL Queries to dplyr Pipelines* [Link]

Teate, Renee MP (2021)**.** *SQL for data scientists: a beginner's guide for building datasets for analysis.* [Link]

Evans, Julia *Become a SELECT star* [Link]