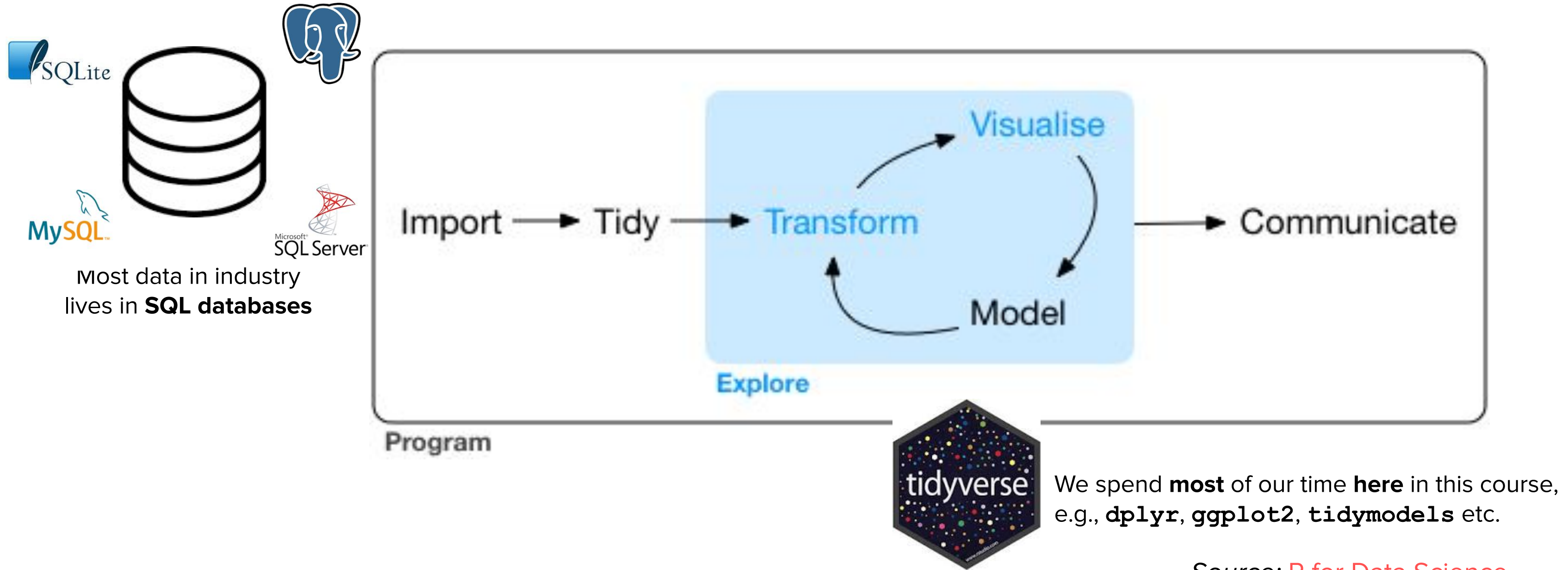
Data Engineering - Lecture 6

A practical approach to SQL - Part 2

Shamindra Shrotriya (CMU)

So ***where*** were we?

Data-driven workflows adopt an interactive pipeline



Source: [R for Data Science]

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

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 grammar comes built-in with keywords (verbs)

	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** follow an ***order* of operations**

Keywords execute in a **diff. order** than which they appear

SQL Code

```
SELECT dest, month, day,
  MIN(arr_delay) AS mnd,
  MAX(arr_delay) AS mxd,
  AVG(arr_delay) AS avd
FROM flights
WHERE month IN (6, 12)
GROUP BY dest, month, day
ORDER BY dest, month DESC,
       day
LIMIT 10;
```

What we see

SELECT → FROM → WHERE → GROUP BY → ORDER BY → LIMIT

How SQL executes

FROM → WHERE → GROUP BY → SELECT → ORDER BY → LIMIT

Adapted from: Julia Evans

Takeaway: grokking the `SQL` execution order enables us to **better reason** with our code

SQL::**KEYWORDS** ↔ dplyr::**functions()**

SQL keywords have a **bidirectional** link 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

SQL::KEYWORDS → dplyr::functions() via tidyquery

tidyquery on tibbles

```
tidyquery::query(
'SELECT dest, month, day,
  MIN(arr_delay) AS mnd,
  MAX(arr_delay) AS mxd,
  AVG(arr_delay) AS avd
FROM nycflights13::flights
GROUP BY dest, month, day
ORDER BY dest, month DESC,
         day
LIMIT 10;')
```

tidyquery → dplyr

```
tidyquery::show_dplyr(
'SELECT dest, month, day,
  MIN(arr_delay) AS mnd,
  MAX(arr_delay) AS mxd,
  AVG(arr_delay) AS avd
FROM nycflights13::flights
GROUP BY dest, month, day
ORDER BY dest, month DESC,
         day
LIMIT 10;')
```

resulting dplyr – amazing!

```
nycflights13::flights %>%
  group_by(dest, month,
day) %>%
  summarise(
mnd = min(arr_delay,
          na.rm = TRUE),
mxd = max(arr_delay,
          na.rm = TRUE),
avd = mean(arr_delay,
           na.rm = TRUE))
%>% ungroup() %>% head(10)
```

Takeaway: tidyquery enables SQL syntax on tibbles and translation to **dplyr**

SQL::KEYWORDS ➜ dplyr::functions() via dbplyr

dbplyr + dplyr

```
flight_summ <-
  tbl(NYC_CONN, "flights")
  group_by(dest, month,
day) %>%
  summarise(
mnd = min(arr_delay,
          na.rm = TRUE),
mxd = max(arr_delay,
          na.rm = TRUE),
avd = mean(arr_delay,
           na.rm = TRUE))
%>% ungroup() %>% head(10)
```

+

dbplyr::render_sql

```
flight_summ %>%
dbplyr::render_sql() %>%
  cat()
```

=

resulting SQL - amazing!

```
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: dbplyr allows for **dplyr** code translation to **SQL**

Adapted from: Source

Recap: `SQL::KEYWORDS` ↔ `dplyr::functions()`

We have the means to **bidirectionally translate** SQL code to dplyr

`tidyquery`: `SQL` ➔ `dplyr`

`dbplyr`: `dplyr` ➔ `SQL`

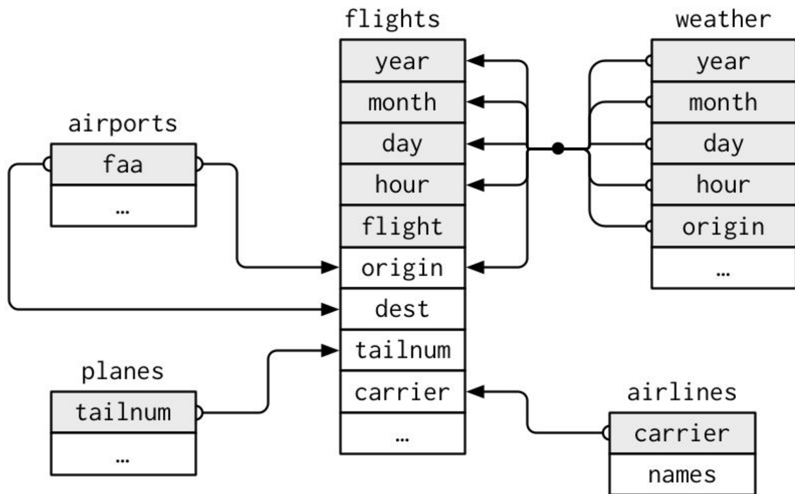
Note: Translations may have **limitations**, e.g., multiple joins in **`tidyquery`**

Takeaway: these amazing tools allow for **bidirectional learning** of **`SQL`** and **`dplyr`**

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**

`SELECT` ↔ `dplyr::select()`

Advanced concepts

We can also `SELECT DISTINCT` variable combinations

What are the unique flight carrier pairings for NYC based flights in 2013?

```
> SELECT DISTINCT carrier, flight FROM flights ORDER BY
carrier, flight;
```

This returns **`DISTINCT`** (unique) flight-carrier combinations

The **`ORDER BY`** is simply for **viewing convenience**

`SELECT` ⟷ `dplyr::mutate()`

Advanced concepts

We can also use CASE WHEN to handle if-then statements

Answer to: how to create columns that are based conditionally on other columns?

Define a **new variable** to classify flights as arriving “early”, “on-time”, or “late”

```
SELECT year, month, day, arr_delay,
       CASE WHEN arr_delay < 0 THEN "Early"
                 WHEN arr_delay = 0 THEN "On Time"
                 WHEN arr_delay > 0 THEN "Late"
                 ELSE "Unknown"
            END AS delay_type
        FROM flights LIMIT 10;
```

Takeaway: **CASE WHEN** enables **if-then-else** logic applied on other columns

SELECT ↔ dplyr::summarize()

Advanced concepts

We can aggregate on columns using `SELECT` + `DISTINCT`

Answer to: how can we count/sum distinct values across a column?

How many **distinct plane types** are there?

```
> SELECT COUNT(DISTINCT type) AS tot_uniq_types from planes;
```

The `DISTINCT` works **across** the **entire** `type` column since we didn't specify a group

Useful to **compare** and **interpret** the difference by using `COUNT(*)` instead

Takeaway: `DISTINCT` clause works well with **aggregate functions (`COUNT`)**

We can aggregate on groups using SELECT + DISTINCT

Answer to: how can we count/sum distinct values by **different groups**?

How many unique plane model types are there **by manufacturer**?

```
> SELECT manufacturer, COUNT(DISTINCT type) AS uniq_types from planes
GROUP BY manufacturer;
```

How many total plane model types are there **by manufacturer**?

```
> SELECT manufacturer, COUNT(*) AS tot_types from planes GROUP BY
manufacturer;
```

Takeaway: DISTINCT aggregations are **very effective** across **groups** of data

subqueries aka *queries* ***within*** *queries*

enable more automation with SQL

We can nest (sub)queries **within** other queries

Answer to: how can we get more automation over filtering, for example?

Q: Count total flights with destination codes **starting with ‘M’**, **grouped by** code

Hmm, let’s first get distinct destination codes starting with ‘M’

```
> SELECT DISTINCT dest FROM flights WHERE dest LIKE "M%";
```

It worked! We also learned to use the `LIKE "M%"` as a SQL wildcard matching

We have the **13 codes**: `("MIA", "MCO", "MSP", "MSY", "MKE", "MEM", "MYR", "MDW", "MHT", "MSN", "MCI", "MTJ", "MVY")`

We can nest (sub)queries **within** other queries (Cont'd)

Now that we have the destination codes, the query is straightforward

```
> SELECT dest, COUNT(*) as tot_flights
  FROM flights
  WHERE dest IN ("MIA", "MCO", "MSP", "MSY", "MKE", "MEM", "MYR", "MDW",
                 "MHT", "MSN", "MCI", "MTJ", "MVY")
  GROUP BY dest
```

Great - but so much **manual typing** in the WHERE clause, so **large room for error**!

We can do better with a subquery approach

Now that we have the destination codes, the query is straightforward

```
> SELECT dest, COUNT(*) as tot_flights
  FROM flights
  WHERE dest IN (SELECT DISTINCT dest FROM flights WHERE dest LIKE "M%")
  GROUP BY dest
```

Amazing - the **`WHERE`** clause could **work directly** with the *output* of our **subquery**

We **don't** have to **change anything** if there is a **new destination** starting with M

Takeaway: subqueries result in more **automated**, **scalable**, and **expressive code**

`SQL` **joins** aka *connecting* tables with other tables

How do we **borrow information** from other tables in `SQL`?

Motivation: How to get the count of total flights in Jun/Jul by plane carrier name?

```
> SELECT carrier, COUNT(*) as tot_flights
  FROM flights
  WHERE month IN (6, 7)
  GROUP BY carrier
```

carrier	tot_flights
9E	2931
AA	5639
AS	122
B6	9606
DL	8377
EV	9097
F9	113
FL	515
HA	61

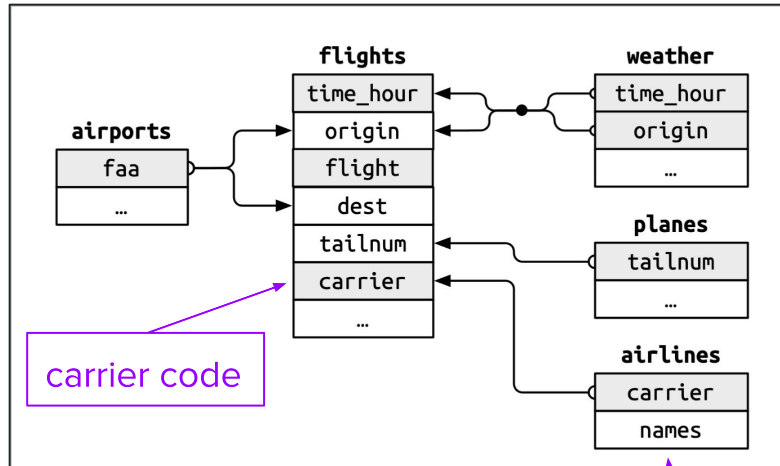
Almost there! But we want the carrier *name*, not carrier *code*.

So *code* **`9E`** corresponds to *name* **`Endeavor Air Inc.`**, for example.

How can we **modify our query** to **obtain** *and* **use** this carrier name information?

We first need to understand *how* the tables are **linked**

We need to understand the **PRIMARY** and **FOREIGN KEY** fields in our database



Source: R for Data Science

carrier code + name

airlines

`carrier` code is a `PRIMARY KEY`, it **uniquely identifies** each observation. It also has the carrier name information in a separate column.

flights

`carrier` code is a `FOREIGN KEY` that corresponds to the `carrier` code `PRIMARY KEY in` **`airlines`**. It is not a unique identifier of observations in **`flights`**.

We can use the idea of a `LEFT JOIN` to link the keys

Let's consider a toy example, taken from R for Data Science

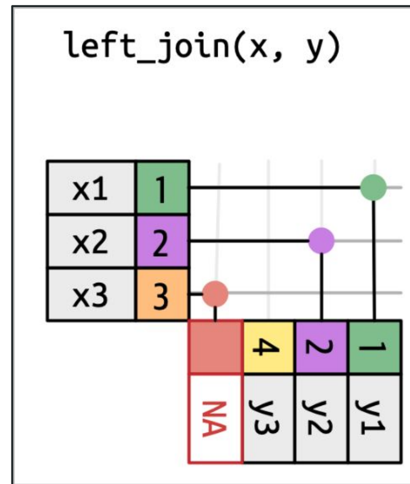
x

key	val_x
1	x1
2	x2
3	x3

y

key	val_y
1	y1
2	y2
4	y3

Goal: to join all **`table y`** values on **`table x`** using keys **`{1, 2, 3, 4}`**, but ensuring that we *retain only keys* from **`table x`**.



The keys on the table on the 'left' will be retained in a LEFT JOIN, i.e., **`table x`**.

key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

Precisely as we wanted

How do we **borrow information** from other tables in `SQL`?

Let's **apply** a `LEFT JOIN` to our original question

```
> SELECT al.name AS airline, COUNT(*) as tot_flights
  FROM flights AS fl
  LEFT JOIN airlines AS al
     ON al.carrier = fl.carrier
  WHERE month IN (6, 7)
  GROUP BY al.name
```

airline	tot_flights
AirTran Airways Corporation	515
Alaska Airlines Inc.	122
American Airlines Inc.	5639
Delta Air Lines Inc.	8377
Endeavor Air Inc.	2931
Envoy Air	4439

The use of table aliases, e.g., `al` for `airlines`, **avoids reference ambiguities**

Takeaway: `JOINS` are powerful, and there are **many** more, i.e., `INNER`, `FULL` ...

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 ***reason with***, ***communicate*** and ***scales*** with data size.

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]