Table manipulations I: dplyr and SQL

2024-09-02

- **M1 MIDS & MFA**
- **Université Paris Cité**
- Année 2024-2025
- Course Homepage

- Moodle

```
stopifnot(
  require(tidyverse),
  require(glue),
  require(cowplot),
  require(patchwork),
  require(nycflights13),
  require(DBI),
  require(RSQLite),
  require(RPostgreSQL),
  require(dtplyr),
  require(dbplyr)
)

old_theme <- theme_set(theme_minimal())
```

Objectives

From the Documentation

> `dplyr` is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:
>
> -`mutate()` adds new variables that are functions of existing variables - `select()` picks variables based on their names. - `filter()` picks cases based on their values. - `summarise()` reduces multiple values down to a single summary. - `arrange()` changes the ordering of the rows.

`dplyr` provides an elegant implementation of table calculus, as embodied by `SQL`.

We will play with the `nycflights13` dataset

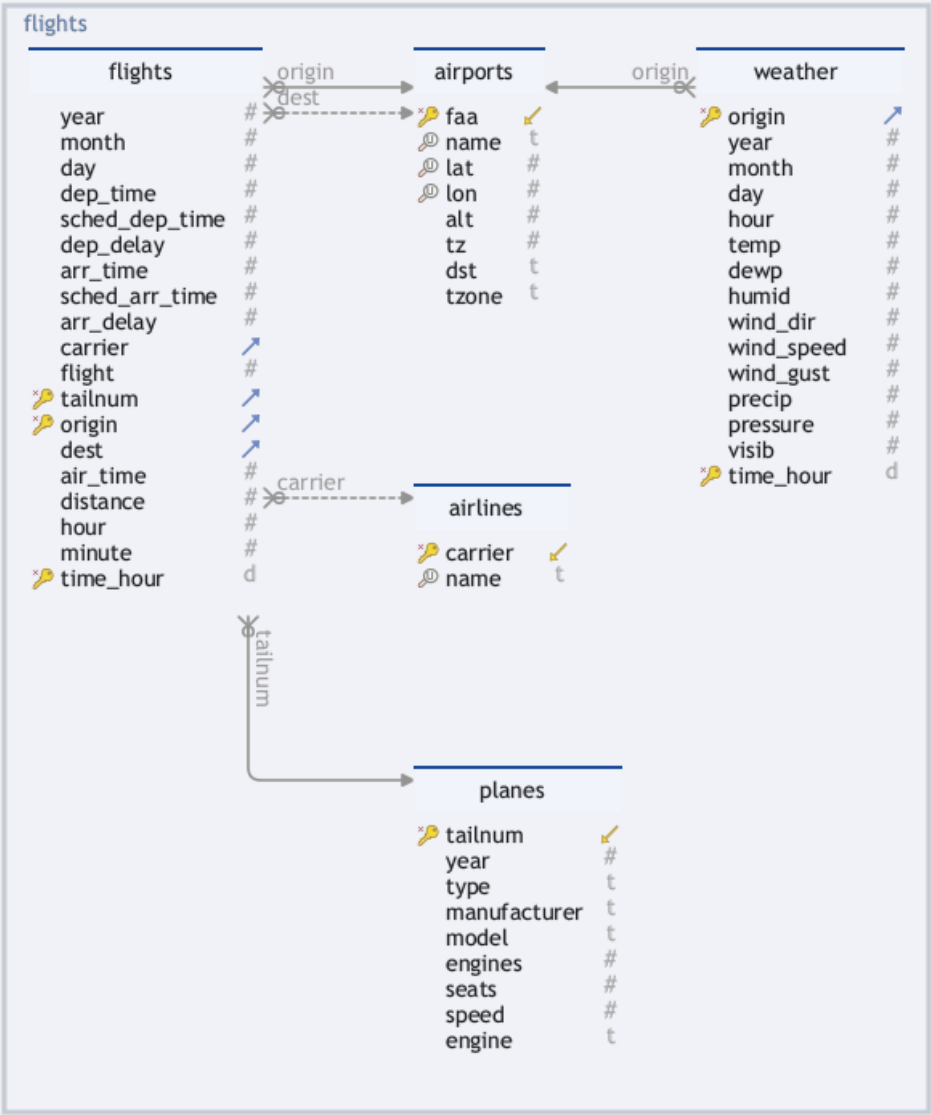

Figure 1: NYCFlights13

Loading `nycflights`

In memory

```
flights <- nycflights13::flights
weather <- nycflights13::weather
airports <- nycflights13::airports
airlines <- nycflights13::airlines
planes <- nycflights13::planes
```

```
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
flights_lite <- copy_to(con, nycflights13::flights)
airports_lite <- copy_to(con, nycflights13::airports)
planes_lite <- copy_to(con, nycflights13::planes)
weather_lite <- copy_to(con, nycflights13::weather)
airlines_lite <- copy_to(con, nycflights13::airlines)
```

Pure relational algebra $\sigma, \pi, \bowtie, \cap, \cup, \setminus$

Projection, π, `select(...)`

Projection of a table R on columns $A_{i_1}, \ldots, A_{i_k}$ results in a table with schema $A_{i_1}, \ldots, A_{i_k}$ and one row for each row of R. Projection is denoted by $\pi(R, A_{i_1}, \ldots, A_{i_k})$.

In SQL this reads as

```
SELECT Ai1, , Aik
FROM R
```

```
R_lite <- copy_to(con, R)
S_lite <- copy_to(con, S)
```

In the sequel, we illustrate operations on the next two toy tables

Table `R`

A	B	C
2024-02-06	4	c
2024-02-07	4	l
2024-02-07	8	n
2024-02-07	7	t
2024-02-06	4	f
2024-02-06	6	j
2024-02-01	7	v
2024-02-08	3	f
2024-02-09	2	q
2024-02-06	6	f

Table `S`

A	D	F
2024-02-01	28	j
2024-02-06	27	u
2024-02-06	28	v
2024-02-06	22	f
2024-02-06	22	z
2024-02-06	27	p
2024-02-06	23	l
2024-02-07	23	y

In Relational Algebra, tables are sets rather than multisets, there are no duplicates. In SQL we are handling multisets of rows, duplicates need to be removed explicitly

```
SELECT DISTINCT Ai1, ..., Aik
FROM R
```

`dplyr` has one verb `select(...)` for π or `SELECT`, and verb `distinct()` for `SELECT DISTINCT ...`.

If we have no intention to remove duplicates:

```
select(R, Ai1, ..., Aik)
# or
R |>
  select(Ai1, ..., Aik)
```

If we want to remove deplicates

```
distinct(R, Ai1, ..., Aik)
# or
R |>
  distinct(Ai1, ..., Aik)
```

$\pi(R, B, C)$ (`SELECT B, C FROM R`) leads to

B	C
4	c
4	l
8	n
7	t
4	f
6	j
7	v
3	f
2	q
6	f

$\pi(R, B)$ and `SELECT DISTINCT B FROM R` lead to

B
4
8
7
6
3
2

> For each departure airport (denoted by `origin'), each day of the year, list the codes (denoted by`carrier‘) of the airlines that have one or more planes taking off from that airport on that day.

Selection, σ, `filter(...)`

Selection of a table R according to condition `expr` is an expression that can be evaluated on each row of R results in a table with the same schema as R and all rows of R where `expr` evaluates to `TRUE`. Selection is denoted by $\sigma(R, \texttt{expr})$.

In SQL this reads as

```
SELECT R.*
FROM R
WHERE expr
```

`dplyr` has one verb `filter(...)` for σ.

> List all the planes built by a manufacturer named like `AIRBUS` between `2005` and `2010`

$\sigma(R, A < \text{2024-02-06} \wedge \text{2024-02-02} \leq A)$ (`SELECT * FROM R WHERE A < CAST('2024-02-06' AS DATE) AND A >= CAST('2024-02-02' AS DATE)`) leads to

A	B	C

`SELECT DISTINCT B FROM R` leads to

B
4
8
7
6
3
2

Joins, $\bowtie$, `xxx_join(...)`

In relational algebra, a θ-join boils down to a selection according to expression θ over a cross product (possibly after renaming some columns)

$$\bowtie (R, S, \theta) \approx \sigma(R \times S, \theta)$$

`dplyr` does not (yet?) offer such a general join (which anyway can be very expensive on a cutting edge RDBMS) several variants of *equijoin*.

`ChatGPT` asserts:

> An equijoin is a type of join operation in relational databases where the join condition involves an equality comparison between two attributes from different tables. In other words, it's a join operation that combines rows from two tables based on matching values in specified columns. These specified columns are usually called the "join columns" or "join keys."

Joins in `dplyr` documentation

- `inner_join()`
- `left_join()`
- `right_join()`
- `full_join()`

but also

- `semi_join()`
- `anti_join()`

the matching columns are stated using optional argument `by=...`

If argument `by` is omitted, `NATURAL JOIN` is assumed.

$\bowtie (R, S)$ (`SELECT * FROM R NATURAL JOIN S`) leads to

```
Joining with `by = join_by(A)`
```

```
Warning in inner_join(R, S): Detected an unexpected many-to-many relationship between `x` and
i Row 1 of `x` matches multiple rows in `y`.
i Row 8 of `y` matches multiple rows in `x`.
i If a many-to-many relationship is expected, set `relationship =
```

```
  "many-to-many"` to silence this warning.
```

```
PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed,
```

Show 10 entries Search:

	A	B	C	D	F
1	2024-02-06	4	c	27	u
2	2024-02-06	4	c	28	v
3	2024-02-06	4	c	22	f
4	2024-02-06	4	c	22	z
5	2024-02-06	4	c	27	p
6	2024-02-06	4	c	23	l
7	2024-02-07	4	l	23	y
8	2024-02-07	8	n	23	y
9	2024-02-07	7	t	23	y
10	2024-02-06	4	f	27	u

Showing 1 to 10 of 28 entries Previous 1 2 3 Next

$\bowtie^{\text{Left outer}}$ (R, S) and `SELECT * FROM R LEFT JOIN S ON (R.A=S.A)` lead to

```
Joining with `by = join_by(A)`
```

```
Warning in left_join(R, S): Detected an unexpected many-to-many relationship between `x` and `y
i Row 1 of `x` matches multiple rows in `y`.
i Row 8 of `y` matches multiple rows in `x`.
i If a many-to-many relationship is expected, set `relationship =
  "many-to-many"` to silence this warning.
```

Show 10 entries Search:

	A	B	C	D	F
1	2024-02-06	4	c	27	u
2	2024-02-06	4	c	28	v
3	2024-02-06	4	c	22	f
4	2024-02-06	4	c	22	z
5	2024-02-06	4	c	27	p
6	2024-02-06	4	c	23	l
7	2024-02-07	4	l	23	y
8	2024-02-07	8	n	23	y
9	2024-02-07	7	t	23	y
10	2024-02-06	4	f	27	u

Showing 1 to 10 of 30 entries Previous 1 2 3 Next

> List weather conditions at departure for all flights operated by airline named `Delta` ....

Agregation, `summarize(...)`

According to ChatGPT:

> In SQL, an aggregation function is a function that operates on a set of values and returns a single aggregated value summarizing those values. These functions are commonly used in SQL queries to perform calculations across multiple rows and produce meaningful results. Some common aggregation functions in SQL include:

- `COUNT`: Counts the number of rows in a result set.
- `SUM`: Calculates the sum of values in a column.
- `AVG`: Calculates the average of values in a column.
- `MIN`: Finds the minimum value in a column.
- `MAX`: Finds the maximum value in a column.

> Count the number of airport whose name starts with `International`

Partition, `group_by`

Following again ChatGPT

> Aggregation functions are often used with the `GROUP BY` clause in SQL queries to group rows that have the same values in specified columns, allowing the aggregation functions to operate on each group separately. This enables powerful analysis and reporting capabilities in SQL, allowing users to extract useful insights from large datasets.

In table R, for each value in column `A` sum the values in column `B`

A	s
2024-02-01	7
2024-02-06	20
2024-02-07	19
2024-02-08	3
2024-02-09	2

`dplyr` offers a `group_by()` verb that proves powerful and flexible. The resulting grouped tibble can be used both for aggregation and for implementing certain kinds of windows.

> For each departure airport, each airline, count the number of flights operated by this airline from this airport.
>
> List the features of planes that have been operated by several airlines

Adding/modifying columns, `mutate(...)`

In the `SELECT` clause of an SQL query, certain columns can be computed. Verb `select` from `dplyr` does not offer this possibility. We have to add the computed columns using verb `mutate` and then to perform projection using `select` (if necessary)

Assume we want to add one day to every value in column `A` from R so as to obtain:

```
# A tibble: 10 x 3
   A              B C
   <date>     <int> <chr>
 1 2024-02-07     4 c
 2 2024-02-08     4 l
 3 2024-02-08     8 n
 4 2024-02-08     7 t
 5 2024-02-07     4 f
 6 2024-02-07     6 j
 7 2024-02-02     7 v
 8 2024-02-09     3 f
 9 2024-02-10     2 q
10 2024-02-07     6 f
```

In SQL we can proceed like this

```sql
SELECT A + 1  AS A, B, C
FROM R
```

Window functions

Asking ChatGPT we obtain

> In SQL, a window function (also known as an analytic function or windowed function) is a type of function that performs a calculation across a set of rows related to the current row within a query result set. Unlike aggregation functions which collapse multiple rows into a single row, window functions operate on a "window" of rows defined by a partition or an ordering.
>
> Key features of window functions include:
>
> Partitioning: The window function can be partitioned by one or more columns, dividing the result set into groups or partitions. The function is applied independently within each partition.
>
> Ordering: The window function can be ordered by one or more columns, defining the sequence of rows within each partition. This determines the rows included in the window for each calculation.
>
> Frame specification: Optionally, a window function can have a frame specification, which further refines the rows included in the window based on their position relative to the current row.
>
> Window functions allow for advanced analytics and reporting tasks that require comparisons or calculations across multiple rows without collapsing the result set. They can be used to compute running totals, calculate moving averages, rank rows within partitions, and perform other complex analyses.
>
> Some common window functions in SQL include `ROW_NUMBER()`, `RANK()`, `DENSE_RANK()`, `NTILE()`, `LAG()`, `LEAD()`, `SUM() OVER()`, `AVG() OVER()`, etc.

If we focus on window queries with a single window built using `PARTITION BY` and `ORDER BY`, we just need to combine, `group_by()`, possibly `arrange()`, and `mutate()`

> For each departure airport, and day, list the 10 most delayed flights.

Use the cheatsheet