Window functions

A **window function** is a variation on an aggregation function. Where an aggregation function, like `sum()` and `mean()`, takes n inputs and return a single value, a window function returns n values. The output of a window function depends on all its input values, so window functions don't include functions that work element-wise, like `+` or `round()`. Window functions include variations on aggregate functions, like `cumsum()` and `cummean()`, functions for ranking and ordering, like `rank()`, and functions for taking offsets, like `lead()` and `lag()`.

In this vignette, we'll use a small sample of the Lahman batting dataset, including the players that have won an award.

```
library(Lahman)

batting <- Lahman::Batting %>%
  as_tibble() %>%
  select(playerID, yearID, teamID, G, AB:H) %>%
  arrange(playerID, yearID, teamID) %>%
  semi_join(Lahman::AwardsPlayers, by = "playerID")

players <- batting %>% group_by(playerID)
```

Window functions are used in conjunction with `mutate()` and `filter()` to solve a wide range of problems. Here's a selection:

```
# For each player, find the two years with most hits
filter(players, min_rank(desc(H)) <= 2 & H > 0)
# Within each player, rank each year by the number of games played
mutate(players, G_rank = min_rank(G))

# For each player, find every year that was better than the previous year
filter(players, G > lag(G))
# For each player, compute avg change in games played per year
mutate(players, G_change = (G - lag(G)) / (yearID - lag(yearID)))

# For each player, find all years where they played more games than they did on average
filter(players, G > mean(G))
# For each, player compute a z score based on number of games played
mutate(players, G_z = (G - mean(G)) / sd(G))
```

Before reading this vignette, you should be familiar with `mutate()` and `filter()`.

Types of window functions

There are five main families of window functions. Two families are unrelated to aggregation functions:

- Ranking and ordering functions: `row_number()`, `min_rank()`, `dense_rank()`, `cume_dist()`, `percent_rank()`, and `ntile()`. These functions all take a vector to order by, and return various types of ranks.

- Offsets `lead()` and `lag()` allow you to access the previous and next values in a vector, making it easy to compute differences and trends.

The other three families are variations on familiar aggregate functions:

- Cumulative aggregates: `cumsum()`, `cummin()`, `cummax()` (from base R), and `cumall()`, `cumany()`, and `cummean()` (from dplyr).
- Rolling aggregates operate in a fixed width window. You won't find them in base R or in dplyr, but there are many implementations in other packages, such as RcppRoll.
- Recycled aggregates, where an aggregate is repeated to match the length of the input. These are not needed in R because vector recycling automatically recycles aggregates where needed. They are important in SQL, because the presence of an aggregation function usually tells the database to return only one row per group.

Each family is described in more detail below, focussing on the general goals and how to use them with dplyr. For more details, refer to the individual function documentation.

Ranking functions

The ranking functions are variations on a theme, differing in how they handle ties:

```
x <- c(1, 1, 2, 2, 2)

row_number(x)
#> [1] 1 2 3 4 5
min_rank(x)
#> [1] 1 1 3 3 3
dense_rank(x)
#> [1] 1 1 2 2 2
```

If you're familiar with R, you may recognise that `row_number()` and `min_rank()` can be computed with the base `rank()` function and various values of the `ties.method` argument. These functions are provided to save a little typing, and to make it easier to convert between R and SQL.

Two other ranking functions return numbers between 0 and 1. `percent_rank()` gives the percentage of the rank; `cume_dist()` gives the proportion of values less than or equal to the current value.

```
cume_dist(x)
#> [1] 0.4 0.4 1.0 1.0 1.0
percent_rank(x)
#> [1] 0.0 0.0 0.5 0.5 0.5
```

These are useful if you want to select (for example) the top 10% of records within each group. For example:

```
filter(players, cume_dist(desc(G)) < 0.1)
#> # A tibble: 1,090 × 7
#> # Groups:   playerID [995]
#>   playerID  yearID teamID     G    AB     R     H
#>   <chr>      <int> <fct>  <int> <int> <int> <int>
#> 1 aaronha01   1963 ML1      161   631   121   201
#> 2 aaronha01   1968 ATL      160   606    84   174
#> 3 abbotji01   1991 CAL       34     0     0     0
#> 4 abernte02   1965 CHN       84    18     1     3
#> # ℹ 1,086 more rows
```

Finally, `ntile()` divides the data up into `n` evenly sized buckets. It's a coarse ranking, and it can be used in with `mutate()` to divide the data into buckets for further summary. For example, we could use `ntile()` to divide the players within a team into four ranked groups, and calculate the average number of games within each group.

```
by_team_player <- group_by(batting, teamID, playerID)
by_team <- summarise(by_team_player, G = sum(G))
#> `summarise()` has grouped output by 'teamID'. You can override using the
#> `.groups` argument.
by_team_quartile <- group_by(by_team, quartile = ntile(G, 4))
summarise(by_team_quartile, mean(G))
#> # A tibble: 4 × 2
#>   quartile `mean(G)`
#>      <int>     <dbl>
#> 1        1      22.7
#> 2        2      91.8
#> 3        3     253.
#> 4        4     961.
```

All ranking functions rank from lowest to highest so that small input values get small ranks. Use `desc()` to rank from highest to lowest.

Lead and lag

`lead()` and `lag()` produce offset versions of a input vector that is either ahead of or behind the original vector.

```
x <- 1:5
lead(x)
#> [1]  2  3  4  5 NA
lag(x)
#> [1] NA  1  2  3  4
```

You can use them to:

- Compute differences or percent changes.

  ```
  # Compute the relative change in games played
  mutate(players, G_delta = G - lag(G))
  ```

  Using `lag()` is more convenient than `diff()` because for `n` inputs `diff()` returns `n - 1` outputs.

- Find out when a value changes.

  ```
  # Find when a player changed teams
  filter(players, teamID != lag(teamID))
  ```

`lead()` and `lag()` have an optional argument `order_by`. If set, instead of using the row order to determine which value comes before another, they will use another variable. This is important if you have not already sorted the data, or you want to sort one way and lag another.

Here’s a simple example of what happens if you don’t specify `order_by` when you need it:

```
df <- data.frame(year = 2000:2005, value = (0:5) ^ 2)
scrambled <- df[sample(nrow(df)), ]

wrong <- mutate(scrambled, prev_value = lag(value))
arrange(wrong, year)
#>   year value prev_value
#> 1 2000     0          4
#> 2 2001     1          0
#> 3 2002     4          9
#> 4 2003     9         16
#> 5 2004    16         NA
#> 6 2005    25          1

right <- mutate(scrambled, prev_value = lag(value, order_by = year))
arrange(right, year)
#>   year value prev_value
#> 1 2000     0         NA
#> 2 2001     1          0
#> 3 2002     4          1
#> 4 2003     9          4
#> 5 2004    16          9
#> 6 2005    25         16
```

Cumulative aggregates

Base R provides cumulative sum (`cumsum()`), cumulative min (`cummin()`), and cumulative max (`cummax()`). (It also provides `cumprod()` but that is rarely useful). Other common accumulating functions are `cumany()` and `cumall()`, cumulative versions of `||` and `&&`, and `cummean()`, a cumulative mean. These are not included in base R, but efficient versions are provided by `dplyr`.

`cumany()` and `cumall()` are useful for selecting all rows up to, or all rows after, a condition is true for the first (or last) time. For example, we can use `cumany()` to find all records for a player after they played a year with 150 games:

```
filter(players, cumany(G > 150))
```

Like lead and lag, you may want to control the order in which the accumulation occurs. None of the built in functions have an `order_by` argument so `dplyr` provides a helper: `order_by()`. You give it the variable you want to order by, and then the call to the window function:

```
x <- 1:10
y <- 10:1
```

```
order_by(y, cumsum(x))
#>  [1] 55 54 52 49 45 40 34 27 19 10
```

This function uses a bit of non-standard evaluation, so I wouldn't recommend using it inside another function; use the simpler but less concise `with_order()` instead.

Recycled aggregates

R's vector recycling makes it easy to select values that are higher or lower than a summary. I call this a recycled aggregate because the value of the aggregate is recycled to be the same length as the original vector. Recycled aggregates are useful if you want to find all records greater than the mean or less than the median:

```
filter(players, G > mean(G))
filter(players, G < median(G))
```

While most SQL databases don't have an equivalent of `median()` or `quantile()`, when filtering you can achieve the same effect with `ntile()`. For example, `x > median(x)` is equivalent to `ntile(x, 2) == 2`; `x > quantile(x, 75)` is equivalent to `ntile(x, 100) > 75` or `ntile(x, 4) > 3`.

```
filter(players, ntile(G, 2) == 2)
```

You can also use this idea to select the records with the highest (`x == max(x)`) or lowest value (`x == min(x)`) for a field, but the ranking functions give you more control over ties, and allow you to select any number of records.

Recycled aggregates are also useful in conjunction with `mutate()`. For example, with the batting data, we could compute the "career year", the number of years a player has played since they entered the league:

```
mutate(players, career_year = yearID - min(yearID) + 1)
#> # A tibble: 20,874 × 8
#> # Groups:   playerID [1,436]
#>   playerID  yearID teamID     G    AB     R     H career_year
#>   <chr>      <int> <fct>  <int> <int> <int> <int>       <dbl>
#> 1 aaronha01   1954 ML1      122   468    58   131           1
#> 2 aaronha01   1955 ML1      153   602   105   189           2
#> 3 aaronha01   1956 ML1      153   609   106   200           3
#> 4 aaronha01   1957 ML1      151   615   118   198           4
#> # ℹ 20,870 more rows
```

Or, as in the introductory example, we could compute a z-score:

```
mutate(players, G_z = (G - mean(G)) / sd(G))
#> # A tibble: 20,874 × 8
#> # Groups:   playerID [1,436]
#>   playerID  yearID teamID     G    AB     R     H    G_z
#>   <chr>      <int> <fct>  <int> <int> <int> <int>  <dbl>
#> 1 aaronha01   1954 ML1      122   468    58   131 -1.16
#> 2 aaronha01   1955 ML1      153   602   105   189  0.519
#> 3 aaronha01   1956 ML1      153   609   106   200  0.519
#> 4 aaronha01   1957 ML1      151   615   118   198  0.411
#> # ℹ 20,870 more rows
```