LOOKING THROUGH THE WINDOW

Jed Rembold

Monday, October 21, 2024

ANNOUNCEMENTS

- HW7 due on Thursday night!
  - Last of the "testable" content for the next exam
  - The last part of the last question does involve some content from today
- Exam 2 is 1 week from Wednesday
  - Review and study materials will be posted on Wednesday
- Polling today: **polling.jedrembold.prof**

REVIEW QUESTION!

Assuming that your server has its timezone set to `'American/Los_Angeles'`, what is the output of the below query?

```sql
SELECT make_time(
    date_part('dow', current_date)::INT,
    date_part('day', now())::INT,
    date_part('hour', localtime)::INT
) AT TIME ZONE 'PDT';
```

A) 01:21:14-07

B) 01:21:13

C) 00:22:12-07

D) 08:21:14

INTRO TO WINDOW FUNCTIONS

WINDOW FUNCTIONS

- One useful piece of kit that is only partially included in the text (in Chapter 11) is that of *window functions*

- A window function is like a mix between a normal column value and an aggregate function
    - Unlike aggregate functions, a window function returns a value for each row
    - Unlike normal column values, a window function can utilize other rows included within its "window" in making an aggregation

- Any normal aggregate function can be used as a window function, though there are specific window functions as well

- Window functions can only be used inside `SELECT` or `ORDER BY` statements
    - They are evaluated *after* any filtering, grouping, or normal aggregations

OVER THE HILL

- The defining characteristic of any window function is the `OVER ()` keyword, which comes after the aggregating window function

- Content inside the `()` determines the "window" of the window function

- By default, if nothing is provided, the entire column is the window

- The below would output the average of the column in every row

  - There will be other ways to do something similar with subqueries, but this is perhaps the most straightforward method

```sql
SELECT AVG(column) OVER()
FROM table;
```

SPECIAL WINDOW FUNCTIONS

DEDICATED WINDOW FUNCTIONS

○ You can use any existing aggregate function as a window function, but there are also **more specific window functions**

Function	Description
`row_number()`	Assigns an ascending row number to each row in a window
`rank()`	Assigns an ascending rank to each row, with possible ties skipping the next value
`dense_rank()`	Assigns an ascending rank to each row, with possible ties not skipping the next value
`first_value($col$)`	Returns the first value in the window of column col
`last_value($col$)`	Returns the last value in the window of column col
`lag($col$, $amt$)`	Returns the previous (or shifted by amt) row of column col
`lead($col$, $amt$)`	Returns the next (or shifted by amt) row of column col
`nth_value($col$, $n$)`	Returns the nth row of column col (`NULL` if doesn't exist)

CHANGING THE WINDOW

DETERMINING ORDER

- Often, to be useful, you may want to define an ordering inside the `OVER()` statement

- As soon as you specify an ordering, **the default window changes**

  - By default, each window now encompasses everything from the start, up to (and including) that current row

    - Easiest to see with classic aggregate functions

```
SELECT COUNT(*) OVER( ORDER BY column )
FROM table;
```

TWEAKING THE WINDOW

○ You can tweak this window by specifying the starting and stopping point, using the syntax:

> ...*type* BETWEEN *offset* PRECEDING AND *offset* FOLLOWING

which appears in the OVER clause, after any provided ordering

  ○ *type* can be either ROWS , RANGE , or GROUP

  ○ *offset* can be

    ○ an non-null, non-negative integer or UNBOUNDED if the type is ROWS or GROUP

    ○ an value that makes sense to add or subtract from the ordered column if the type is RANGE

EXCLUDING THE WINDOW

○ Can also exclude the current row or group from the window:

```
...  EXCLUDE  type
```

○ *type* here can be:

- ○ `CURRENT ROW` , which excludes the current row from the window
- ○ `GROUP` which excludes the current row and all other rows currently tied with it
- ○ `TIES` which just excludes the tied rows, but keeps the current row

PARTITIONING

- Additionally, you can specify a partition for the window
- The default partition is the entire column
- Partitioning here is determined *before* the rows comprising the window are computed
  - This has a similar feel to `GROUP BY`, but *every* row will get a value here
- Window functions evaluated within each window within each partition

```
SELECT AVG(column) OVER (
    PARTITION BY column
  )
FROM table;
```

ACTIVITY!

A SWEET DATASET

- Contained **here** is an SQL file to generate a table of Halloween trick-or-treater data, which contains two columns:
  - The timestamp when a trick-or-treater visited and was given a piece of candy
  - What candy was given to the trick-or-treater
- You started out with a supply of 50 of each type of candy
- You will have several questions you are trying to answer.

TODAY'S GROUPS

- Group 1: Tippy, Dayton, Sam J
- Group 2: AJ, Myles, Matthew, Nick
- Group 3: Hannah, Grace, Michael
- Group 4: Aurora, Mallory, Greg
- Group 5: Jordan, Marcus, Evan
- Group 6: Harleen, Jerrick, Jack
- Group 7: Finn, Connor, Sam H
- Group 8: Haley, Tiffany, Sergio

Front of room

6	7	8
3	4	5
1	2	

Back of room

Door

Group Areas
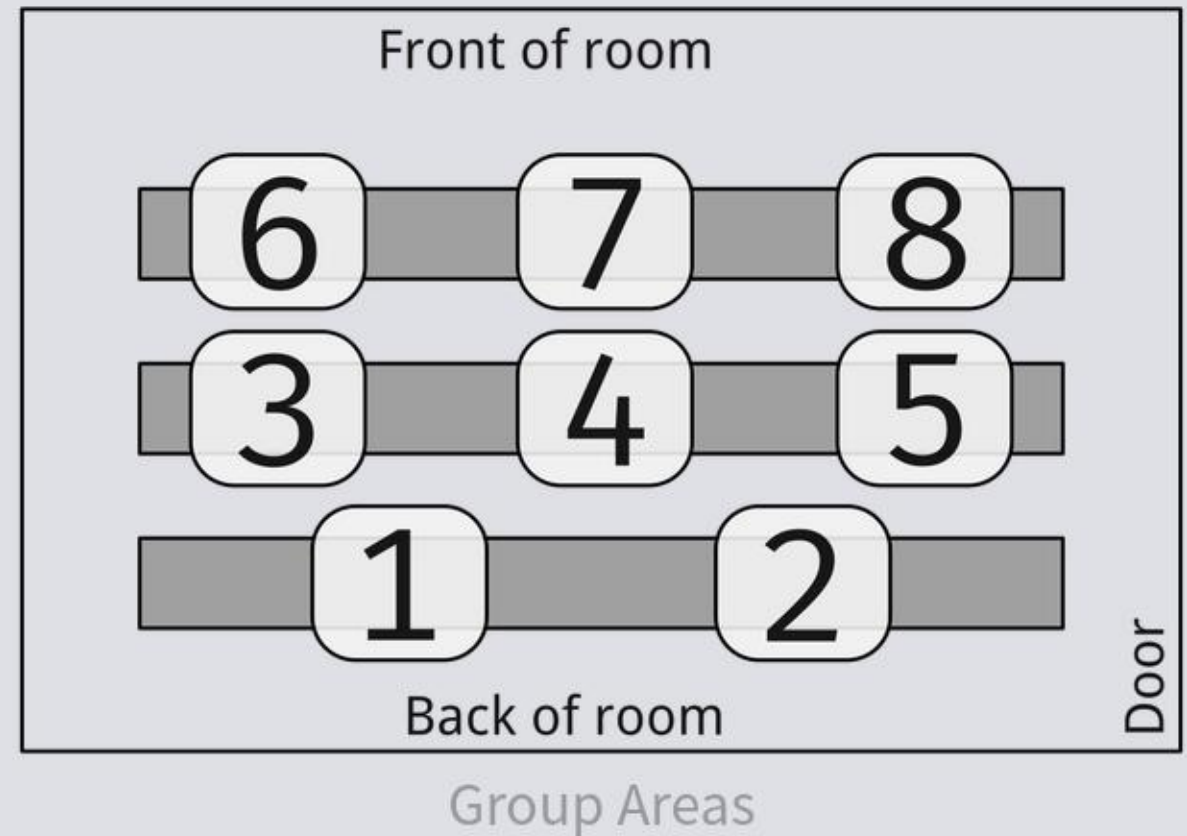
QUESTIONS

- Reminder: You started out with a supply of 50 of each type of candy
- Using the dataset, answer the following questions:
  - Between the 20th and 30th minutes, what were the three most popular candies given?
  - What candies did you never run out of?
  - At what time did you run out of Skittles?
  - What times did you run out of the three most popular candies?
  - Construct a 5-minute rolling average of the number of trick-or-treaters you saw each minute of the evening. According to this, about when were things busiest?