FULL SCALE MINING

Jed Rembold

Wednesday, November 6, 2024

ANNOUNCEMENTS

- Homework 8 due tomorrow
- Test 2 results back on Monday
- TechBytes tomorrow! Where does ethics belong in the data science curriculum?
- Women in Tech: November 13th
  - RSVP here
  - Both speakers are awesome!
- Polling today: polling.jedrembold.prof

REVIEW QUESTION

Which of the below phrases would match to the following regular expression:

```
[a-z]+-\d{2}$
```

A) June-13-2022

B) dec-8

C) Jan-feb-28

D) 9-11

REGULAR EXPRESSIONS IN SQL

BACK TO SQL

- One of the main ways we previously used pattern matching was for filtering
- You can also use regexes for pattern matching!
  - `~` is a case sensitive match using the following regex
  - `~*` is a case insensitive match using the following regex
  - Either can have a `!` in front to negate the search (where things do not match the regex)

```
SELECT colname
FROM tablename
WHERE colname ~ '[a-z]*\s\d{2}';
```

EXTRACTING DATA

- Another hugely common use of regex is to extract only the data you want from a much larger string
- This can be particularly useful when cleaning data or constructing useful database tables
- If you have just a single piece of information to extract, `SUBSTRING(`*text*`, `*regex*`)` is probably the most straightforward method
  - If no capture group provided, the entire match is returned
  - If a single capture group is provided, only that will be returned
  - Just the first matching instance is returned

```
SELECT substring('today is November 6, 2024', '(\w+) \d{2}');
```

EXTRACTING MORE DATA

- If you want to capture multiple groups at once, `regexp_match(`*`str`*`, `*`regex`*`)` is what you want
  - What is returned is whatever is in any *capture groups* you may have included in your regex, or the entire match if there are no capture groups
  - Output is returned as an array, to allow for potentially multiple capture groups
  - You can index values out of the array or "unpack it" with `UNNEST`

```
SELECT regexp_match(
  'today is November 6, 2024',
  '(\w+) \d{2}.*(\d{4})')
);
```

REGULAR SPLITTING

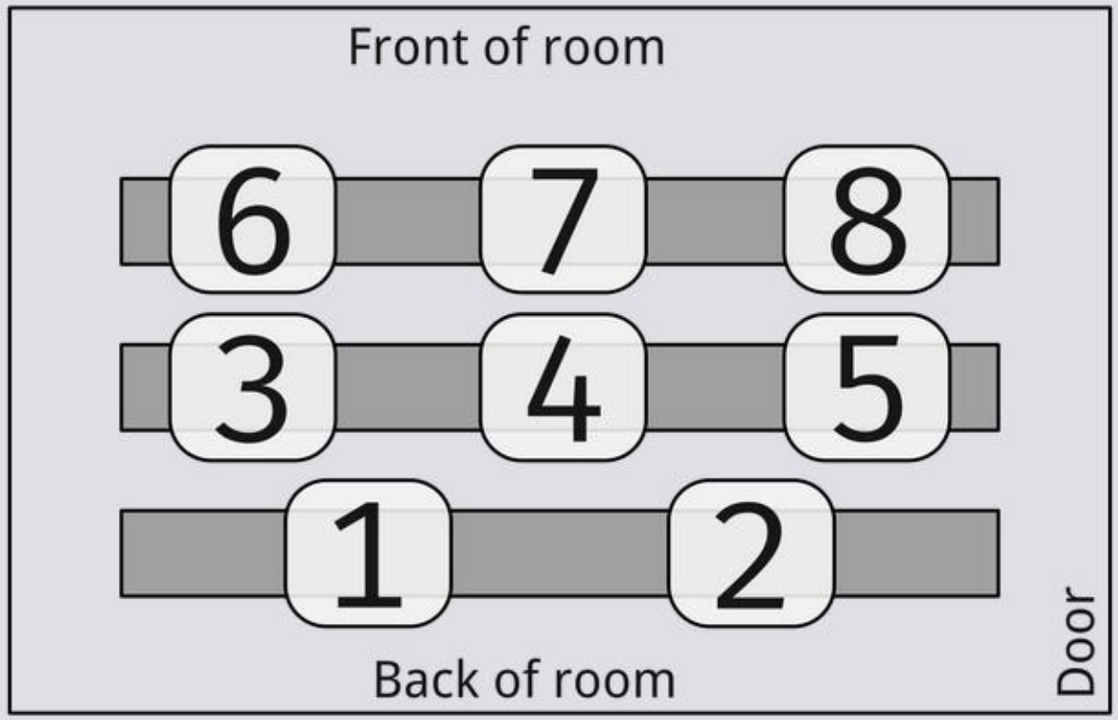
- You can also use regular expressions to replace or split text
- `regexp_replace(`*text*`, `*re*`, `*replacement*`)` will replace the first matches of re in text with the replacement
  - Add another `'g'` argument on the end to do so globally (replacing all matches)
- `regexp_split_to_array(`*text*`, `*re*`)` will split the text into an array on each match to re
- `regexp_split_to_table(`*text*`, `*re*`)` will split the text into a table column on each match to re. This is just like unnesting the array.

NOW TO YOU!

- In the pairings below, take a look at this CSV file, which contains a simple subset of artists and dimensions from the MoMA data set
- Create the simple table and import in the data from the CSV
- See if you can achieve the following using regular expressions:
  - Create and populate new columns for first, middle, and last name
  - Create and populate new columns to hold the width and height in inches (in `1 1/4` form)
  - If the above was easy and you finish quickly, create new columns to hold decimal equivalents of the width and height

TODAY'S GROUPS

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



Group Areas

FULL TEXT SEARCHING

FULL TEXT SEARCH

- Pattern matching or regular expressions are great for parsing longer text for particular known patterns
- What if you want more general information about the text within a longer passage?
  - We need some different tools
- Using these new tools will require using some new functions and data types:
  - `tsvector`
  - `tsquery`

TEXT TO VECTOR

- English (and other languages) have many connecting words that help convey meaning, intent, or relationships
  - to, on, at, with, of, etc.
- These are commonly not what are desired when searching text for particular ideas
- Instead it is useful to focus on *lexemes*
  - A lexeme is an abstract unit of meaning that underlies a set of words
  - RUN: run, runs, ran, running
- Postgres's `to_tsvector` will break a string down into its component lexemes, and keep track of where each occured in the original string

```
SELECT to_tsvector('I flew back to Salem on Monday');

>> 'back':3 'flew':2 'monday':7 'salem':5
```

LEXEME QUERIES

- Text that you want to search through will need to be converted to a tsvector through `to_tsvector`
- Text that you want to search for will be converted using `to_tsquery`
- `to_tsquery` takes a sequence of words with symbols connecting them conveying relationships
  - `&` - and
  - `|` - or
  - `!` - not
  - `<->` - followed by

```
SELECT to_tsquery('fly & monday')
```

COMBINING VECTORS AND QUERIES

- To actually complete a full text search, you ask Postgres to look through a `tsvector` object for a particular `tsquery`
- The syntax to do so utilizes the match operator, which is two "at" symbols: `@@`

```
SELECT some tsvector @@ some tsquery;
```

- Using `@@` is a true/false search, so the query is either found or not
  - This means it can be used for filtering with `WHERE` as well!

INDEXING TSVECTORS

- Individual `tsvector`s can not be easily ordered, so indexing a column with `tsvector` contents using the normal B-Tree method would not be effective
- For this sort of content, use a *Generalized Inverted Index* or GIN indexing method instead

```
CREATE INDEX index name ON table
USING GIN(column);
```

- Seriously consider adding an index to your `tsvector` column, as it can *significantly* speed up these sorts of searches

GETTING MORE INFO

- Sometimes it can be useful to get a bit more information about *where* a match shows up in the text
- The `ts_headline` function can capture snippets of text around a match and display them
  - `ts_headline` operates on the original text, not the tsvector! This will absolutely make it slower, so use it wisely!
- There are a few required parameters and some options for `ts_headline`:
  - the text to search
  - the tsquery to look for
  - Other options appear in an option string:
    - `StartSel` / `StopSel` : the delimiters that will showcase the word
    - `MinWords` / `MaxWords` : the min or max number of words to show around the match
    - `MaxFragments` : a max number of fragments to show if the match occurs multiple times

RANKING

- Sometimes a query might return many results, such that you would want a method to rank or compare them and only select the “best”
- Ranking by relevancy is vague and tends to be very application specific, but it can be useful
- `ts_rank` will give an arbitrary rank based on how many times your query words appear in the text
  - This might mean that longer texts will always receive a higher rank! You can normalize by the text length by providing an extra numeric code as a third argument (2 or 8 probably best)
- `ts_rank_cd` does similar, but also considers the proximity of searched lexemes
- Both functions take 2 required arguments:
  - a tsvector of the contents to be ranked
  - a tsquery determining how they will be ranked

YOUR TURN!

The file here contains the SQL commands to generate and populate a simple table `alice` which hold the raw chapter contents of the book: Alice in Wonderland. You will need to set up your own `tsvector` column and index. In the same groups as earlier, see if you can use the data to answer the following:

- In what chapters does the “Chesire cat” appear?
- In what chapter does the word “mushroom” appear the most? How many times does it appear?