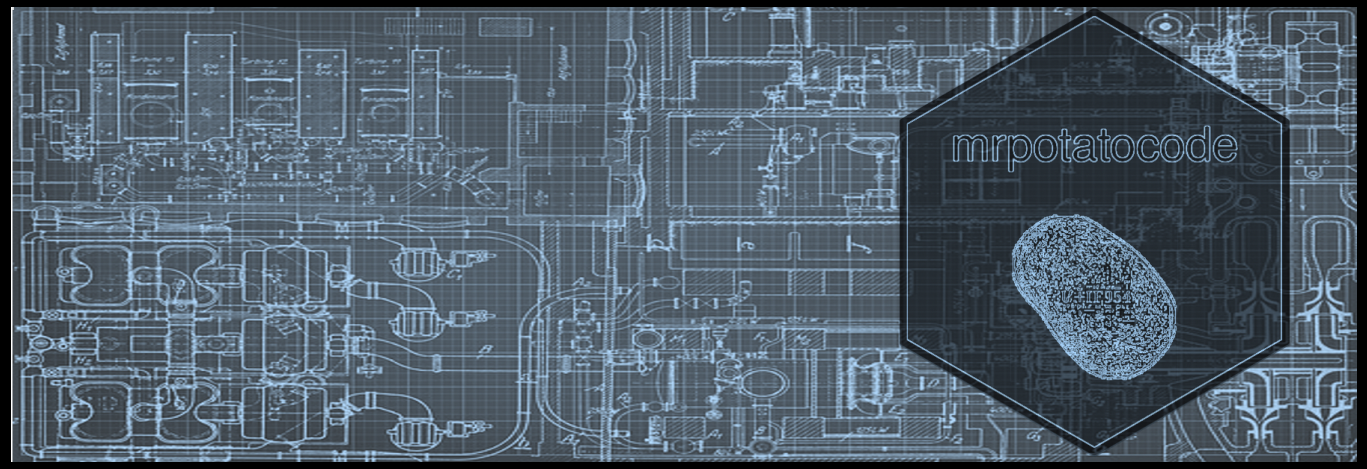



# Introduction to Data Access and Storage

Thomas Rosenthal - DSI @ UofT

Module 03

# Essential Techniques:

Subqueries

Temporary Tables & CTEs

Windowed Functions

Datetime Functions

ISNULL

# Essential Techniques:

Subqueries

Temporary Tables & CTEs

Windowed Functions

Datetime Functions

ISNULL

# Subqueries

- SQL allows us to query the results of another query
    - We call this a subquery
- Subqueries can be used in both `JOIN` and `WHERE` clauses
    - In the case of `JOIN`:
        - you want the subquery to add columns to your output
        - you are using a subquery because you are joining complex criteria that require manipulation
            - it is often the case that you are joining two or more tables within a subquery to another table
    - In the case of `WHERE`:
        - you want to filter results
        - you are using a subquery because it is simpler than joining and filtering the columns otherwise
        - it's important to note: *you can only return a single column* in your subquery
            - **Why do we think this is?**
- In a subquery, all columns need to be uniquely named
- Subqueries can usually be run for testing purposes by highlighting them, IDE dependent

# Subqueries

(Subqueries live coding)

# Essential Techniques:

Subqueries

Temporary Tables & CTEs

Windowed Functions

Datetime Functions

ISNULL

# Temporary Tables & CTEs

## Temporary Tables

## CTEs

# Temporary Tables & CTEs

Temporary Tables

CTEs

# Temporary ("temp") Tables

- Table objects created on the fly
- Automatically saved to a reserved `temp` schema
- Accessible across SQL queries in the same session
- Cleared from memory when SQL is closed (or the server connection is terminated)
- Temporary tables can be chained in the same query
    - You can place one temporary table into another
- Must be dropped (deleted from memory) to recreate them with the same name
- Some older versions of SQL don't allow temporary tables
- They are *fantastic placeholders*
    - **What scenarios can we think of where a temporary table would be particularly useful?**

# Temporary Tables

(Temporary Table live coding)

# Temporary Tables & CTEs

Temporary Tables

CTEs

# Common Table Expressions (CTEs)

- Similar to temporary tables
  - CTEs were developed *before* temp tables
    - some SQL versions/flavours (especially much older ones) might not support temp tables, so CTEs are an important skill
- Instantiated query results created on the fly
  - utilize the `WITH` command
    - many RDBMs require a semicolon terminating the `WITH` clause
    - multiple CTEs don't use more than one `WITH` clause, but rather follow one another with a comma
  - need to be written *before* the final `SELECT` statement
- Sometimes easier than a subquery
  - if subqueries become overly complex, they can be harder to read
- Stored in memory
- Limited to your current query window only

# Common Table Expressions (CTEs)

(CTEs live coding)

# Essential Techniques:

Subqueries

Temporary Tables & CTEs

Windowed Functions

Datetime Functions

ISNULL

# Windowed Functions

Purpose

OVER

PARTITION BY

ROW_NUMBER()

Other Windowed Functions

# Windowed Functions

Purpose

OVER

PARTITION BY

ROW_NUMBER()

Other Windowed Functions

# Purpose

- Windowed Functions allow us to create groupings within groupings ("partitions")
- Allow for greater complexity than simple SQL
    - In Module 2, we mentioned briefly a rolling total, e.g. a `SUM` and a `COUNT`; windowed functions allow us to return these types of results
- Often used with a subquery
    - One of the most common techniques is creating a row number `ROW_NUMBER()` per group
        - when combined with `ORDER BY`, the associated row number will be the *highest* or *lowest* per grouping
        - this allows you to select the min or max by setting the row number = 1 in the "outer" query (i.e. not the "inner" subquery)

# Windowed Functions

Purpose

OVER

PARTITION BY

ROW_NUMBER()

Other Windowed Functions

# OVER

- Syntax for windowed function always requires the `OVER` clause
  - `{desired_windowed_function} OVER (ORDER BY [a column])`
- The `ORDER BY` clause is required
- Think of the `OVER` clause as applying the function of your choice
  - e.g. create row numbers based on the ordering of this column
  - e.g. rank these values from highest to lowest

# Windowed Functions

Purpose

OVER

PARTITION BY

ROW_NUMBER()

Other Windowed Functions

# PARTITION BY

- Within an `OVER` clause, we can optionally use `PARTITION BY` to create groupings for the function to be applied to
  - `{desired_windowed_function} OVER (PARTITION BY [a column] ORDER BY [a column])`
- Now, the function is being applied to different groups
  - e.g. rank these values from highest to lowest within these groups
    - the ranking will restart for each group
    - think of this like the Olympics: the top three competitors for each event get gold, silver, and bronze — the `PARTITION BY` is the event, the `ORDER BY` is the time `ASC` or points `DESC` that determine the outcome of the event
- Both the `PARTITION BY` and `ORDER BY` arguments can take more than one column
  - e.g. life expectancy by country by continent

# Windowed Functions

Purpose

OVER

PARTITION BY

ROW_NUMBER()

Other Windowed Functions

# ROW_NUMBER()

- `ROW_NUMBER()` is the simplest windowed function, but also one of the most useful
  - There are no mathematical functions being applied, just an incremental value by group
  - Determining the top (or bottom) per group is often done through `ROW_NUMBER()`
- `ROW_NUMBER()` might feel a bit like ranking `RANK()`...but it's not quite
  - What is the difference between `ROW_NUMBER()` and `RANK()`?

# Windowed Functions

Purpose

OVER

PARTITION BY

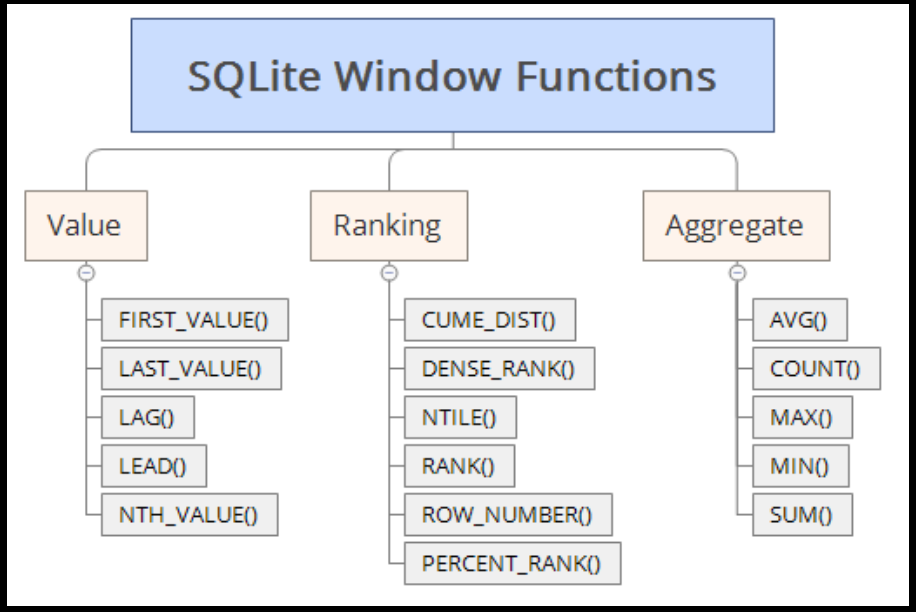
ROW_NUMBER

Other Windowed Functions

# Other Windowed Functions

- SQL supports quite a few other windowed functions
- `NTILE` for example will assign rows to buckets (4: quartile, 5: quintile, 10: decile, etc)
  - As such, the `NTILE` function requires an argument passed to it
    - `NTILE(4) OVER (PARTITON BY...ORDER BY...)`

- `LAG` and `LEAD` allow us to create an offset of another column
  - e.g. show a previous_year_total next to a current_year_total for easy comparison
- Knowing how and why to use these can make querying a lot easier



# Windowed Functions

(Windowed Functions live coding)

# Essential Techniques:

Subqueries

Temporary Tables & CTEs

Windowed Functions

Datetime Functions

ISNULL

# Datetime Functions

Formats

'NOW'

STRFTIME

Adding Dates

Difference between Dates

# Datetime Functions

Formats

'NOW'

STRFTIME

Adding Dates

Difference between Dates

# Formats

- Date formats vary widely in SQL databases
  - A general rule of thumb when working with multiple date fields is to force them all into a similar format
    - This may seem obvious, but different source systems may write dates different in SQL DBs
- It is not uncommon to store date values as integers YYYYMMDD to increase optimization and decrease storage size
- Manipulating dates varies by flavour
- SQLite is *less* flexible with dates, requiring all dates to either be:
  - "YYYY-MM-DD" *strings*
  - Julian Day *fractions*
  - Seconds from Unix Time *integers*

# Datetime Functions

Formats

'NOW'

STRFTIME

Adding Dates

Difference between Dates

# 'NOW' (or GETDATE() or DATE, flavour dependent)

- These functions (there are actually more of them) get the current date and time
  - Some will return UTC time if requested (this can be useful) e.g. `GETUTCDATE()`
- When combined with other Datetime functions, this can serve as a dynamic value
  - e.g. "yesterday", "last year", and so on
- SQLite uses `DATE()` and `DATETIME()` (without any arguments) *or* `DATE('now')` *or* simply `'now'`
  - SQLite's flexibility with single vs double quotes is important here:
    - "now" is the word now
    - 'now' is July 23, 2022

# 'NOW'

('NOW' live coding)

# Datetime Functions

Formats

'NOW'

STRFTIME

Adding Dates

Difference between Dates

# STRFTIME

- `STRFTIME` converts DATE and DATETIME values into different formats
- `STRFTIME` also allows you to extract specific "dateparts"
    - e.g. `SELECT STRFTIME('%Y','NOW')`
- The first argument of `STRFTIME` is flexible — you can specify more than one datepart at a time *and* any formatting
    - e.g. `SELECT STRFTIME('%Y-%m','NOW')` would return 2022-07
- `STRFTIME` also allows modification to date dynamically
    - e.g. `SELECT STRFTIME('%Y-%m-%d', '2022-07-24', 'start of month')`
    - How do we go about subtracting dates rather than adding them?
- Modifiers include:
    - +/- N years/months/days/hours/minutes/seconds
    - start of year/month/day
    - weekday
- Be mindful: because outcome is a *string*, modification should be done within the `STRFTIME` argument to ensure it is correct
- Some flavours have built in convenience dateparts, like `YEAR`, `MONTH`, etc that make extracting values a bit easier

# STRFTIME

(`STRFTIME` live coding)

# Datetime Functions

Formats

'NOW'

STRFTIME

Adding Dates

Difference between Dates

# Adding Dates (sometimes DATEADD or DATE_ADD, flavour dependent)

- SQLite supports two means of adding increments of time to a date:
  - `STRFTIME` as mentioned previously
  - Using `DATE`
    - e.g. `SELECT DATE('2022-07-24', 'start of month')`
- Both of these methods allow you to chain modifiers
  - e.g. `SELECT DATE('2022-07-24', 'start of month', '-1 day')`

**What do we see as the difference between these?**

- This syntax is fairly unique to SQLite, but is conceptually the same, so briefly I will touch on `DATEADD`
  - Generally, we specify a datepart, add/subtract a value, and the date

# Datetime Functions

Formats

'NOW'

STRFTIME

Adding Dates

Difference between Dates

# Difference between Dates (an extension of STRFTIME or DATEDIFF, flavour dependent)

- The difference between dates can vary in complexity
- We can use `STRFTIME`, subtracting the two dates from one another, using '%s' as our unit
  - e.g. `SELECT (STRFTIME("%s", Date1) - Date2) / {increment, e.g. 3600.0 for hours, 60.0 for minutes, etc}`
    - Be sure to include *.0* for float precision: `ROUND` or `CAST` to integer if desired
  - `STRFTIME` works well for calculating months and years
  - e.g., months until winter ❄ `SELECT STRFTIME('%m','2022-12-21') - STRFTIME('%m','NOW')`

- We can use also use `JULIANDAY`:
  - Julian Days are *fractional* by nature and result in a difference of days
  - e.g., difference in hours `SELECT CAST((JULIANDAY(Date1) - JULIANDAY(Date2) * 24) AS INT)`

- This syntax is also fairly unique to SQLite, but is conceptually the same, so briefly I will touch on `DATEDIFF`
  - Generally, we specify a datepart, startdate, and enddate

# Essential Techniques:

Subqueries

Temporary Tables & CTEs

Windowed Functions

Datetime Functions

ISNULL

# ISNULL (and/or COALESCE, flavour dependent)

- `ISNULL` allows us to return a replacement value for NULLs
  - Replacement values can be another column, a calculated value, or static
    - e.g. when col1 is NULL, it is replaced with values from col2
      - values from col2 are only present if col1 is NULL
      - if col2 is NULL, then NULL will be returned

- `COALESCE` does this as well, but behaves slightly differently

  - `COALESCE` allows you to replace NULLs from replacement values themselves
    - e.g. when col1 is NULL, it's replaced with col2; when col2 is NULL, it's replaced with col3, etc
  - `ISNULL` has to be wrapped around another (set of) `ISNULL` function(s) in order to mimic this behaviour

- Both are acceptable, `ISNULL` may be faster in some cases, though this isn't totally clear

  - `ISNULL` is also less flexible for mixed data types

# ISNULL (and/or COALESCE, flavour dependent)

(`ISNULL` & `COALESCE` live coding)

Lingering Questions?