

# Essential SQL Techniques

```
$ echo "Data Sciences Institute"
```

# Essential Techniques

→ [Aggregation Functions](#)

Subqueries

Temporary Tables & CTEs (Common Table Expressions)

Datetime Functions

# Aggregation Functions Overview

- **GROUP BY:** Organizes your results based on selected columns.
- **COUNT:** Returns the number of rows that match a specified condition.
- **SUM & AVG:** Calculate total and average values of a specified column.
- **MIN & MAX:** Find the smallest and largest values.
- **Arithmetic Operations:** Perform calculations on column values.
- **HAVING:** Apply filters after aggregations.

# GROUP BY: Foundation of Aggregation

- Aggregations generally require a *group*
- GROUP BY is mandatory any time a column aside from the one being aggregated is present

# GROUP BY: Foundation of Aggregation

For example, a query wanting to know the number of days in each month:

```
SELECT  
COUNT(days)  
,months  
  
FROM calendar  
GROUP BY months
```

- GROUP BY comes after a WHERE clause
- Remember: with all aggregation functions, values will be calculated *per group*

# COUNT: Quantifying Rows

- COUNT performs counts of any given column or set of columns
- COUNT(\*) provides a count of rows in a given table
  - no GROUP BY is required here
- Multiple counts are treated as separate columns
  - e.g. counting the number of days and months in a year

```
SELECT  
COUNT(days)  
, COUNT(DISTINCT months)  
, years  
  
FROM calendar  
GROUP BY years
```

# COUNT: Quantifying Rows

- When `COUNT` is combined with `DISTINCT`, only unique values are counted
  - `SELECT COUNT(DISTINCT product_id) ...` might produce a different value than `SELECT COUNT(product_id) ...` depending on the context of the table

vendor_id	product_id	market_date
1	1	2025-01-01
1	1	2025-01-02
1	2	2025-01-01
1	2	2025-01-02
1	2	2025-01-03
1	3	2025-01-02
1	4	2025-01-02

vendor_id	count_of_distinct_products
1	4

vendor_id	count_of_products
1	7

# COUNT: Quantifying Rows

( COUNT live coding)

# SUM & AVG: Calculating Totals and Averages

- `SUM` performs the sum total of any numeric column
  - Be wary, SQLite may be more permissive for columns with numbers; it's best practice to coerce (`CAST`) these values into numbers before summing to be certain of their validity
    - e.g. `SUM(CAST(column1 AS INTEGER)) AS column1`
- SUM can accommodate multiple columns using the plus `+` operator
  - e.g. `SUM(column1 + column2)`
- Thinking about `SUM` and `COUNT` combined (i.e. a rolling total)? We'll get to that in the next session!

# SUM & AVG: Calculating Totals and Averages

- `AVG` performs the average of any numeric column
- Like `SUM`, it can accommodate multiple columns
  - we can also use other mathematical operations for `SUM` and `AVG`, like `-`, `*`, `/`, `%` (i.e. modulo, not percent), etc

# SUM & AVG: Calculating Totals and Averages

Watch out! Don't average an average column when using `GROUP BY`

Imagine, `market_avg_temp` stored in the `market_date_info` table:

market_day	market_avg_temp
Saturday	36
Sunday	33
Wednesday	25
Saturday	28
Sunday	31
Wednesday	31
Saturday	29

# SUM & AVG: Calculating Totals and Averages

If we `GROUP BY market_day`, we can produce an average for each day of the week:

market_day	dow_market_avg_temp
Saturday	31
Sunday	32
Wednesday	28

# SUM & AVG: Calculating Totals and Averages

Avoid averaging `dow_market_avg_temp` to get an `overall_market_avg_temp` :

actual_avg	avg_of_avgs
30.42857	30.33333

# SUM & AVG: Calculating Totals and Averages

( SUM & AVG live coding)

# MIN & MAX: Finding Extremes

- `MIN` takes the single minimum value of a given column; `MAX` takes the maximum
- Be wary of combining `MIN` & `MAX` with other aggregating functions like `SUM` or `AVG`
- **What do we think happens when `MIN` is performed on a string? Error? Something else? What about `MAX` ?**

# MIN & MAX: Finding Extremes

( MIN & MAX live coding)

# Arithmetic in SQL

- SQL can perform many basic (and some complex) calculations
  - addition, subtraction, multiplication, division, power, etc
  - geometric/trigonometric functions sin, cos, tan, degrees, radians, etc
- These calculations can also be combined inside aggregation functions
  - e.g. multiplication inside a SUM `... SUM(quantity * cost)` would create a column like `total_spent` per group
- SQL is similar to other programming languages in its ability to handle floating point values
- **Because columns are type specific, how would we perform integer division on two numbers?**

# Arithmetic

(Arithmetic live coding)

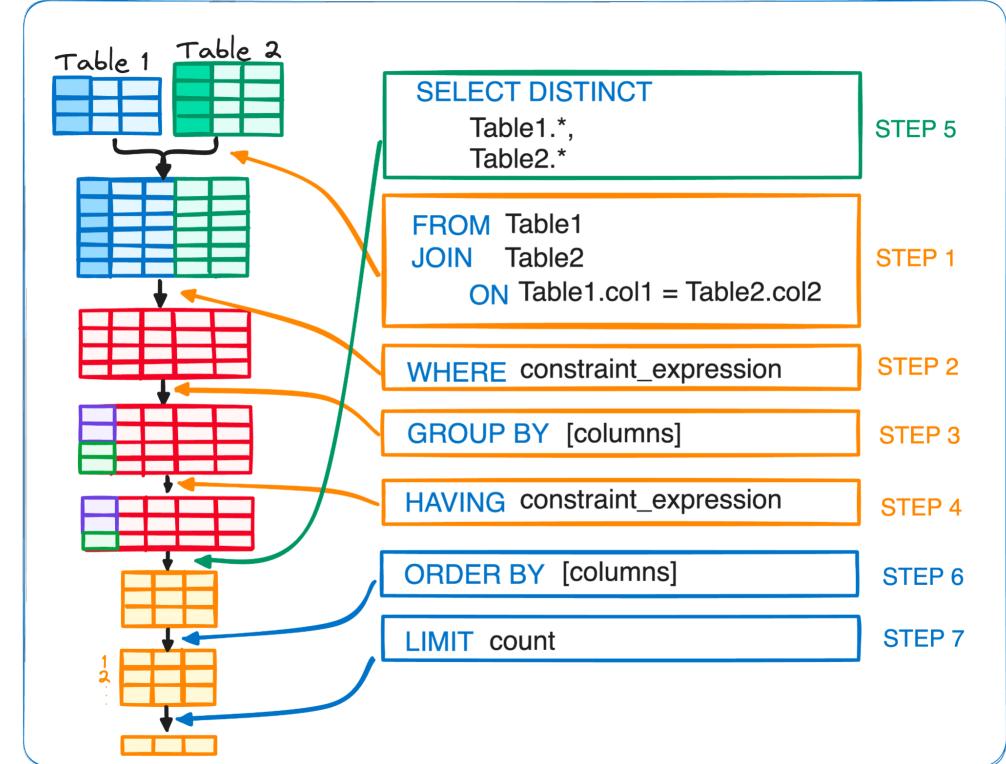
# HAVING: Filtering Aggregated Results

- WHERE clauses filter rows *before* an aggregation occurs
  - ...so HAVING clauses allow us to filter rows *after* an aggregation is calculated
- HAVING clauses come after GROUP BY , but before ORDER BY
- HAVING clauses only filter aggregated calculations
  - you can have both WHERE and HAVING
  - they are not interchangeable

# HAVING: Filtering Aggregated Results

( HAVING live coding)

Image: Joseph Ferrer, KDnuggets



# What questions do you have about Aggregations?

# Subqueries: Queries Within Queries

- SQL allows us to query the results of another query
  - We call this a subquery
- In a subquery, all columns need to be uniquely named
- Subqueries can usually be run for testing purposes by highlighting them, IDE dependent

# Subqueries: Queries Within Queries

- Subqueries can be used in both `JOIN` and `WHERE` clauses
  - In the case of `JOIN` (or `FROM`):
    - you want the subquery to add columns to your output
    - you are using a subquery because you are joining/selecting complex criteria that require manipulation
      - it is often the case that you are joining two or more tables within a subquery to another table

```
SELECT *  
FROM  
(  
    SELECT  
        vendor_id  
        ,market_date  
        ,product_id  
        ,original_price  
        ,ROW_NUMBER() OVER (PARTITION BY vendor_id  
                            ORDER BY original_price DESC) AS price_rank  
    FROM vendor_inventory  
) x  
WHERE x.price_rank = 1
```

*outer query:  
everything from the subquery*

*subquery:  
ordering each vendor's  
items by price*

*outer query:  
filtered by price\_rank*

# Subqueries: Queries Within Queries

- 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?

```
SELECT market_date  
,customer_id  
,vendor_id  
,quantity * cost_to_customer_per_qty price  
FROM customer_purchases  
WHERE market_date IN  
(  
    SELECT market_date  
    FROM market_date_info  
    WHERE market_rain_flag = 1  
)
```

outer query:  
each sale that's occurred

subquery:  
all days it has rained

# Subqueries

(Subqueries live coding)

**SELECT Questions? FROM (Questions?)**

# 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

# Temporary ("temp") Tables

- 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?  Think, Pair, Share

# Temporary Tables

(Temporary Table live coding)

# 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

# Common Table Expressions (CTEs)

- 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)

# What questions do you have?

# Datetime Functions: Managing Dates and Times

- **Formats:** Understand and convert between different date formats.
- **'NOW':** Get the current date and time.
- **STRFTIME:** Format dates and times based on specific patterns.
- **Adding and Subtracting Dates:** Calculate date intervals and future/past dates.
- **Difference between Dates:** Find the interval between two 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

# '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()`, `DATETIME()`, `TIME()` (without any arguments) or  
`DATE('now')`

# 'NOW'

( 'NOW' live coding)

# 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 2025-10

# STRFTIME

- `STRFTIME` also allows modification to date dynamically
  - e.g. `SELECT STRFTIME( '%Y-%m-%d', '2025-11-11', '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)

# 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('2025-11-11', 'start of month')`
- Both of these methods allow you to chain modifiers
  - e.g. `SELECT DATE('2025-11-11', '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

# 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 next total solar eclipse  `SELECT STRFTIME( '%m', '2026-08-12' ) - STRFTIME( '%m', 'NOW' )`

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

- We can 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

# Datetime Functions

(Adding Dates and Difference between Dates live coding)

**What questions do you have about anything from today?**