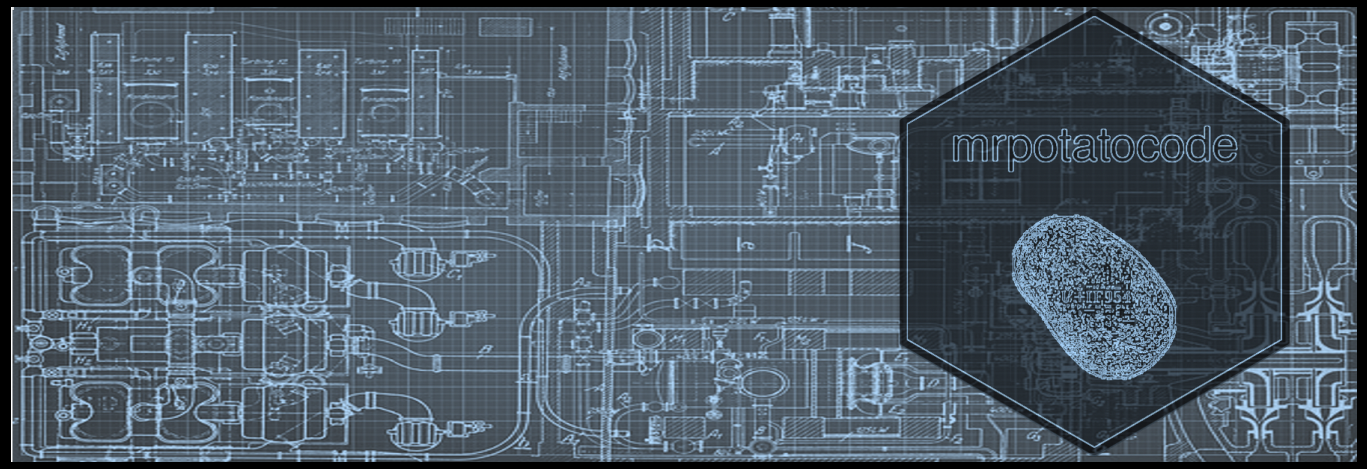


Introduction to Data Access and Storage

Thomas Rosenthal - DSI @ UofT

Module 02

Building Queries:

Two More Commands

JOIN

Aggregation Functions

Building Queries:

Two More Commands

JOIN

Aggregation Functions

Two More Commands

CASE

DISTINCT

Two More Commands

CASE

DISTINCT

CASE

- `CASE` statements allow us to introduce conditional logic into our `SELECT` statements
- They are generally similar to `if` or `if else` statements in python, R, and other languages
    - When a condition is introduced, we check whether it evaluates to TRUE
        - If it is true, we proceed with a desired command, calculation, value, etc
        - If it is not true, we move to the next condition
            - If it is true, we proceed with another desired command, calculation, value, etc
                - ...all the way until we run out of conditions
    - For all FALSE conditions, we can use an `ELSE` statement if we want to
- The results of a `CASE` statement will be a new column
- Best practice is to name the new column using `AS new_column_name`

```
CASE
  WHEN [something is true]
    THEN [value or calculation]
  WHEN [something else is true]
    THEN [value or calcuation]
  ELSE [value or calcuation]
END
```

CASE

(CASE live coding)

Two More Commands

CASE

DISTINCT

DISTINCT

- Not all query results will result in unique rows (i.e. duplicates are present)
  - Can we think of why this is?
- `DISTINCT` has two possible spots within a query:
  - One comes immediately after `SELECT`, before column names are specified
    - e.g. `SELECT DISTINCT songs, albums, artists...`
    - This `DISTINCT` will govern the entire query
  - The other comes within aggregation
    - e.g. `COUNT(DISTINCT products)`
    - This `DISTINCT` will only affect this specific aggregation

DISTINCT

(`DISTINCT` live coding)

Building Queries:

Two More Commands

JOIN

Aggregation Functions

JOIN

INNER JOIN

LEFT (OUTER) JOIN

RIGHT (OUTER) JOIN

FULL OUTER JOIN

Multiple Table Joins

JOIN

- Joins are used to combine data stored in different tables into a single table
- Joins are the Cartesian product of two tables with conditional selection of specific rows
    - A Cartesian product combines all possible row values with another
    - An easy example is a deck of cards:

      combining four suits: {♠, ♥, ♦, ♣}

      with thirteen ranks: {A, K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2}

      produces 52 cards (4 * 13)
- Joins require relationships (one exception, we'll get to later) between tables
- Different joins create different results
    - Join names specify which conditional selection is desired
- There are three join types in SQL but different joining criteria can further limit results
- The most permitting join is a `FULL OUTER JOIN` and the least permitting is an `INNER JOIN`
    - Let's explore what this means by looking at each of them

JOIN

JOIN Syntax

Syntax for a join is as follows:

```
SELECT [columns]
FROM [left table]
JOIN [right table]
ON [left table.matching column] = [right table.matching column]
```

A couple of notes:

- You will need to specify which join type is desired:
  - e.g. `INNER JOIN`
- Matching columns do not need to have the same name, just the same value
  - e.g. `ON table1.LetterGrade = table2.Alphabet` will work because A=A, B=B, C=C, etc
- You can specify more than one column to be joined
  - e.g. `ON table1.FirstName = table2.FirstName AND table1.LastName = table2.LastName`

JOIN

INNER JOIN

LEFT (OUTER) JOIN

RIGHT (OUTER) JOIN

FULL OUTER JOIN

Multiple Table Joins

INNER JOIN

- `INNER JOIN` filters both tables to rows present in both tables
- `INNER JOIN` does not produce `NULL` values
- `INNER JOIN` is the "default" join
    - i.e. queries do not need to specify "INNER", though it's good practice to write INNER

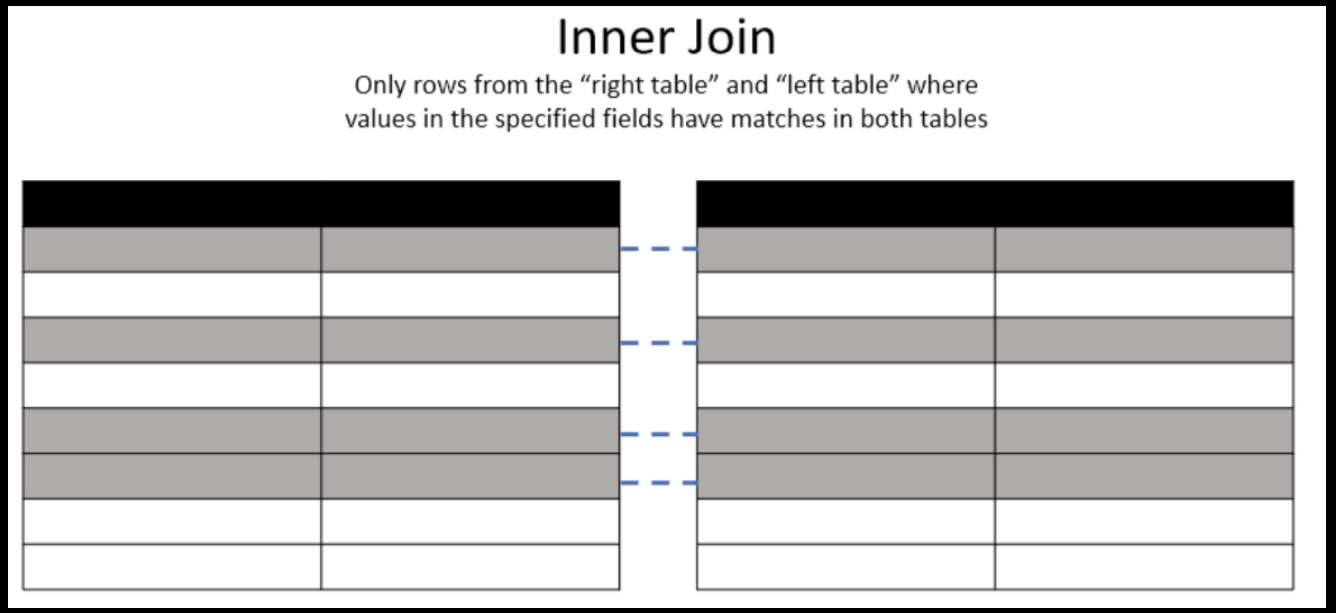


Image: Teate, Chapter 5

INNER JOIN

A quick note on table aliasing:

- It is very common practice to alias table names
  - It makes join criteria much more concise
  - It simplifies `SELECT` statements when column names are the same
  - This is a common error: *"ambiguous column name"*
    - SQL requires you to specify *which* table you are returning the result from
- Generally, tables are aliased with the first letter (or first few letters) of the table so they can be easily referenced
  - `product AS p`
  - `product_category AS pc`

INNER JOIN

(`INNER JOIN` live coding)

JOIN

INNER JOIN

LEFT (OUTER) JOIN

RIGHT (OUTER) JOIN

FULL (OUTER) JOIN

Multiple Table Joins

LEFT (OUTER) JOIN

- `LEFT JOIN` filters the "right" table to rows present in the "left" table
- `LEFT JOIN` will most often produce `NULL` values
- The "OUTER" in `LEFT OUTER JOIN` is optional
  - Generally, OUTER seems to be excluded but both are correct
- LEFT *is not* optional, there is no "OUTER JOIN"

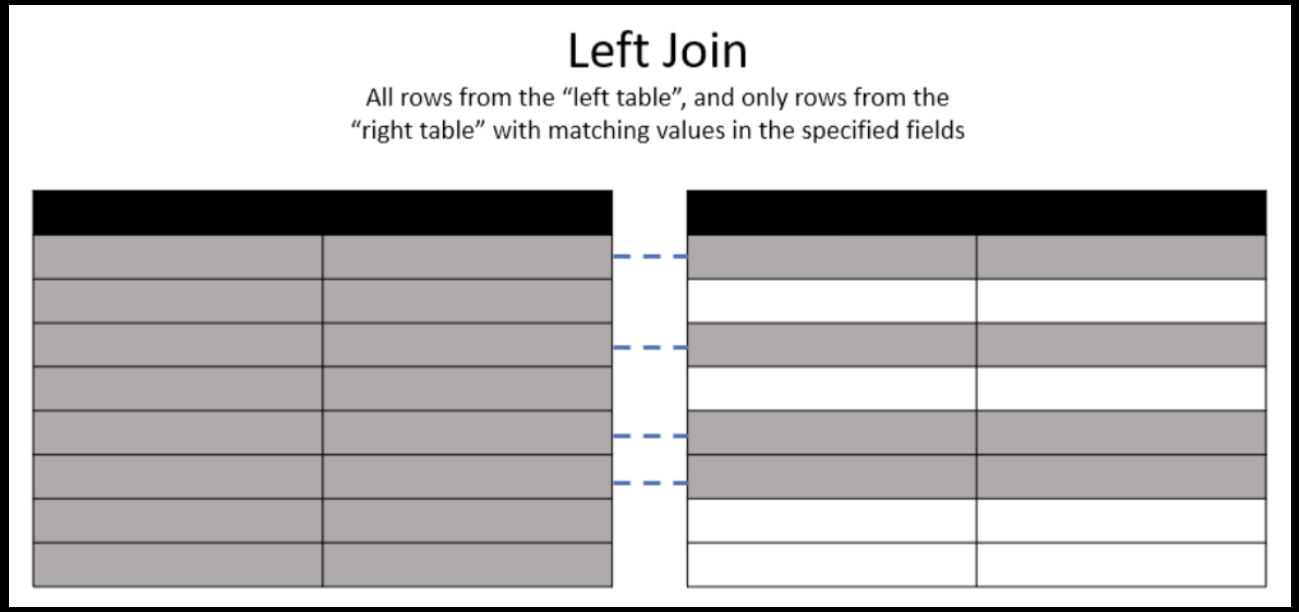


Image: Teate, Chapter 5

LEFT (OUTER) JOIN

(`LEFT JOIN` live coding)

JOIN

INNER JOIN

LEFT (OUTER) JOIN

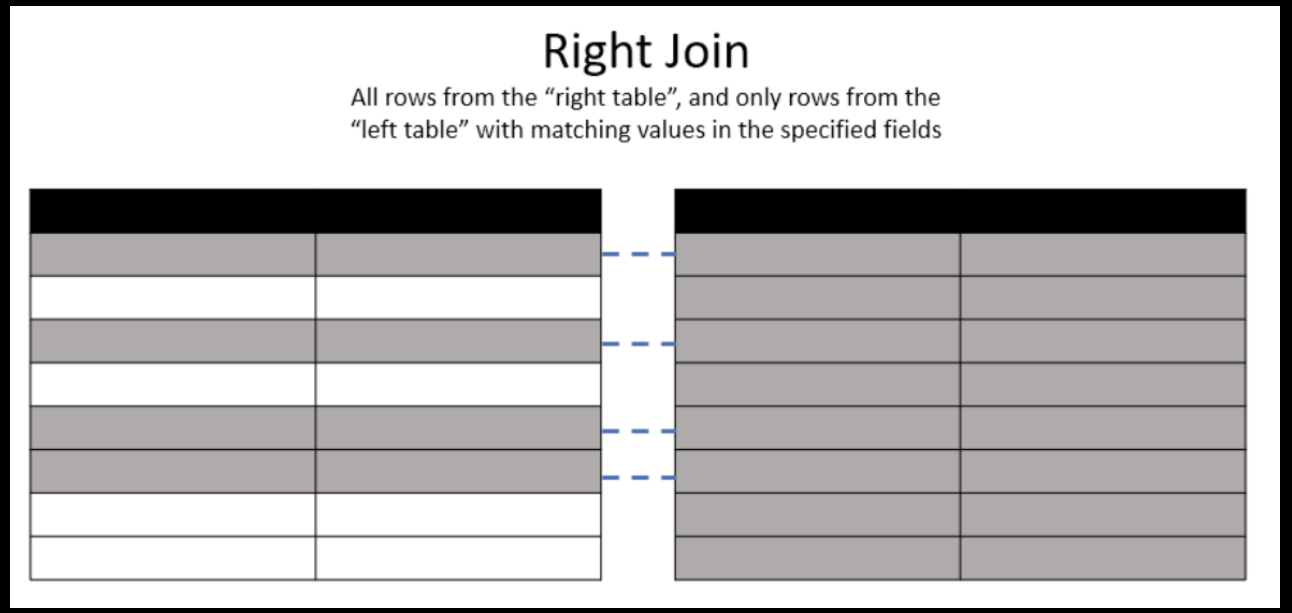
RIGHT (OUTER) JOIN

FULL (OUTER) JOIN

Multiple Table Joins

RIGHT (OUTER) JOIN

- `RIGHT JOIN` filters the "left" table to rows present in the "right" table
- `RIGHT JOIN` will most often produce `NULL` values
- The "OUTER" in `RIGHT OUTER JOIN` is optional
  - Generally, OUTER seems to be excluded but both are correct
- `RIGHT JOIN` is somewhat frowned upon, but sometimes they make sense
  - Often your query can be reorganized to use a `LEFT JOIN` instead
  - SQLite does not currently support `RIGHT JOIN`



JOIN

INNER JOIN

LEFT (OUTER) JOIN

RIGHT (OUTER) JOIN

FULL (OUTER) JOIN

Multiple Table Joins

FULL (OUTER) JOIN

- `FULL OUTER JOIN` does not filter either "left" or "right" table
- Expect `NULL` values to be produced from a `FULL OUTER JOIN`
- My experience has been to write `FULL OUTER JOIN` rather than `FULL JOIN` but this is personal preference
- Annoyingly, SQLite does not support `FULL OUTER JOIN` (*it really should*), but there is a work around to produce the results (we'll do it in Module 4)

Filtering a FULL (OUTER) JOIN

- All OUTER JOIN syntax can be filtered to exclude the *matching* criteria
  - Often called an ANTI JOIN, i.e. what's *not* in the other table

```
SELECT *
FROM table_1
{LEFT | RIGHT | FULL} OUTER JOIN table_2
ON table_1.key = table_2.key
WHERE {table_1.key IS NULL | table_2.key IS NULL |
       table_1.key IS NULL OR table_2.key IS NULL}
```

JOIN

INNER JOIN

LEFT (OUTER) JOIN

RIGHT (OUTER) JOIN

FULL OUTER JOIN

Multiple Table Joins

Multiple Table Joins

- More than one table can be joined at a time
- The order and type of joins will have significant effect on the final table
  - Pause to imagine what these scenarios might look like based on your knowledge of different `JOIN` types

Multiple Table Joins

For example, let's combine these three tables:

FirstName	Number
A	1
Thomas	2

(*Table 1*)

LastName	Number
Mahfouz	1
Rosenthal	2

(*Table 2*)

FirstName	LastName	Symbol
A	Smith	!
Edward	Rosenthal	?

If the desired outcome table is:

FirstName	LastName	Number	Symbol
A	Mahfouz	1	!
Thomas	Rosenthal	2	?

What are the correct `JOIN` criteria?

Multiple Table Joins

(Multiple Table Joins live coding)

Building Queries:

Two More Commands

JOIN

Aggregation Functions

Aggregation Functions

GROUP BY

COUNT

SUM & AVG

MIN & MAX

Arithmetic

Aggregation Functions

GROUP BY

COUNT

SUM & AVG

MIN & MAX

Arithmetic

GROUP BY

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

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*

Aggregation Functions

GROUP BY

COUNT

SUM & AVG

MIN & MAX

Arithmetic

COUNT

- `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(months) ,years

  FROM calendar GROUP by years `
- When `COUNT` is combined with `DISTINCT`, only unique values are counted

COUNT

(`COUNT` live coding)

Aggregation Functions

GROUP BY

COUNT

SUM & AVG

MIN & MAX

Arithmetic

SUM & AVG

SUM

- `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. `CAST(SUM(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!

AVG

- `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

(`SUM` & `AVG` live coding)

Aggregation Functions

GROUP BY

COUNT

SUM & AVG

MIN & MAX

Arithmetic

MIN & MAX

- `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

(`MIN` & `MAX` live coding)

Aggregation Functions

GROUP BY

COUNT

SUM & AVG

MIN & MAX

Arithmetic

Arithmetic

- 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)

Aggregation Functions

GROUP BY

COUNT

SUM & AVG

MIN & MAX

Arithmetic

HAVING

- `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

(`HAVING` live coding)

Lingering Questions?