

# Introduction to Data Access and Storage

Thomas Rosenthal - DSI @ UofT

Module 02

Building Queries:  
Fundamental Three Commands  
Two More Commands  
Putting Things Together with JOIN

# Building Queries:

## Fundamental Three Commands

## Two More Commands

## Putting Things Together with JOIN

# Fundamental Three Commands

SELECT

FROM

WHERE

# Fundamental Three Commands

- Our first three commands (**SELECT**, **FROM**, **WHERE**) are essential to nearly every SQL query
- The template for our initial SQL statement is as such:

**SELECT** : *the columns we want to retrieve*

**FROM** : *the table we are querying*

**WHERE** : *filters/conditions (optional)*

**ORDER BY** : *column sorting: ascending ASC or descending DESC (optional)*

**LIMIT** : *how many rows we want to return (optional)*

# Fundamental Three Commands

- Always specified in this order:
  - `SELECT` will come first
  - `FROM` will come after `SELECT`
    - when we are querying more than one table at a time, each will come after `FROM` but before `WHERE` (more on this later)
  - `WHERE` will come after `FROM`
  - `ORDER BY` will come after `WHERE` clauses
- We'll sometimes use the `LIMIT` clause to look at data
  - This comes at the very end of a query
  - `LIMIT` shouldn't be used for analytics unless you have a specific reason
  - `ORDER BY` often impacts the usefulness of `LIMIT`
- Remember:
  - In SQL, we use two dashes `--` to comment out lines, rather than `#`

# Fundamental Three Commands

SELECT

FROM

WHERE

# SELECT

- At its simplest `SELECT` specifies column names we are retrieving
  - commas come between each column name
    - `SELECT student, course, grade ...`
  - column names with a space need to be enclosed in square brackets
    - `SELECT [poorly named column], better_column_name, AnotherColumnName`
- Within `SELECT` statements we can perform manipulations on columns
  - e.g. rename a column
    - `SELECT [poorly named column] AS better_col`
  - combine two text columns
  - perform math on a numeric column
  - ...and many more things

# SELECT

- We can use `SELECT` to perform math without a `FROM` statement
  - `SELECT 1 + 1`
  - `SELECT 10*5, cos(2), pi()`
- And we can use `SELECT` to specify constant values
  - `SELECT 2023 AS this_year, 'January' AS this_month`
- When selecting columns, they need to exist in the table!

# Fundamental Three Commands

SELECT

FROM

WHERE

# FROM

- `FROM` statements indicate which table the data is from and where the table is located
  - in more complicated RDBMs, you will often have multiple databases on the same server and multiple schema within those databases
  - a fully qualified location of a table would thus be `database.schema.table`
- `SELECT * FROM table_name` indicates *everything* in the table
- Best practice suggests that we should explicitly call each column, even if we want all of them
  - **Why do we think this is the case?**

# FROM

(SELECT & FROM live coding)

# Fundamental Three Commands

SELECT

FROM

WHERE

# WHERE

- WHERE clauses are conditions that the query will follow
- When we want to have multiple conditions, we use a single WHERE and then additional logical operations

```
SELECT *
FROM students
WHERE first_name = 'Thomas'
AND last_name = 'Rosenthal'
```

- Notice we put string values in single quotes
  - SQLite also allows double quotes, with a few minor caveats
- WHERE clauses always return rows evaluating to TRUE
  - Follows Boolean rules if more than one condition is present
- My favourite WHERE statement is WHERE 1=1
  - Any guesses as to why?

# WHERE

## Logical Operators

- AND
- OR
- NOT
- NOT IN
- equals: =
- does not equal: <> !=
  - (flavour dependent)
- greater than (equal to): > >=
- less than (equal to): < <=
- BETWEEN
- EXISTS
  - table specific
- IS
  - NULL specific

# WHERE

## NULL

- `NULL` is not a value (it's the absence of a value)
  - to check null values, we use `IS NULL` or `IS NOT NULL`
  - `= NULL` will not work

## LIKE

- `LIKE` allows for string wildcards
- `%` specifies the wildcard placement
  - `country_name LIKE 'and%`'
    - Andorra
  - `country_name LIKE '%and'`
    - Finland, Iceland ...more
  - `country_name LIKE '%and%`'
    - all of the above, plus Antigua and Barbuda, Netherlands, Rwanda ...more!
  - `country_name LIKE '%an%d%'`
    - Canada ...surely more!

# WHERE

(WHERE live coding)



# Building Queries:

## Fundamental Three Commands

### Two More Commands

## Putting Things Together with JOIN

# 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 queries 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 (we'll get to this later)
    - e.g. `COUNT(DISTINCT products)`
    - This **DISTINCT** will only affect this specific aggregation

# DISTINCT

(DISTINCT live coding)



Building Queries:  
Fundamental Three Commands  
Two More Commands  
Putting Things Together with JOIN

# Putting Things Together with JOIN

INNER JOIN

LEFT (OUTER) JOIN

RIGHT (OUTER) JOIN

FULL OUTER JOIN

Multiple Table Joins

# Putting Things Together with 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 ( $4 * 13$ )

# Putting Things Together with JOIN

- 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

# Putting Things Together with 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`

# Putting Things Together with 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

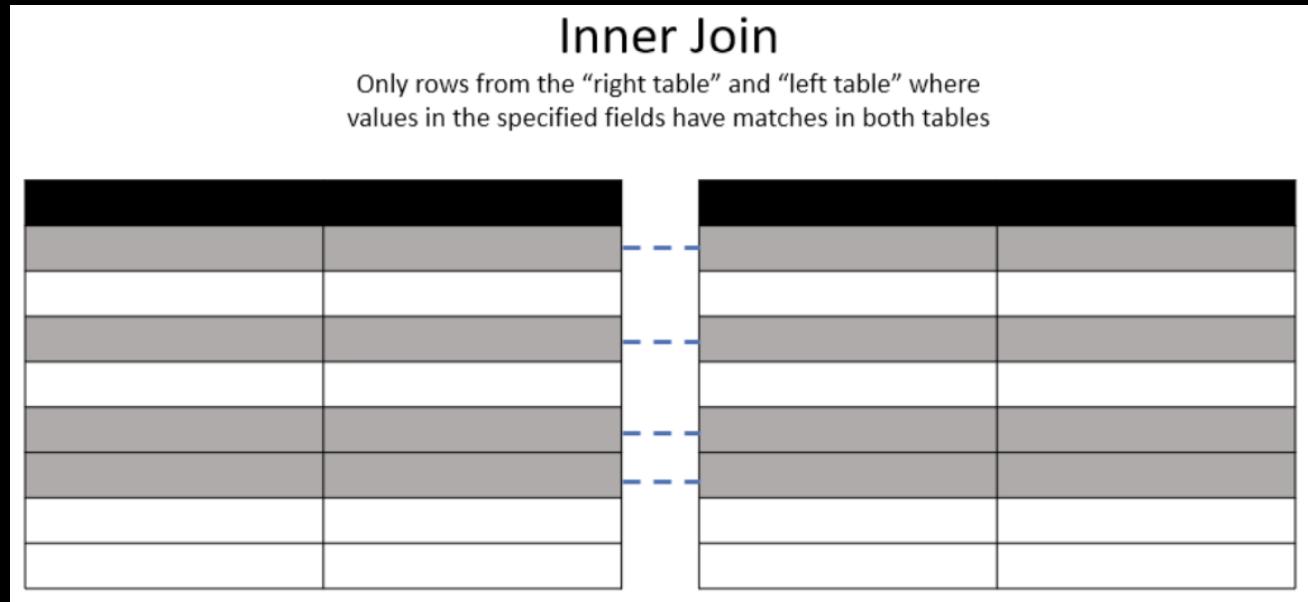


Image: Teate, Chapter 5

# 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)

# Putting Things Together with 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"

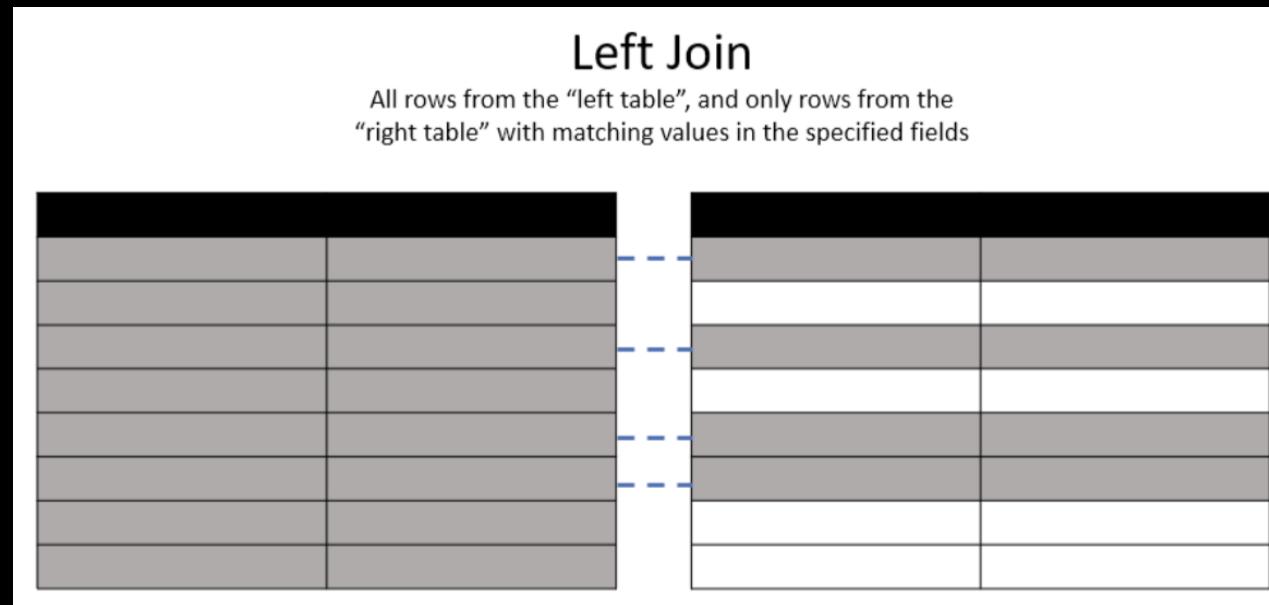


Image: Teate, Chapter 5

# LEFT (OUTER) JOIN

(LEFT JOIN live coding)

# Putting Things Together with JOIN

INNER JOIN

LEFT (OUTER) JOIN

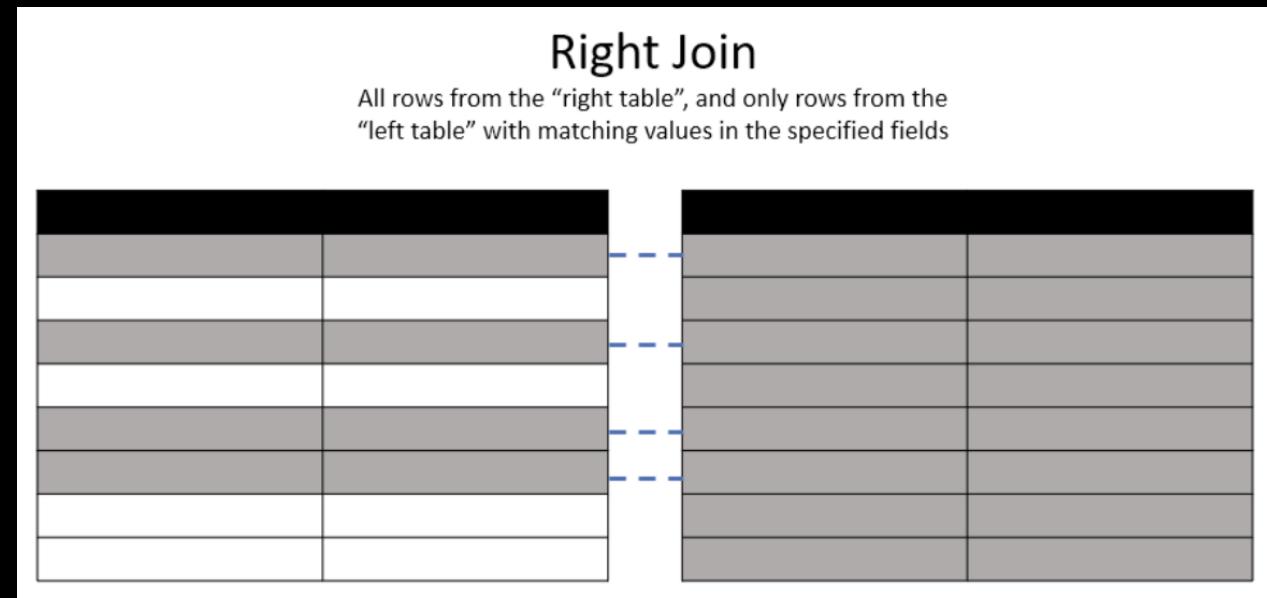
RIGHT (OUTER) JOIN

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`



# Putting Things Together with 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 workaround to produce the results (we'll do it in a later module)

## 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}
```

# Putting Things Together with 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)

