SQL Challenge Edition – Tutorial Cheat Sheet

1. SELECT — Retrieving Data

What it does: Tells SQL what columns you want to see and from which table.

Why it matters: Every SQL query begins with SELECT. You're literally selecting what you want to view from a table.

Example:

```
SELECT Name, Genre, Price
FROM Games;
```

2. WHERE — Filtering Rows

What it does: Limits which rows are returned based on a condition.

Why it matters: Lets you focus on specific data instead of everything in the table.

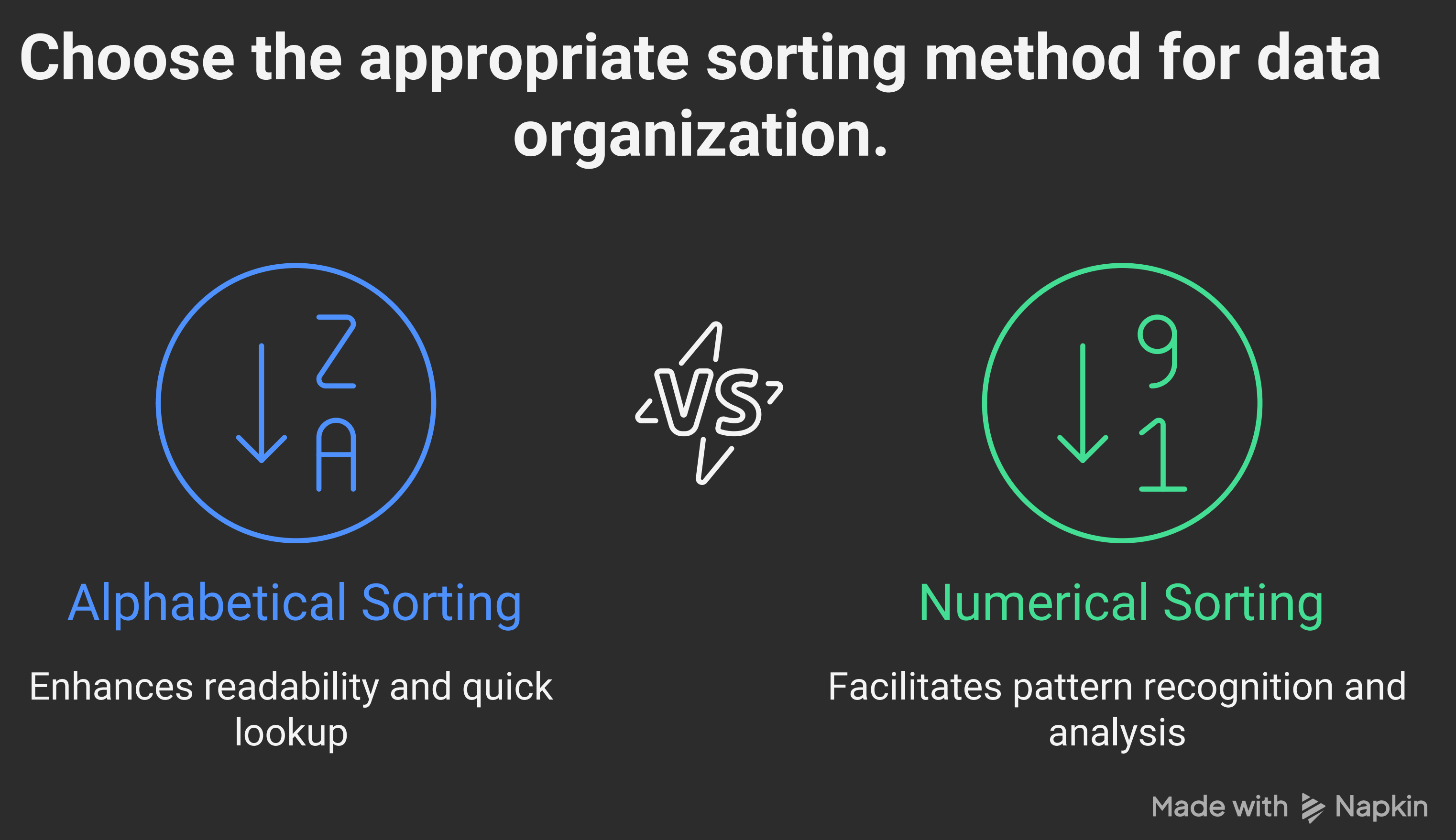
Example:

```
SELECT *
FROM Games
WHERE Price < 30;

SELECT Name, Genre
FROM Games
WHERE Rating >= 9;
```

3. ORDER BY — Sorting Data

What it does: Sorts your results alphabetically or numerically.


Choose the appropriate sorting method for data organization.
Z
A
VS
9
1
Alphabetical Sorting
Enhances readability and quick lookup
Numerical Sorting
Facilitates pattern recognition and analysis
Made with Napkin

Why it matters: It helps you see patterns — like top prices, best ratings, or newest dates.

Example:

```
SELECT Name, Price
FROM Games
ORDER BY Price DESC;

-- or alphabetically ORDER BY Name ASC;
```

4. DISTINCT — Removing Duplicates

What it does: Shows unique values only (no repeats).

Why it matters: Useful for finding categories, cities, or statuses.

Example:

```
SELECT DISTINCT Genre
FROM Games;
```

AND / OR in SQL — Combining Conditions

What it does:

> AND and OR let you combine multiple conditions inside a WHERE clause.Use AND when **all** conditions must be true.Use OR when **any** condition can be true.

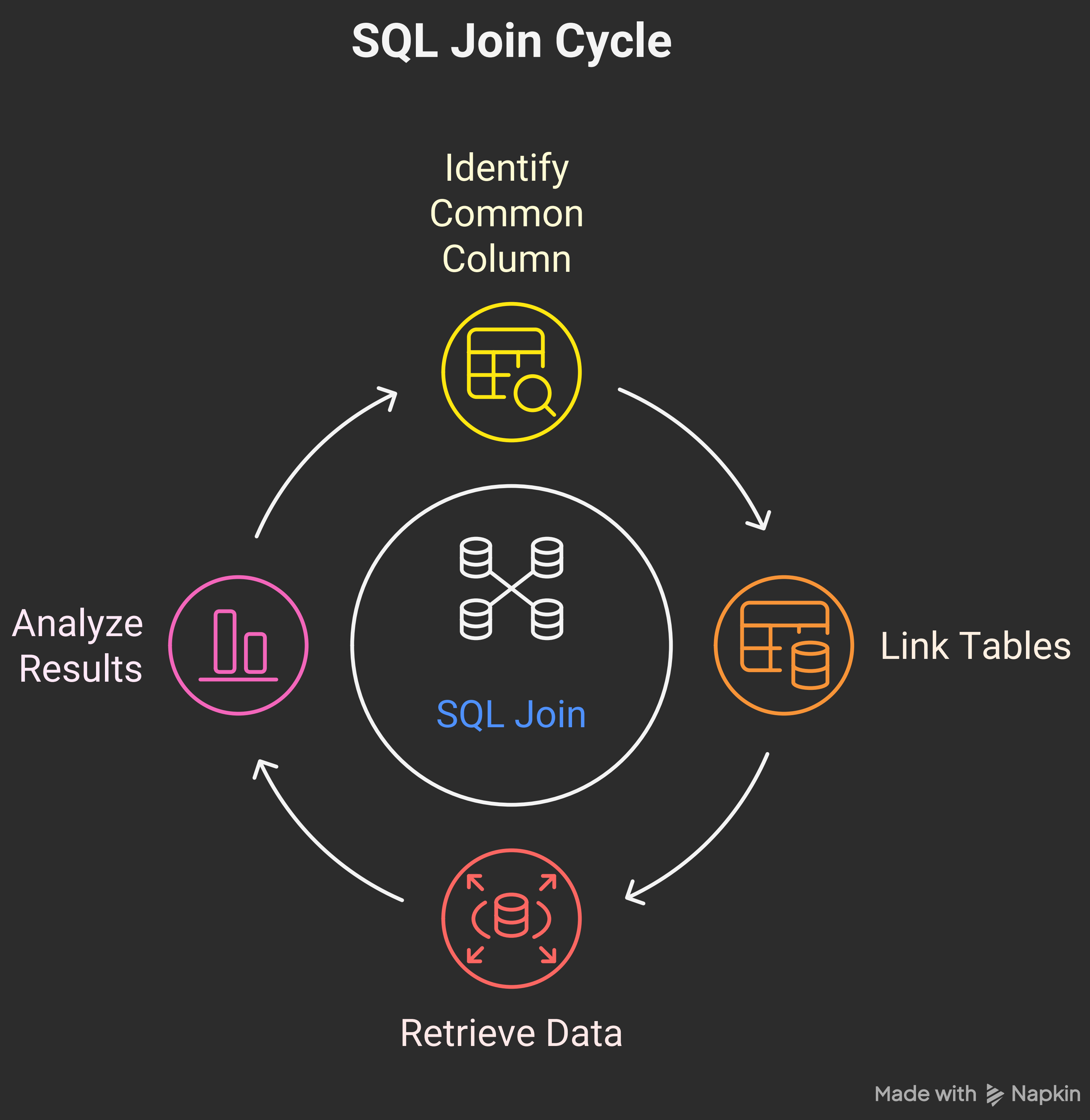
Example 1 — Using AND

```
SELECT Title, Genre, Price
FROM Games
WHERE Genre = 'Action'
  AND Price < 50;
```

Meaning:“Only show games that are *Action* **and** cost less than £50.”

5. JOIN — Combining Tables

What it does: Connects rows from different tables based on a shared column (usually a Primary Key ↔ Foreign Key).



Why it matters: Real data is split across tables — joins let you combine it.

Example:

```
SELECT g.Name, s.SaleDate, s.Quantity
FROM Games g
JOIN Sales s ON g.GameID = s.GameID;
```

INNER JOIN — Only Matching Rows

Shows rows where there's a match in **both tables**.

Example:

```
SELECT g.Name, s.SaleDate, s.Quantity
FROM Games g
INNER JOIN Sales s
  ON g.GameID = s.GameID;
```

Explanation:

- Only returns games that have sales.
- If a game hasn't been sold, it won't appear.

LEFT JOIN — All from Left, Matches from Right

Returns **all rows from the left table (Games)** and the matching ones from **Sales**.If no match, it fills in NULL.

Example:

```
SELECT g.Name, s.SaleDate, s.Quantity
FROM Games g
LEFT JOIN Sales s
  ON g.GameID = s.GameID;
```

Explanation:

- Every game appears, even if it has no sales.
- Missing sales data will show as NULL.

RIGHT JOIN — All from Right, Matches from Left

Returns **all rows from the right table (Sales)** and the matching rows from **Games**.If a sale refers to a missing game, it'll still appear with NULLs for game info.

Example:

```
SELECT s.SaleID, g.Name, s.SaleDate, s.Quantity
FROM Games g
RIGHT JOIN Sales s
  ON g.GameID = s.GameID;
```

Explanation:

- All sales show up, even if their corresponding game record doesn't exist.
- Used less often, but useful for data validation.

FULL OUTER JOIN — Everything Combined

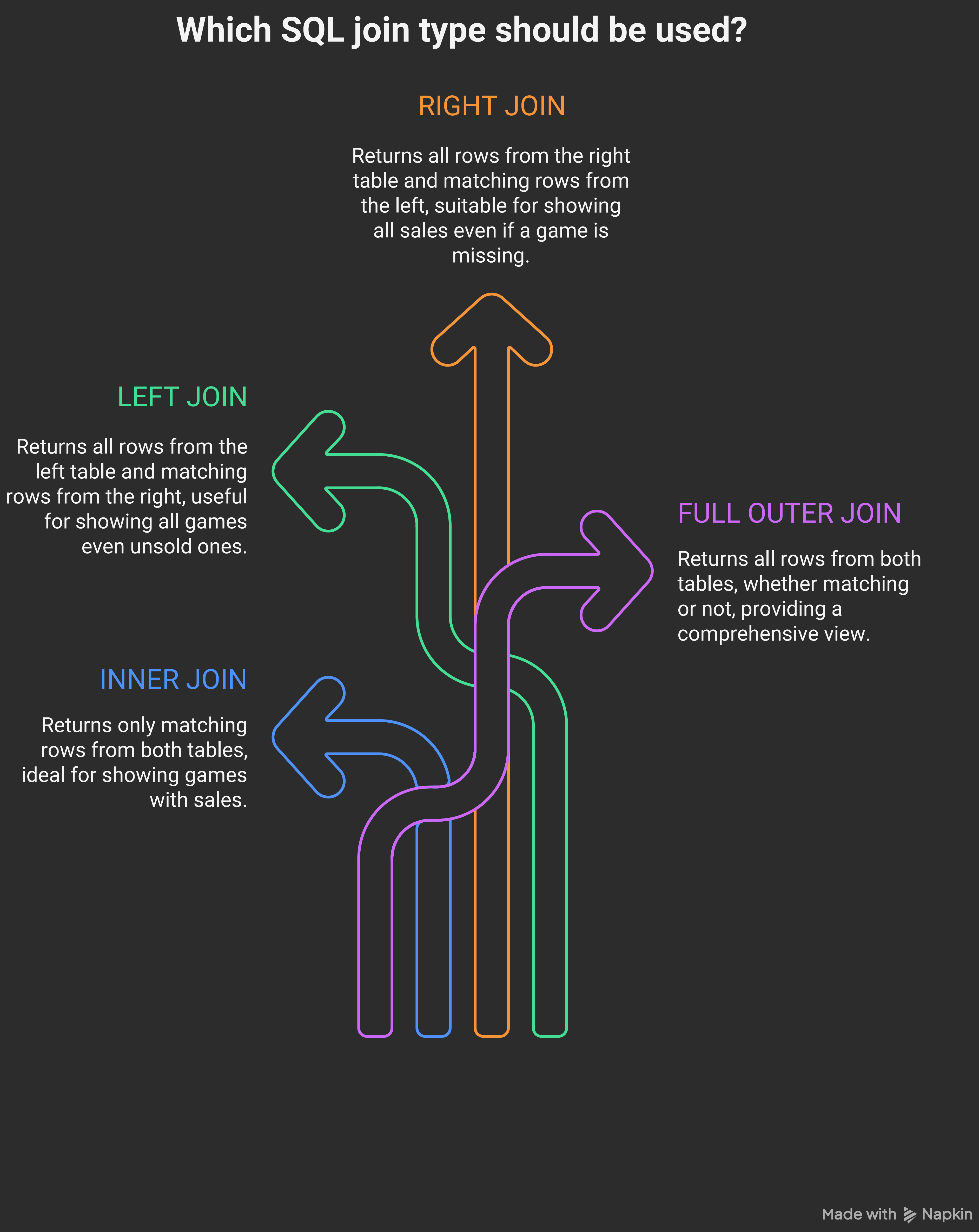
Returns **all rows from both tables**, matched or not.Rows without matches will show NULLs on the side they don't exist.

Example:

```
SELECT g.Name, s.SaleDate, s.Quantity
FROM Games g
FULL OUTER JOIN Sales s
  ON g.GameID = s.GameID;
```

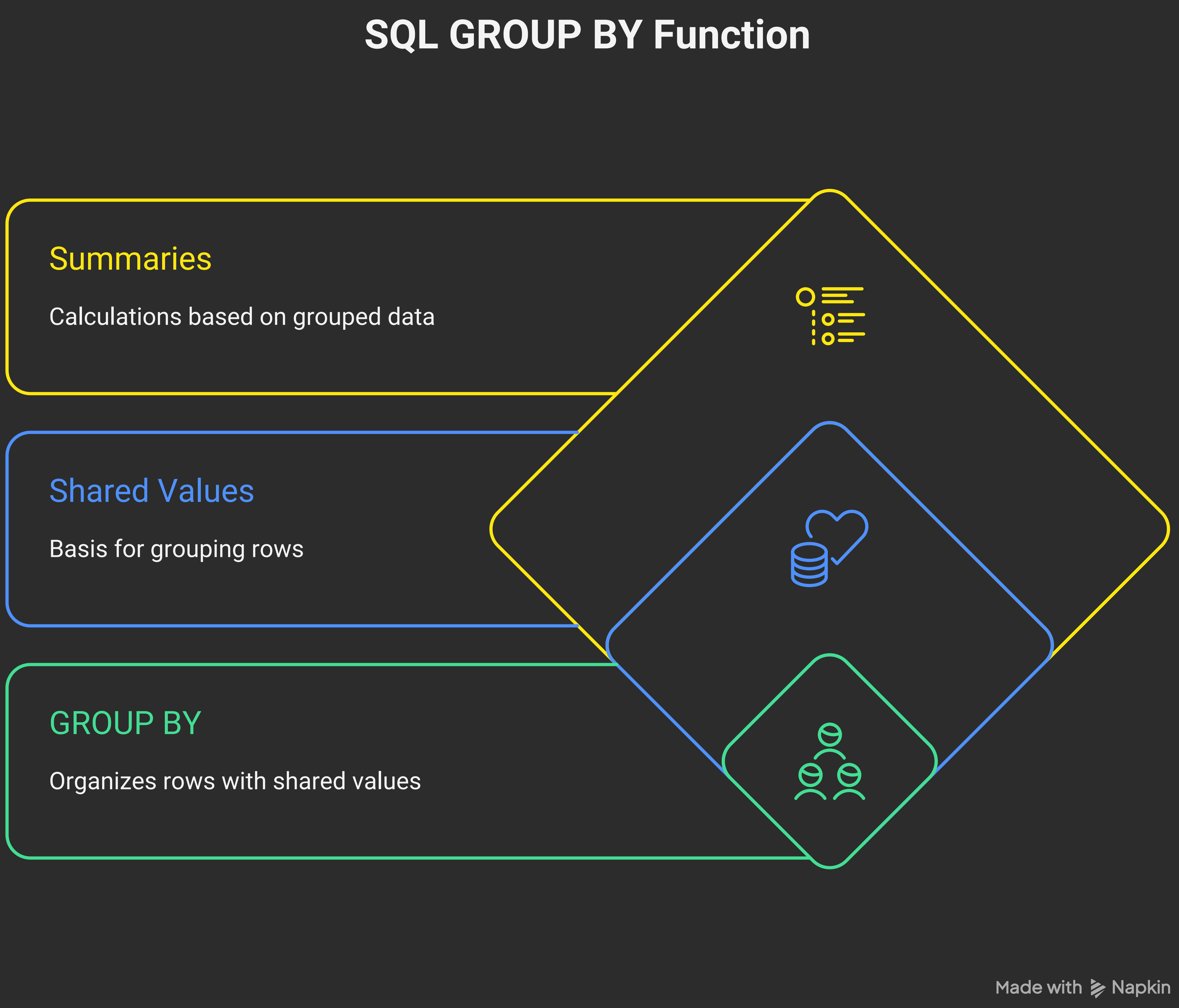
Explanation:

- Includes every game and every sale.
- If a game has no sale → NULL in sales columns.
- If a sale's game is missing → NULL in game columns.


Which SQL join type should be used?
RIGHT JOIN
Returns all rows from the right table and matching rows from the left, suitable for showing all sales even if a game is missing.
LEFT JOIN
Returns all rows from the left table and matching rows from the right, useful for showing all games even unsold ones.
FULL OUTER JOIN
Returns all rows from both tables, whether matching or not, providing a comprehensive view.
INNER JOIN
Returns only matching rows from both tables, ideal for showing games with sales.
Made with Napkin

6. GROUP BY — Summarizing Data

What it does: Groups rows that share a common value so you can calculate summaries.



Why it matters: It turns raw data into insights (e.g. total sales per game or per genre).

Example:

```
SELECT g.Name, SUM(s.Quantity) AS TotalSold
FROM Games g
JOIN Sales s ON g.GameID = s.GameID
GROUP BY g.Name;
```

7. HAVING — Filtering Grouped Results

What it does: Filters the summarized results after GROUP BY.

Why it matters:
WHERE can't filter aggregated results — HAVING can.
Example:

```
SELECT g.Name, SUM(s.Quantity) AS TotalSold
FROM Games g
JOIN Sales s ON g.GameID = s.GameID
GROUP BY g.Name
HAVING SUM(s.Quantity) > 2;
```

8. TOP — Restricting Results

What it does: Shows only a certain number of results.

Why it matters: Perfect for “Top 3 best-selling” or “5 most recent” queries.

Example (SQL Server):

```
SELECT TOP 3 Name, Price
FROM Games
ORDER BY Price DESC;
```

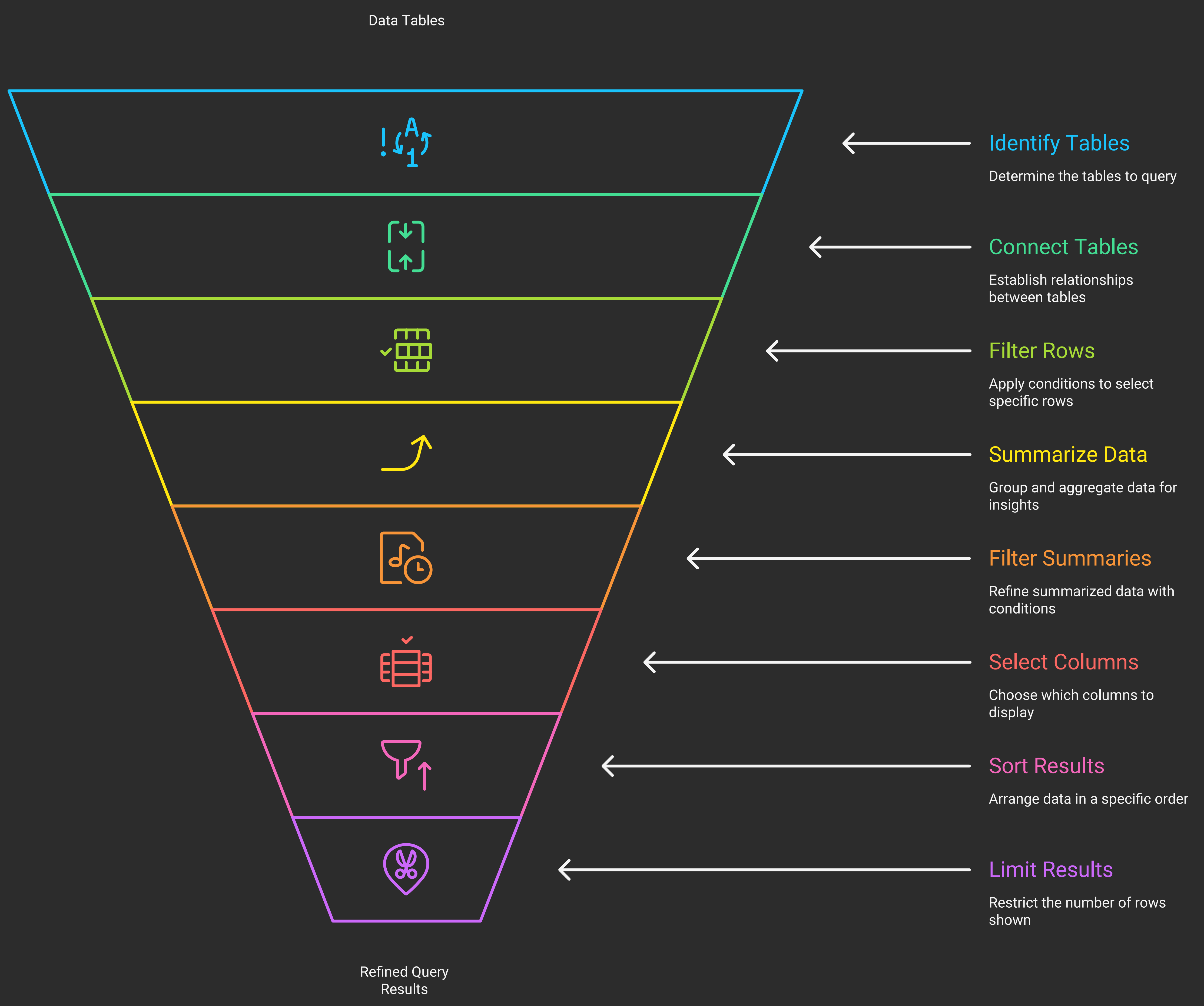
9. Real-World Examples (Mini Challenges)

1. Total revenue per game → SUM(Quantity * Price) + JOIN
2. Games sold more than 2 copies → GROUP BY + HAVING
3. Top 2 best-selling genres → GROUP BY + LIMIT
4. Games never sold → LEFT JOIN + IS NULL

10. SQL Flow Recap

SQL reads like a sentence — start with what you want, then where it comes from, and finally how to shape it.

SQL Query Refinement Process



Data Tables
Identify Tables
Determine the tables to query
Connect Tables
Establish relationships between tables
Filter Rows
Apply conditions to select specific rows
Summarize Data
Group and aggregate data for insights
Filter Summaries
Refine summarized data with conditions
Select Columns
Choose which columns to display
Sort Results
Arrange data in a specific order
Limit Results
Restrict the number of rows shown
Refined Query Results