**INTRODUCTION TO SQL**

**Ch1: Relational databases**

Rows = ‘records’

Columns = ‘fields’

Table names and field names should be lowercase and not include spaces (use ‘\_’). If they do have a space, you need to enclose them in double quotes.

Field names should be singular

Field names should not share a name with the table name

There are different data types for strings, integers and floats, depending on the length.

DB schema: a db design with the tables included, their relationships, their fields and data types

**Ch2: Querying**

**Select fields from a table**

*SELECT name*

*FROM table;*

(It is best practice to end the query with a semi-colon to indicate that the query is complete)

*SELECT field1, field2*

*FROM table;*

*SELECT \**

*FROM table;*

**Select fields from a table and change their name**

*SELECT field1 AS my\_field, field2*

*FROM table;*

**Select unique values from a field**

*SELECT DISTINCT field1,*

*FROM table;*

**Select unique values from a combination of fields**

*SELECT DISTINCT field1, field*

*FROM table;*

**View**: a table that is the result of a saved SQL SELECT statement

When accessed, view automatically update in response to updates in the underlying data.

*CREATE VIEW view\_name AS*

*SELECT field2, field2*

*FROM table;*

Once a view is created, you can query it just as a normal table

*SELECT field2*

*FROM view\_name*

**SQL Flavors:** different SQL versions, dialects of the same language

* **PostgreSQL:** free, open-source, relational database system
  + Limit the number of records returned / return the first N records

*SELECT id, field1*

*FROM table*

*LIMIT 2;*

* **SQL Server:** has free and paid (T-SQL) versions, created by Microsoft
  + Limit the number of records returned / return the first N records

*SELECT id, field1*

*FROM table*

*TOP 2;*

**-- INTERMEDIATE SQL**

**-- Ch1: Selecting data**

**-- Return the number of records with a value in a field**

*SELECT COUNT(field1) AS count\_field1*

*FROM table;*

**Count multiple fields**

*SELECT COUNT(field1) AS count\_field1, COUNT(field2) AS count\_field2*

*FROM table;*

**-- Count the total number of records in a table (including missing values)**

*SELECT COUNT(\*) AS total\_records*

*FROM table;*

**-- Count the number of unique values of a field**

*SELECT COUNT(DISTINCT field2) AS count\_distinct\_field1*

*FROM table;*

**-- Order of execution:** SQL code is NOT processed in the order in which it is written.

-- First, the “FROM” statement, then “WHERE”, then the “SELECT” statement (before that the aliasing statements), then the “LIMIT” statement

**-- Debugging SQL code:** most common errors are misspelling, incorrect capitalization, and incorrect or missing punctuation, specially commas.

**-- SQL style:** formatting (new lines, capitalization, indentation) are NOT required, but there are style standards.

**-- Ch2: Filtering records**

**-- Filtering numbers**

*SELECT title*

*FROM films*

*WHERE release\_year > 2010;*

*SELECT title*

*FROM films*

*WHERE release\_year = 2010;*

*SELECT title*

*FROM films*

*WHERE release\_year <> 2010;*

**-- Filtering strings:** use single quotation marks

*SELECT title*

*FROM films*

*WHERE country = ‘Japan’;*

**-- Multiple criteria**

*SELECT \**

*FROM coats*

*WHERE color = ‘yellow’*

*OR color = ‘black;*

*SELECT \**

*FROM coats*

*WHERE color = ‘yellow’*

*AND length = ‘short’;*

*SELECT \**

*FROM coats*

*WHERE buttons*

*BETWEEN 1 AND 5;*

(between is INCLUSIVE)

*SELECT \**

*FROM coats*

*WHERE (color = ‘yellow’ OR color=’black’)*

*AND (length = ‘short’ OR length = ‘medium’);*

*Multiple OR conditions*

*SELECT \**

*FROM coats*

*WHERE color IN (‘red‘, ’black’, ‘blue’);*

**-- More string filtering techniques**

* **LIKE:** search for a pattern in a field
  + Wild card “%”: match zero, one or many characters

*SELECT name*

*FROM people*

*WHERE name LIKE ‘Jua%’*

Matches names like: Juan, Juan Bautista, Juan Alfredo. It is case sensitive.

* + Wild card “\_”: match a single character

*SELECT name*

*FROM people*

*WHERE name LIKE ‘Jua%’*

Matches names like: Juan, Juas

* **NOT LIKE:** match records that do not include a pattern

*SELECT name*

*FROM people*

*WHERE name NOT LIKE ‘Juan%’*

**-- Find strings that end with ‘r’:**

*SELECT name*

*FROM people*

*WHERE name LIKE ‘%r’*

**-- Find strings that start with ‘B’:**

*SELECT name*

*FROM people*

*WHERE name LIKE 'B%'*

**-- Find strings that have ‘r’ as a second letter:**

*SELECT name*

*FROM people*

*WHERE name LIKE '\_r%'*

**-- Filter data that includes NULL values**

*SELECT name*

*FROM people*

*WHERE birthdate IS NULL;*

**-- Count missing values**

*SELECT COUNT(\*) AS no\_birthdates*

*FROM people*

*WHERE birthdate IS NULL;*

**-- Count non-missing values**

*SELECT COUNT(\*) AS count\_birthdates*

*FROM people*

*WHERE birthdate IS NOT NULL;*

**-- Ch3: Aggregate functions**

**-- Calculate an AVERAGE (only on numerical fields)**

*SELECT AVG(budget) AS average\_budget*

*FROM films;*

**-- Calculate a SUM (only on numerical fields)**

*SELECT SUM(budget) AS total\_budget*

*FROM films;*

**-- Calculate MIN (works with non-numerical data like strings or dates)**

*SELECT MIN(budget) AS min\_budget*

*FROM films;*

**-- Calculate MAX (works with non-numerical data like strings or dates)**

*SELECT MAX(budget) AS max\_budget*

*FROM films;*

**-- Round decimal numbers: ROUND(number to round, decimal places), default is 0**

*SELECT ROUND(AVG(budget), 2) AS max\_budget*

*FROM films;*

**-- Round numbers to the nearest unit/ten/hundred, etc.**

*SELECT ROUND(AVG(budget), -3) AS max\_budget*

*FROM films;*

**-- Using arithmetic**

*SELECT (1 + 2);*

*SELECT (1 - 2);*

*SELECT (1 \* 2);*

*SELECT (4 / 3);*

**-- This returns 1 (division without remainder). To get the precise result**

*SELECT (4.0 / 3.0);*

**-- Arithmetics add the records horizontally**

**-- Substract one field from another**

*SELECT (revenue – budget) AS profit*

*FROM films;*

**-- Ch3: Sorting and grouping**

**-- Sorting results according to a field (ascending order by default) (we don’t need to select the query we are sorting for)**

*SELECT title, budget*

*FROM films*

*ORDER BY budget;*

*SELECT title, budget*

*FROM films*

*ORDER BY budget ASC;*

**-- Sort in descending order**

*SELECT title, budget*

*FROM films*

*ORDER BY budget DESC;*

*SELECT title, budget*

*FROM films*

*WHERE budget IS NOT NULL*

*ORDER BY budget DESC;*

**-- Sort by multiple fields**

*SELECT title, budget*

*FROM films*

*ORDER BY budget DESC, title ASC;*

**-- GROUP BY a single field (here, get amount of movies by certification type)**

*SELECT certification, COUNT(title) AS title\_count*

*FROM films*

*GROUP BY certification;*

**-- GROUP BY returns an error if you try to select a field which is not present in the GROUP BY clause. This gives an error:**

*SELECT certification, title*

*FROM films*

*GROUP BY certification;*

**-- You would need to add an aggregate function around title**

**-- Group by multiple fields**

*SELECT certification, language, COUNT(title) AS title\_count*

*FROM films*

*GROUP BY certification, language;*

**-- You can also reference columns by their position in the SELECT statement**

*SELECT certification, language, COUNT(title) AS title\_count*

*FROM films*

*GROUP BY 1, 2;*

**-- Group by, make a calculation, and order the results**

*SELECT*

*certification,*

*COUNT(title) as title\_count*

*FROM films*

*GROUP BY certification*

*ORDER BY title\_count DESC;*

**Filtering grouped data.**

**We can’t filter aggregate functions with WHERE clauses. For example, this will NOT work:**

*SELECT release\_year, COUNT(title) AS title\_count*

*FROM films*

*GROUP BY release\_year*

*WHERE COUNT(title) > 10;*

**You need to use a HAVING clause**

*SELECT release\_year, COUNT(title) AS title\_count*

*FROM films*

*GROUP BY release\_year*

*HAVING COUNT(title) > 10;*

**A more complex example**

*SELECT*

*certification,*

*COUNT(title) AS title\_count*

*FROM films*

*WHERE certification*

*IN (‘G’, ‘PG’, ‘R’)*

*GROUP BY certification*

*HAVING COUNT(title) > 500*

*ORDER BY title\_count DESC*

*LIMIT 3;*

**-- JOINING DATA IN SQL**

**The join clause is usually written before the select clause, mainly because of aliasing.**

**INNER JOIN: returns records whose key is present in both tables**

*SELECT prime\_ministers.country, prime\_ministers.continent, prime\_minister, president*

*FROM prime\_ministers*

*INNER JOIN presidents*

*ON prime\_ministers.country = presidents.country;*

**When selecting fields that exist in both tables, you need to write table\_name.field\_name**

**You can make this easier by aliasing tables**

*SELECT p1.country, p1.continent, prime\_minister, president*

*FROM prime\_ministers AS p1*

*INNER JOIN presidents AS p2*

*ON p1.country = p2.country;*

**Also, when you join on identical column names, you can make it more succinct**

*SELECT p1.country, p1.continent, prime\_minister, president*

*FROM prime\_ministers AS p1*

*INNER JOIN presidents AS p2*

*USING(country);*

**When writing joins, many SQL users prefer to write the SELECT statement *after* writing the join code, in case the SELECT statement requires using table aliases.**

**Defining table relationships:**

**- One-to-may: author to books**

**- One-to-one: individual to fingerprints**

**- Many-to-many: countries to languages**

**MULTIPLE JOINS combined in a single query**

*SELECT \**

*FROM left\_table*

*INER JOIN right\_table*

*ON left\_table.id = right\_table.id*

*IINER JOIN another\_table*

*ON left\_table.id = another\_table.id*

**Joining on MULTIPLE FIELDS**

*SELECT \**

*FROM left\_table*

*INNER JOIN right\_table*

*ON left\_table.id = right\_table.id*

AND *left\_table.date = right\_table.date*

**LEFT JOIN**

*SELECT p1.country, prime\_minister, president*

*FROM prime\_ministers AS p1*

*LEFT JOIN presidents AS p2*

*USING(country)*

***It can also be written as LEFT OUTER JOIN***

**RIGHT JOIN**

*SELECT \**

*FROM left\_table*

*RIGHT JOIN right\_table*

*ON left\_table.id = right\_table.id*

***It can also be written as RIGHT OUTER JOIN***

**FULL JOIN: combines left and right joins**

*SELECT p1.country AS country, prime\_minister, president*

*FROM prime\_minister AS p1*

*FULL JOIN presidents as p2*

*ON p1.country = p2.country\_code;*

**CROSS JOIN: creates all possible combinations of the ids from the two tables**

*SELECT id1, id2*

*FROM table1*

*CROSS JOIN table2;*

***SELF JOIN: a table is joined to itself. They are used to compare values from part of a table from other values within the same table.***

**Suppose you want to create a table with the pair of countries from the same continents, and you have a table that has ‘country’ and ‘continent’ fields**

*SELECT*

*c1.country as country1*

*c2.country as country2*

*c1.continent*

*FROM countries as c1*

*INNER JOIN countries as c2*

*ON c1.conitnent = c2.continent;*

*AND c1.country <> c2.country* **– exclude joins with two equal countries**

**Ch3: Set theory for SQL joins**

**Diagram

Description automatically generated**

**!!! For all set operations, the number of selected columns and their respective data types must be identical.**

**UNION: takes two tables as input and returns all records from both tables. If two records are identical, union only returns them once.**

*SELECT \**

*FROM left\_table*

*UNION*

*SELECT \**

*FROM right\_table;*

*SELECT monarch AS leader, country*

*FROM monarchs*

*UNION*

*SELECT prime\_minister, country*

*FROM prime\_ministers*

*ORDER BY country, leader;*

**The ‘monarch’ and ‘prime\_minister’ fields will be combined under ‘leader’ even though we only aliased the monarch field.**

**UNION: the same as UNION but returns identical records as duplicates.**

*SELECT \**

*FROM left\_table*

*UNION ALL*

*SELECT \**

*FROM right\_table;*

**INTERSECT: takes two tables as inputs and returns only the records that are present in both tables. It returns duplicated records only once (inner join would return duplicates)**

*SELECT id, val*

*FROM left\_table*

*INTERSECT*

*SELECT id, val*

*FROM right\_table;*

**EXCEPT: identify the records that are present in the left table but not in the right table**

*SELECT monarch, country*

*FROM monarchs*

*EXCEPT*

*SELECT prime\_minister, country*

*FROM prime\_ministers* ***– here you would bet monarchs who are not also prime\_ministers***

**Ch3: Subqueries**

**ADDITIVE JOINS: a join that adds fields to the left table. Fields with different names in both tables are added with their original names, and fields that have the same name in both columns are added so you get duplicated columns with the same name.**

**NON-ADDITIVE JOINS: they do not expressly use join key words and are not additive in the same way. Instead of using join or set operations, we can leverage the where clause to specify the records to include.**

**SEMI JOIN: chooses records in the first table where a condition is met in the second table**

**Diagram

Description automatically generated**

**For example, having one table for countries with their continent and presidents, and another table with the year of independence of countries, find the countries that gained independence before 1800:**

*SELECT country, continent, president*

*FROM countries*

*WHERE country IN*

*(SELECT country*

*FROM states*

*WHERE indep\_year < 1800);*

**This returns only the records of the fields selected from the countries table whose country field matches the list of countries returned by the SUBQUERY.**

**It will only work if the data type of ‘country’ is the same as the data type of ‘country’ in the subquery.**

**ANTI JOIN: chooses records in the first table where col1 does not find a match in col2**

**Diagram

Description automatically generated**

**For example, having one table for countries with their continent and presidents, and another table with the year of independence of countries, find the countries that gained independence after 1800:**

*SELECT country, continent, president*

*FROM countries*

*WHERE country NOT IN*

*(SELECT country*

*FROM states*

*WHERE indep\_year < 1800);*

**Subqueries inside WHERE and SELECT**

**The semi- and anti- joins seen so far involve subqueries inside WHERE clauses.**

**Subqueries in SELECT clauses**

**Eg. count the number of monarchs in each continent**

*SELECT DISTINCT continent,*

*(SELECT COUNT(\*)*

*FROM monarchs*

*WHERE states.continent = monarch.continent) AS monarch\_count*

*FROM states;*

**Count the number of cities per country**

*SELECT countries.name AS country,*

*(SELECT COUNT(name) as cities\_num*

*FROM cities*

*WHERE countries.code = cities.country\_code) AS cities\_num*

*FROM countries*

*ORDER BY cities\_num DESC, country*

*LIMIT 9;*

**SUBQUERIES inside a FROM clause**

**We can add multiple tables in a FROM clause**

*SELECT left\_table.id, left\_val*

*FROM left\_table, right\_table*

*WHERE left\_table.id = right\_table.id*

*Table

Description automatically generated with medium confidence*

**Dropping duplicates**

*SELECT DISTINCT left\_table.id, left\_val*

*FROM left\_table, right\_table*

*WHERE left\_table.id = right\_table.id*

*Table

Description automatically generated with medium confidence*

**Eg. return continents with monarchs and the year the most recent country in that continent gained independence**

SELECT DISTINCT monarchs.continent, sub.most\_recent

FROM monarchs,

(SELECT

continent,

MAX(indep\_year) as most\_recent

FROM states

GROUP BY continent) AS sub

WHERE monarchs.continent = sub.continent

ORDER BY continent;

**DATA MANIPULATION IN SQL**

**Ch1: We'll take the CASE**

**CASE statements: if-else statements. Composed of**

**- A WHEN clause: tests a given condition**

**- If the condition is true, it returns the item specified at the THEN clause**

**- If all WHEN statements are not true, it returns what is specified after the ELSE clause**

**- When the CASE statement is completed, include the term END, and then give it an alias**

*CASE WHEN x = 1 THEN ‘a’*

*WHEN x = 2 THEN ‘b’*

*ELSE ‘c’ END AS new\_column*

**E.g.: create a new column that identifies home team wins, away team wins, and ties:**

*SELECT*

*Id*

*Home\_goal*

*Away\_goal*

*CASE WHEN home\_goal > away\_goal THEN ‘Home Team Win’*

*WHEN home\_goal < away\_goal THEN ‘Away Team Win’*

*ELSE ‘Tie’ END AS match\_outcome*

*FROM match*

*WHERE season = ‘2013/2014’*

**-- Test multiple logical conditions in a case statement using AND**

*SELECT date, hometeam\_id, awayteam\_id,*

*CASE WHEN hometeam \_id = 8455 AND home\_goal > away\_goal*

*THEN ‘Chelsea home win!’*

*WHEN awayteam \_id = 8455 AND home\_goal < away\_goal*

*THEN ‘Chelsea away win!’*

*ELSE “Chelsea loss or tie” END AS outcome*

*FROM match*

*WHERE hometeam \_id = 8455 OR awayteam \_id = 8455*

**-- Removing the ELSE clause will return NULL values for the records that do not comply with the conditions**

*SELECT date, hometeam\_id, awayteam\_id,*

*CASE WHEN hometeam \_id = 8455 AND home\_goal > away\_goal*

*THEN ‘Chelsea home win!’*

*WHEN awayteam \_id = 8455 AND home\_goal < away\_goal*

*THEN ‘Chelsea away win’ END AS outcome*

*FROM match*

*WHERE hometeam \_id = 8455 OR awayteam \_id = 8455*

**-- To get rid of this NULL values, you can place the entire CASE statement in a WHERE clause and add END IS NOT NULL instead of an alias**

*SELECT date, hometeam\_id, awayteam\_id,*

*CASE WHEN hometeam \_id = 8455 AND home\_goal > away\_goal*

*THEN ‘Chelsea home win!’*

*WHEN awayteam \_id = 8455 AND home\_goal < away\_goal*

*THEN ‘Chelsea away win’ END AS outcome*

*FROM match*

*WHERE CASE WHEN hometeam \_id = 8455 AND home\_goal > away\_goal*

*THEN ‘Chelsea home win!’*

*WHEN awayteam \_id = 8455 AND home\_goal < away\_goal*

*THEN ‘Chelsea away win’ END IS NOT NULL*

**-- CASE statements with AGGREGATE FUNCTIONS**

**-- CASE WHEN with COUNT**

**-- Eg. count all Liverpool home matches won by season**

*SELECT season*

*COUNT(CASE WHEN hometeam\_id = 8650*

*AND home\_goal > away\_goal*

*THEN id END) AS home\_wins*

*FROM match*

*GROUP BY season*

**-- Instead of returning a text after THEN, you return the match id, then it gets counted within each season. But you can actually return anything you like, because SQL will only count its instances and return a number.**

**-- CASE WHEN with SUM**

**-- Eg. count all Liverpool home goals by season**

*SELECT season*

*SUM(CASE WHEN hometeam\_id = 8650*

*THEN home\_goal END) AS home\_goals*

*FROM match*

*GROUP BY season*

**-- CASE WHEN with AVG**

**-- Eg. average Liverpool home goals by season**

*SELECT season*

*ROUND(AVG(CASE WHEN hometeam\_id = 8650*

*THEN home\_goal END), 2) AS avg\_home\_goals*

*FROM match*

*GROUP BY season*

**-- PERCENTAGES with CASE WHEN and AVG**

**-- Eg: what percentage of its home and away games did Liverpool win in each season?**

*SELECT season*

*ROUND(AVG(CASE WHEN hometeam\_id = 8650 AND home\_goal > away\_goal*

*THEN 1*

*WHEN hometeam\_id = 8650 AND home\_goal < away\_goal*

*THEN 0*

*END), 1) AS pct\_homewins,*

*ROUND(AVG(CASE WHEN awayteam\_id = 8650 AND home\_goal < away\_goal*

*THEN 1*

*WHEN awayteam \_id = 8650 AND home\_goal > away\_goal*

*THEN 0*

*END), 1) AS pct\_awaywins*

*FROM match*

*GROUP BY season*

**Ch3: Short and Simple Subqueries**

**-- SUBQUERY: it can be placed in any part of a query (SELECT, FROM, GROUP BY). A subquery can return a variety of information such as:**

**-- - Scalar quantities**

**-- - A list to use for filtering or joining information**

**-- - A table to extract and further transform data**

**-- Subqueries allow you to compare summarized values with detailed data, reshaping data, and combining data that cannot be joined**

**-- Simple subquery: get all matches where home scored more than the average**

*SELECT home\_goal*

*FROM match*

*WHERE home\_goal > (*

*SELECT AVG(home\_goal)*

*FROM match);*

**-- It could be evaluated independently from the outer query**

*SELECT AVG(home\_goal)*

*FROM match;*

**-- It is evaluated only once in the entire query. SQL process the subquery, gets the information it needs, and moves on to processing information in the outer query**

**-- SUBQUERY in the WHERE clause. Useful for filtering results**

*SELECT date, hometeam\_id, awayteam\_id,*

*FROM match*

*WHERE season = ‘2012/2013’*

*AND home\_goal > (SELECT AVG(home\_goal)*

*FROM match);*

**-- SUBQUERY filtering list with IN**

*SELECT team\_long\_name*

*team\_short\_name AS abbr*

*FROM team*

*WHERE*

*team\_api\_id IN*

*(SELECT hometeam\_id*

*FROM match*

*WHERE country\_id = 1234);*

**-- SUBQUERY in a FROM statement. Useful to**

**-- - Restructure and transform your data (eg from long to wide, or prefiltering it)**

**-- - Calculating aggregate of aggregates**

**Eg: get the first 3 teams with the highest average home goals scores in the 2011/2012 season**

*SELECT team, home\_avg*

*FROM (SELECT*

*t.team\_long\_name as team,*

*AVG(m.home\_goal) as home\_avg*

*FROM match as m*

*LEFT JOIN team as t*

*ON m.hometeam.id = t.team\_api\_id*

*WHERE season = ‘2011/2012’*

*GROUP BY team) AS subquery*

*ORDER BY home\_avg DESC*

*LIMIT 3;*

**-- SUBQUERY in SELECT: include an aggregate value to compare individual values, used in mathematical calculations**

**-- Eg compare the total number of matches played in each season to the total number of matches overall**

*SELECT*

*season,*

*COUNT(id) AS matches*

*(SELECT COUNT(id) FROM match) AS total\_matches*

*FROM match*

*GROUP BY season*

**-- The subquery needs to return a single value**

**-- Properly filter each subquery: where clauses usually have to be repeated in the query and in the subquieries**

**-- Annotate SQL code**

**/\* This is a multiple**

**line comment \*/**

SELECT **-- This is an inline commnet**

**Ch3: Correlated Queries, Nested Queries, and Common Table Expressions**

**-- CORRELATED SUBQUERIES: special kind of subqueries that use values from the outer subquery in order to generate the final results. The subquery is re-executed each time a new row in the final data set is returned in order to properly generate each new piece of information.**

**-- They are used for special types of calculations such as: advanced joining, filtering, and evaluating data**

**-- For example, if you want to join the match data base with the team data base to retrieve both the name of the home and away teams, you cannot do this with only one join.**

**-- E.g.: which match stages have a higher than average number of goals scored?**

*SELECT*

*s.stage,*

*ROUND(s.avg\_goals, 2) as avg\_goal,*

*(SELECT AVG(home\_goal + away\_goal)*

*FROM match*

*WHERE season = ‘2012/2013’) as overall\_avg*

*FROM (SELECT*

*stage,*

*AVG(home\_goal + away\_goal) as avg\_goals*

*FROM match*

*WHERE season = ‘2012/2013’*

*GROUP BY stage) AS s*

*WHERE s. avg\_goals > (SELECT AVG(home\_goal + away\_goal)*

*FROM match as m*

*WHERE s.stage > m.stage)*

**-- Differences between simple subqueries (1) and correlated subqueries (2)**

**-- - (1) can be run independently from the main query, while (2) cannot be executed on its own because it is dependent on the value of the main query**

**-- - (1) is evaluated only once in the entire statement, while (2) Is evaluated in loops, once for each row generated by the data set (will slow performance)**

**-- E.g: what is the average number of goals scored in each country across all seasons?**

**-- - This is the way to do it without a correlated subquery**

*SELECT*

*c.name as country*

*AVG(home\_goal + away\_goal) AS avg\_goals*

*FROM country AS c*

*LEFT JOIN matches AS m*

*ON c.id = m.country\_id*

*GROUP BY country;*

**-- This is the way to do it with a correlated query, which does not need to use the join**

*SELECT*

*c.name as country*

*(SELECT*

*AVG(home\_goal + away\_goal)*

*FROM match AS m*

*WHERE m.country\_id = c.id) AS avg\_goals*

*FROM country as c*

*GROUP BY country;*

**-- Another example:** examine matches with scores that are extreme outliers for each country -- above 3 times the average score

SELECT

    -- Select country ID, date, home, and away goals from match

    main.country\_id,

    date,

    main.home\_goal,

    main.away\_goal

FROM match AS main

WHERE

    -- Filter the main query by the subquery

    (home\_goal + away\_goal) >

        (SELECT AVG((sub.home\_goal + sub.away\_goal) \* 3)

         FROM match AS sub

         -- Join the main query to the subquery in WHERE

         WHERE main.country\_id = sub.country\_id);

**-- Another example:** what was the highest scoring match for each country, in each season?

SELECT

    -- Select country ID, date, home, and away goals from match

    main.country\_id,

    date,

    main.home\_goal,

    main.away\_goal

FROM match AS main

WHERE

    -- Filter for matches with the highest number of goals scored

    (home\_goal + away\_goal) =

        (SELECT MAX(sub.home\_goal + sub.away\_goal)

         FROM match AS sub

         WHERE main.country\_id = sub.country\_id

               AND main.season = sub.season);

**-- NESTED SUBQUERIES: subqueries within subqueries**

-- Sometimes you require multiple layers of transformation and filtering of data before extracting the data in the main query.

-- Eg: How does each month’s total goals differ from the average monthly total of goals scored?

*SELECT*

*EXTRACT (MONTH FROM date) as month*

*SUM(m.home\_goal + m.away\_goal) AS total\_goals,*

*SUM(m.home\_goal + m.away\_goal) –*

*(SELECT AVG(goals)*

*FROM (SELECT*

*EXTRACT (MONTH FROM date) as month*

*SUM(m.home\_goal + m.away\_goal) AS goals*

*FROM match*

*GROUP BY month)) AS avg\_diff*

*FROM match as m*

*GROUP BY month;*

**-- Extracting DATE objects from a date (MONTH, YEAR)**

*SELECT*

*EXTRACT(YEAR FROM date) AS year,*

*EXTRACT(MONTH FROM date) AS month*

*EXTRACT (YEAR\_MONTH FROM date) as year\_and\_month*

*FROM dates;*

*SELECT*

*YEAR(date) AS year,*

*MONTH(date) AS month,*

*MONTHNAME(date) AS month\_name,*

*DATE\_FORMAT(date, '%Y-%m') AS year\_and\_month*

*FROM dates;*

**-- CORRELATED NESTED SUBQUERIES**

-- What is each country’s average goals scored in the 2011/2012 season?

*SELECT*

*c.name as country*

*(SELECT AVG(home\_goal + away\_goal),*

*FROM match as m*

*WHERE m.country\_id = c.id*

*AND id IN (* **-- Here id is match id**

*SELECT id*

*FROM match*

*WHERE season = ‘2011/2012’)) AS ag\_goals*

*FROM country as c*

**-- Another example:** get the max number of goals per match for each season, the max number of goals per match for all the data, and the max number of goals per match in the month of July across all seasons

SELECT

    -- Select the season and max goals scored in a match

    season,

    MAX(home\_goal + away\_goal) AS max\_goals,

    -- Select the overall max goals scored in a match

   (SELECT MAX(home\_goal + away\_goal) FROM match) AS overall\_max\_goals,

   -- Select the max number of goals scored in any match in July

   (SELECT MAX(home\_goal + away\_goal)

    FROM match

    WHERE id IN (

          SELECT id FROM match WHERE EXTRACT(MONTH FROM date) = 07)) AS july\_max\_goals

FROM match

GROUP BY season;

**-- Another example:** What's the average number of matches per season where a team scored 5 or more goals? How does this differ by country?

SELECT

  c.name AS country,

    -- Calculate the average matches per season

  AVG(outer\_s.matches) AS avg\_seasonal\_high\_scores

FROM country AS c

-- Left join outer\_s to country

LEFT JOIN (

  SELECT country\_id, season,

         COUNT(id) AS matches

  FROM (

    SELECT country\_id, season, id

  FROM match

  WHERE home\_goal >= 5 OR away\_goal >= 5) AS inner\_s

  -- Close parentheses and alias the subquery

  GROUP BY country\_id, season) AS outer\_s

ON c.id = outer\_s.country\_id

GROUP BY country;

**-- COMMON TABLE EXPRESSIONS (CTEs): common method for improving readability and accessibility of information in subqueries**

**-- They are a special type of subquery that are declared ahead of your main query**

**-- Instead of wrapping subqueries inside, say, you FROM statement, you name it using a WITH statement and then reference it by name later in your FROM as if it were any other table on your data base**

**-- Benefits over using a subquery elsewhere: the CTE is run only once and the stored in memory, it is better for organizing long complex queries, you can reference another CTE inside CTE, and a CTE can reference itself in a recursive CTE (explained in another course)**

*WITH cte1 AS (*

*SELECT col1, col2*

*FROM table1),*

*cte2 AS (SELECT col1, col2*

*FROM table2)*

*SELECT AVG(col1 + col2) AS avg\_cols*

*FROM cte;*

**-- Example:** count the number of matches with more than 10 goals for each country

WITH s AS (

SELECT country\_id, id

FROM match

WHERE (home\_goal + away\_goal) >= 10

)

SELECT

c.name AS country

COUNT(s.id) AS matches

FROM country AS c

INNER JOIN s

ON c.id = s.country\_id

GROUP BY country;

**-- RELEVANT EXAMPLE: three ways to do the same thing**

**--** How do you get both the home and away team names into one final query result?

**-- SUBQUERIES in FROM + JOINS**

SELECT

  m.date,

    -- Get the home and away team names

    home.hometeam,

    away.awayteam,

    m.home\_goal,

    m.away\_goal

FROM match AS m

-- Join the home subquery to the match table

LEFT JOIN (

  SELECT match.id, team.team\_long\_name AS hometeam

  FROM match

  LEFT JOIN team

  ON match.hometeam\_id = team.team\_api\_id) AS home

ON home.id = m.id

-- Join the away subquery to the match table

LEFT JOIN (

  SELECT match.id, team.team\_long\_name AS awayteam

  FROM match

  LEFT JOIN team

  -- Get the away team ID in the subquery

  ON match.awayteam\_id = team.team\_api\_id) AS away

ON away.id = m.id;

­**-- CORRELATED SUQUERIES**

SELECT

    m.date,

    (SELECT team\_long\_name

     FROM team AS t

     WHERE t.team\_api\_id = m.hometeam\_id) AS hometeam,

    -- Connect the team to the match table

    (SELECT team\_long\_name

     FROM team AS t

     WHERE t.team\_api\_id = m.awayteam\_id) AS awayteam,

    -- Select home and away goals

     m.home\_goal,

     m.away\_goal

FROM match AS m;

**-- CTEs**

WITH home AS (

  SELECT m.id, m.date,

       t.team\_long\_name AS hometeam, m.home\_goal

  FROM match AS m

  LEFT JOIN team AS t

  ON m.hometeam\_id = t.team\_api\_id),

-- Declare and set up the away CTE

away AS (

  SELECT m.id, m.date,

       t.team\_long\_name AS awayteam, m.away\_goal

  FROM match AS m

  LEFT JOIN team AS t

  ON m.awayteam\_id = t.team\_api\_id)

-- Select date, home\_goal, and away\_goal

SELECT

  home.date,

    home.hometeam,

    away.awayteam,

    home.home\_goal,

    away.away\_goal

-- Join away and home on the id column

FROM home

INNER JOIN away

ON home.id = away.id;

**Ch4: Window functions**

**-- Another common limitation you encounter is that you must group results when using aggregate functions. Thus you can’t compare aggregate values to non aggregate data. You can work around this window functions.**

**-- WINDOW FUNCTION: a class of functions that can perform calculations on a result set that has been already generated (a ‘window’).**

**-- You can use them to perform aggregate calculations without grouping your data.**

**-- - Similar to subqueries in SELECT**

**-- - Can calculate: running totals, rankings, moving averages**

**-- Example of replacing a subquery in SELECT:** how many goals were scored in each match in 2011/2012 and how did it compare to the average?

**-- First solve it with a subquery in select:**

*SELECT*

*date,*

*(home\_goal + away\_goal) AS goals,*

*(SELECT AVG(home\_goal + away\_goal)*

*FROM match*

*WHERE season = ‘2011/2012’) AS overall\_avg*

*FROM match*

*WHERE season = ‘2011/2012’;*

**-- Now using a window function**

*SELECT*

*date,*

*(home\_goal + away\_goal) AS goals,*

*AVG(home\_goal + away\_goal) OVER() AS overall\_avg* **– OVER() tells SQL, ‘pass this aggregate value over the existing subset’**

*FROM match*

*WHERE season = ‘2011/2012’;*

**-- Example of calculating a RANK:** what is the rank of matches based on the number of goals scored in 2011/2012?

*SELECT*

*date,*

*(home\_goal + away\_goal) AS goals,*

*RANK() OVER(ORDER BY home\_goal + away\_goal DESC) AS goals\_rank*

**-- A particular behaviour of RANK is that, if there are two rows with the same values (eg, 5 and 5 goals) then both will have the same number of rank (eg. 1º and 1º), but then the next row (e.g. 4 goals) will have 3º as a rank.**

*FROM match*

*WHERE season = ‘2011/2012’;*

***-- Another example:*** create a data set of ranked matches according to which leagues, on average, score the most goals in a match.

SELECT

    -- Select the league name and average goals scored

    l.name AS league,

    AVG(m.home\_goal + m.away\_goal) AS avg\_goals,

    -- Rank leagues in descending order by average goals

    RANK() OVER(ORDER BY AVG(m.home\_goal + m.away\_goal) DESC) AS league\_rank

FROM league AS l

LEFT JOIN match AS m

ON l.id = m.country\_id

WHERE m.season = '2011/2012'

GROUP BY l.name

ORDER BY league\_rank;

**-- Key considerations when using window functions:**

**-- -they are processed after the entire query, except the final ORDER BY statement, thus they use the result set to calculate information as opposed to using the data base directly.**

**-- - They are NOT available in SQLite**

**-- OVER with a PARTITION BY**

**-- A partition allows you to calculate separate values for different categories established within a partition. This is one way to calculate different aggregate values within one column of data and pass them down a data set, instead of having to calculate them in different columns.**

**--** Eg: average for each season

*AVG(home\_goal) OVER(PARTITION BY season)*

**-- In a query (first without partition):** how many goals were scored in each match, and how did that compare to the overall average?

*SELECT*

*Date,*

*(home\_goal + away\_goal) as goals,*

*AVG(home\_goal + away\_goal) OVER() AS overall\_avg*

*FROM match;*

**-- In a query (using PARTITION BY):** how many goals were scored in each match, and how did that compare to the season’s average?

*SELECT*

*Date,*

*(home\_goal + away\_goal) as goals,*

*AVG(home\_goal + away\_goal) OVER(PARTITION BY season) AS season\_avg*

*FROM match;*

**-- PARTITION BY multiple columns**

*SELECT*

*c.name as country,*

*m.season,*

*(home\_goal + away\_goal) AS goals,*

*AVG(home\_goal + away\_goal) OVER(PARTITION BY season, c.name) AS season\_country\_avg*

*FROM country AS c*

*LEFT JOIN match AS m*

*ON c.id = m.country\_id;*

**- Another example**: creating a data set of games played by Legia Warszawa (Warsaw League), the top ranked team in Poland, and comparing their individual game performance to the overall average for that season.

SELECT

    date,

    season,

    home\_goal,

    away\_goal,

    CASE WHEN hometeam\_id = 8673 THEN 'home'

         ELSE 'away' END AS warsaw\_location,

    -- Calculate the average goals scored partitioned by season

    AVG(home\_goal) OVER(PARTITION BY season) AS season\_homeavg,

    AVG(away\_goal) OVER(PARTITION BY season) AS season\_awayavg

FROM match

-- Filter the data set for Legia Warszawa matches only

WHERE

    hometeam\_id = 8673

    OR awayteam\_id = 8673

ORDER BY (home\_goal + away\_goal) DESC;

**-- SLIDING WINDOWS**

**-- Window functions can be used to calculate information that changes with each subsequent row in a data set. These types of window functions are called “sliding windows”.**

**-- These are functions that perform calculations relative to the current row of a data set.**

**-- You can use them to calculate information that aggregates one row at a time down your data set: running totals, count, moving averages, etc.**

**-- They can be partitioned by one or more columns.**

**-- It contains specific functions within the over clause to specify the data you want to use in your calculations. The general syntax is:**

*ROWS BETWEEN <start> AND <finish>*

**-- For the “start” and “finish” in the statement, you can use the following keywords:**

**-- - PRECEDING: specify the number of rows before the current row that you want to include in a calculation**

**-- - FOLLOWING: specify the number of rows after the current row that you want to include in a calculation**

**-- - UNBOUNDED PRECEDING: include very row since the beginning of the data set in the calculation**

**-- - UNBOUNDED FOLLOWING: include very row since the enf of the data set in the calculation**

**-- - CURRENT ROW: stop calculation at the current row**

**-- Example:** calculate the sum of home goals for Manchester City from the start of the 2012/2013 season for each match in the season:

*SELECT*

*date,*

*home\_goal,*

*away\_goal,*

*SUM(home\_goal)*

*OVER(ORDER BY date ROWS BETWEEN*

*UNBOUNDED PRECEDING AND CURRENT ROW) as running\_total*

*FROM match*

*WHERE hometeam\_id = 8990 AND season = ‘2012/2013’*

*A picture containing table

Description automatically generated*

**-- SLIDING WINDOW FRAME: you can calculate sliding windows with a more limited frame**

**-- Example:** create a column with the home goals for Manchester City for the previous match in season ‘2012/2013’

*SELECT*

*date,*

*home\_goal,*

*away\_goal,*

*SUM(home\_goal)*

*OVER(ORDER BY date ROWS BETWEEN*

*1 PRECEDING AND 1 PRECEDING) as last2*

*FROM match*

*WHERE hometeam\_id = 8990 AND season = ‘2012/2013’*

**-- BRINGING IT ALL TOGETHER**

**-- Example:** who defeated Machester City in the 2013/2014 season? Get team names with CTE, get the match outcome with a CASE statement, ranking matches by the number of goals they lost a match using a WINDOW FUNCTION

-- Set up the home team CTE

WITH home AS (

  SELECT m.id, t.team\_long\_name,

    CASE WHEN m.home\_goal > m.away\_goal THEN 'MU Win'

       WHEN m.home\_goal < m.away\_goal THEN 'MU Loss'

         ELSE 'Tie' END AS outcome

  FROM match AS m

  LEFT JOIN team AS t ON m.hometeam\_id = t.team\_api\_id),

-- Set up the away team CTE

away AS (

  SELECT m.id, t.team\_long\_name,

    CASE WHEN m.home\_goal > m.away\_goal THEN 'MU Loss'

       WHEN m.home\_goal < m.away\_goal THEN 'MU Win'

         ELSE 'Tie' END AS outcome

  FROM match AS m

  LEFT JOIN team AS t ON m.awayteam\_id = t.team\_api\_id)

-- Select columns and rank the matches by goal difference

SELECT DISTINCT

    date,

    home.team\_long\_name AS home\_team,

    away.team\_long\_name AS away\_team,

    m.home\_goal, m.away\_goal,

    RANK() OVER(ORDER BY ABS(home\_goal - away\_goal) DESC) as match\_rank

-- Join the CTEs onto the match table

FROM match AS m

LEFT JOIN home ON m.id = home.id

LEFT JOIN away ON m.id = away.id

WHERE m.season = '2014/2015'

      AND ((home.team\_long\_name = 'Manchester United' AND home.outcome = 'MU Loss')

      OR (away.team\_long\_name = 'Manchester United' AND away.outcome = 'MU Loss'))

ORDER BY match\_rank;

**POSTGRESQL SUMMARY STATS AND WINDOW FUNCTIONS**

Graphical user interface, text, application, email

Description automatically generated

**-- WINDOW FUNCTIONS:** perform an operation across a set of rows that are somehow related to the current row. They are similar to group by aggregate functions, but instead of all rows being grouped into a single row, all rows remain in the output.

-- What can you do with them:

-- - Assign row numbers

-- - Fetch values from preceding or following rows

-- - Assign ordinal ranks to rows based on their ‘values’ position in a sorted list.

-- - Running total, moving averages.

**-- Syntax of a window function**

*FUNCTION\_NAME() OVER(…), where … can be*

*-- ORDER BY*

*-- PARTITION BY*

*-- ROWS/RANGE PRECEDING/FOLLOWING/UNBOUNDING*

**-- Assign a ROW NUMBER to each row**

-- Example: assign row numbers to easily see, for example, in which year the 13th summer Olympic was held.

SELECT

  Year,

  -- Assign numbers to each year

  ROW\_NUMBER() OVER() AS Row\_N

FROM (

  SELECT DISTINCT year

  FROM Summer\_Medals

  ORDER BY Year ASC

) AS Years

ORDER BY Year ASC;

**-- ORDER BY:** one of the subclauses in the OVER() clause. Bear in mind that the ORDER BY inside the OVER clause takes effect before any ORDER BY outside it.

**-- LAG(column, n): OVER(…):** returns column’s value at the row n rows before the current row. Default is 1 lag.

**--** Example: get the last country that won

Text, timeline

Description automatically generated with medium confidenceA picture containing text

Description automatically generated

**-- PARTITION BY:** an OVER() subclause that splits the table into partitions based on a column’s unique values. Unlike group by, the results aren’t rolled into one column.

-- Partitions are operated on separately by window function. For example, ROW\_NUMBER() will reset for each partition; LAG() will only fetch a row’s previous value if its previous row is in the same partition.

-- Example: create the column ‘preceding champion’ for more than one type of Event

WITH Discuss\_Gold AS (…)

SELECT

Year, Event, Champion,

LAG(Champion) OVER(

PARTITION BY Event

ORDER BY Event ASC, Year ASC) AS Last\_Champion

FROM Discuss\_Gold

ORDER BY Event ASC, Year ASC;

**Ch2: Fetching, ranking, and paging**

**-- FETCHING: fetch values from different parts of the tables into one row.**

**-- - LAG(column, n): returns column’s value at the row n before the current row. (relative fetching function)**

**-- - LEAD(column, n): returns column’s value at the row n after the current row. (relative fetching function)**

**-- - FIRST\_VALUE(column): returns the first value in a table or partition. (absolute fetching function)**

**-- - LAST\_VALUE(column): returns the last value in a table or partition. (absolute fetching function)**

**-- !!! By default, a window starts at the beginning of a table or partition and ends at the current row. One should take into account this if wanting to select the last row of the table, and not the current row (see example below).**

**- Example:** create columns for the next two host cities for the Olympics

*WITH Hosts AS (*

*SELECT DISTINCT Year, City*

*FROM Summer\_Medals)*

*SELECT*

*Year, City,*

*LEAD(City, 1) OVER(ORDER BY Year ASC) AS Next\_City,*

*LEAD(City, 2) OVER(ORDER BY Year ASC) AS After\_Next\_City*

*FROM Hosts*

*ORDER BY Year ASC;*

**A picture containing graphical user interface

Description automatically generated**

**-- Example:** get the first and last host city for the Olympics

*SELECT*

*Year, City*

*FIRST\_VALUE(City) OVER(ORDER BY Year ASC) AS First\_City,*

*LAST\_VALUE(City) OVER (*

*ORDER BY Year ASC*

*RANGE BETWEEN*

*UNBOUNDED PRECEDING AND*

*UNBOUNDED FOLLOWING) AS Last\_City* **– The “unbounded” part is to tell SQL that it must return the last row un the table, and not the current row.**

*FROM Hosts*

*ORDER BY Year ASC;*

**A picture containing graphical user interface

Description automatically generated**

**-- RANKING FUNCTIONS: ranking rows according to their values.**

**-- - ROW\_NUMBER(): it always assigns different numbers even if those numbers are the same**

**-- - RANK(): assigns the same number to rows with identical values, skipping over the numbers in such cases**

**-- - DENSE\_RANK(): assigns the same number to rows with identical values, but does not skip over the next numbers**

**-- Example:** ranking countries by the amount of Olympic they participated in, assigning the same number to countries with the same amount, but skipping over those numbers.

*WITH Country\_Games AS (SELECT*

*Country, COUNT(DISTINCT Year) AS Games*

*FROM Summer\_medals*

*GROUP BY Country*

*ORDER BY Games DESC;)*

SELECT

Country,

RANK() OVER(ORDER BY Games DESC) AS Ranking

FROM *Country\_Games*

*ORDER BY Games DESC, Country ASC*

**Calendar

Description automatically generated with medium confidence**

**-- Example:** Rank each country's athletes by the count of medals they've earned -- the higher the count, the higher the rank -- without skipping numbers in case of identical values.

WITH Athlete\_Medals AS (

  SELECT

    Country, Athlete, COUNT(\*) AS Medals

  FROM Summer\_Medals

  WHERE

    Country IN ('JPN', 'KOR')

    AND Year >= 2000

  GROUP BY Country, Athlete

  HAVING COUNT(\*) > 1)

SELECT

  Country,

  -- Rank athletes in each country by the medals they've won

  Athlete,

  DENSE\_RANK() OVER (PARTITION BY Country

                ORDER BY Medals DESC) AS Rank\_N

FROM Athlete\_Medals

ORDER BY Country ASC, RANK\_N ASC;

**­­--PAGING: splitting data into approximately equal chunks.**

**-- - Many APIs return data in ‘pages’ to reduce the data being sent**

**-- - When data is sorted by a metric, separating it into quantiles may be useful.**

**-- NTILE(n): split the data into (approximately) n equal pages (it creates a new column with a number that determines to which page each row belongs to)**

**-- Example:** split all Olympic routines into 15 groups

*WITH Disciplines AS (SELECT*

*DISTINCT Discipline*

*FROM Summer\_Medals)*

*SELECT*

*Discipline, NTILE(15) OVER() AS Page*

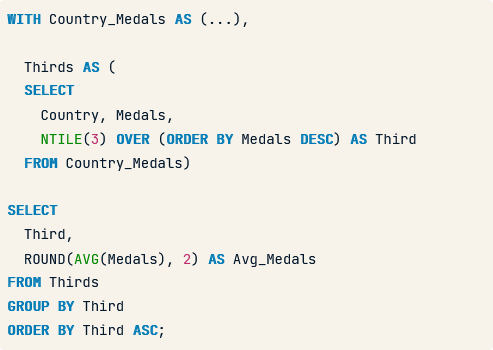
*FROM Disciplines*

*ORDER BY Page ASC;*

**-- Example:** calculate the average number of medals for the three tertiles of countries with the most medals won.

**Graphical user interface, text, application, email

Description automatically generated**

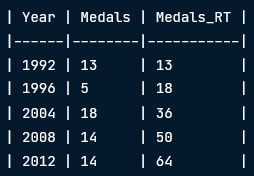
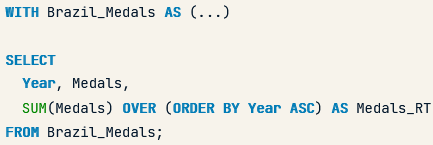
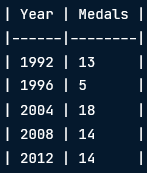
** A picture containing table

Description automatically generated**

**Ch3: Aggregate window functions and frames**

**-- AGGREGATE WINDOW FUNCTIONS: you can use aggregate functions (MAX(), MIN(), SUM(), AVG(), COUNT()) as window functions.**

**-- Example:** calculate the cumulative sum /running total of medals earned so far by Brazil

****

**-- FRAMES: another way to change a window function’s behaviour. Eg: “RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING”.**

**-- By default, a frame starts at the beginning of a table or partition and ends at the current row.**

**-- A frame always starts with RANGE BETWEEN.. or ROWS BETWEEN….**

**-- RANGE BETWEEN [START] AND [END]**

**-- - n PRECEDING: n rows before the current row**

**-- - CURRENT ROW**

**-- - n FOLLWING: n rows after the current row**

**-- Example:** Return the athletes, medals earned, and the maximum medals earned, comparing only the last two and current athletes, ordering by athletes' names in alphabetical order.

WITH Chinese\_Medals AS (

  SELECT

    Athlete, COUNT(\*) AS Medals

  FROM Summer\_Medals

  WHERE

    Country = 'CHN' AND Medal = 'Gold'

    AND Year >= 2000

  GROUP BY Athlete)

SELECT

  -- Select the athletes and the medals they've earned

  Athlete,

  Medals,

  -- Get the max of the last two and current rows' medals

  MAX(Medals) OVER (ORDER BY Athlete ASC

            ROWS BETWEEN 2 PRECEDING

            AND CURRENT ROW) AS Max\_Medals

FROM Chinese\_Medals

ORDER BY Athlete ASC;

**-- MOVING AVERAGE:**

**Shape

Description automatically generated with low confidenceGraphical user interface, application

Description automatically generated**

**-- Example:** calculate a 3-year moving average for Russia’s gold medals

WITH Russian\_Medals AS (

  SELECT

    Year, COUNT(\*) AS Medals

  FROM Summer\_Medals

  WHERE

    Country = 'RUS'

    AND Medal = 'Gold'

    AND Year >= 1980

  GROUP BY Year)

SELECT

  Year, Medals,

  --- Calculate the 3-year moving average of medals earned

  AVG(Medals) OVER

    (ORDER BY Year ASC

     ROWS BETWEEN

     1 PRECEDING AND 1 FOLLOWING) AS Medals\_MA

FROM Russian\_Medals

ORDER BY Year ASC;

**-- MOVING TOTAL:**

**Graphical user interface, text, application

Description automatically generated**

**-- Difference between ROWS and RANGE in the frame: RANGE treats duplicates in the columns in the ORDER BY subclause inside OVER() as single entities, whereas ROWS does not.**

**A screenshot of a computer

Description automatically generated with low confidence**

**-- EXTENSIONS: equivalent of packages of functions**

**-- Syntax:** *CREATE EXTENSION OF NOT EXISTS tablefunc;.* **“Create extension” makes extra functions in an extension (here, tablefunc) available for use.**

**Ch4: Beyond window functions**

**- PIVOTING: transform tables from/to wide/long format.**

**- CROSSTAB(column): pivot a table by a certain column (from the extension “tablefunc”)**

**- Sintax:**

*CREATE EXTENSION OF NOT EXISTS tablefunc;.*

*SELECT \* FROM CROSSTAB($$*

*source\_sql TEXT*

*$$) AS ct (column\_1 DATA\_TYPE\_1,*

*column\_2 DATA\_TYPE\_2,*

*.*

*.*

*.*

*column\_n DATA\_TYPE\_N);*

**-- Example:**

****

-- Create the correct extention to enable CROSSTAB

CREATE EXTENSION IF NOT EXISTS tablefunc;

SELECT \* FROM CROSSTAB($$

  SELECT

    Gender, Year, Country

  FROM Summer\_Medals

  WHERE

    Year IN (2008, 2012)

    AND Medal = 'Gold'

    AND Event = 'Pole Vault'

  ORDER By Gender ASC, Year ASC;

-- Fill in the correct column names for the pivoted table

$$) AS ct (Gender VARCHAR,

           "2008" VARCHAR,

           "2012" VARCHAR)

ORDER BY Gender ASC;

Table

Description automatically generated

CREATE EXTENSION IF NOT EXISTS tablefunc;

SELECT \* FROM CROSSTAB($$

  WITH Country\_Awards AS (

    SELECT

      Country,

      Year,

      COUNT(\*) AS Awards

    FROM Summer\_Medals

    WHERE

      Country IN ('FRA', 'GBR', 'GER')

      AND Year IN (2004, 2008, 2012)

      AND Medal = 'Gold'

    GROUP BY Country, Year)

  SELECT

    Country,

    Year,

    RANK() OVER

      (PARTITION BY Year

       ORDER BY Awards DESC) :: INTEGER AS rank

  FROM Country\_Awards

  ORDER BY Country ASC, Year ASC;

-- Fill in the correct column names for the pivoted table

$$) AS ct (Country VARCHAR,

           "2004" INTEGER,

           "2008" INTEGER,

           "2012" INTEGER)

Order by Country ASC;

**-- ROLLUP: a group by subclause that includes extra rows for group-level aggregations.**

**-- It is useful when you want to create a row that calculates group-level totals.**

**-- Example:**

**Text

Description automatically generated with medium confidence (Notice the rows with “Total”)**

*SELECT*

*Country, Medal, COUNT(\*) AS Awards*

*FROM summer\_medals,*

*WHERE*

*Year = 2008 AND Country IN(‘CHN’, ‘RUS’)*

*GROUP BY ROLLUP(Medal)*

*ORDER BY Country ASC, Medal ASC;*

**--** Grouping by Country and ROLLUP(Medal) will count all Country- and Medal-level totals, then count only Country-level totals and fill in Medal with nulls from these rows.

-- If we had included both Country and Medal inside the ROLLUP function, then we would have also calculated the grand total for the table.

**-- ROLLUP is hierarchical: the order of the columns in the function affect the output, de-aggregating from the leftmost provided column to the right-most.**

-- - ROLLUP(Country, Medal) returns Country-level totals.

-- - ROLLUP(Medal, Country) returns Medal-level totals.

-- - Both will include the grand total because all columns are included inside ROLLUP.

A screenshot of a computer

Description automatically generated with low confidence

**--** The rows with only null in Medal are country level totals. The row with null in Country and Medal contains the grand total for the table.

**-- What if you wanted both Country- and Medal-level totals?**

**­-- CUBE(): like ROLLUP, but not hierarchical. It generates all possible group-level aggregations.**

*SELECT*

*Country, Medal, COUNT(\*) AS Awards*

*FROM summer\_medals,*

*WHERE*

*Year = 2008 AND Country IN(‘CHN’, ‘RUS’)*

*GROUP BY CUBE(Country, Medal)*

*ORDER BY Country ASC, Medal ASC;*

**--** Here, CUBE generates Country-, Medal- level, and grand totals.

A screenshot of a computer

Description automatically generated with low confidence

**-- What if you want to replace the “null” values with something that indicates that these rows are group-level totals?**

**-- COALESCE(): takes a list of values and returns the first non-null value, going from left to right.**

**-- So “COALESCE(null, 1, null, 2) would return 1.**

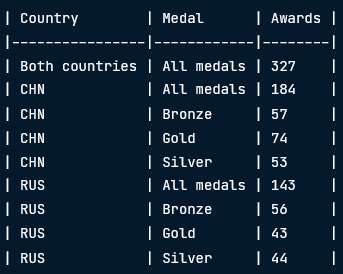
**-- Useful when using SQL operations such as**

**-- - ROLLUP and CUBE**

**-- - Pivoting**

**-- - LAG and LEAD**

**-- Example**: using the previous example, reproduce the following table



*SELECT*

*COALESCE(Country, “Both countries) AS Country,*

*COALESCE(Medal, ‘All medals’) AS Medal,*

*COUNT(\*) AS Awards*

*FROM summer\_medals,*

*WHERE*

*Year = 2008 AND Country IN(‘CHN’, ‘RUS’)*

*GROUP BY ROLLUP(Country, Medal)*

*ORDER BY Country ASC, Medal ASC;*

**-- Compressing data: when you rank a table by a newly created rank variable, and then sort it by that variable, this variable becomes redundant.**

**- Instead of returning the results as a table, you could return a list with one row, listing all names in the order according to their ranks.**

**Graphical user interface, application

Description automatically generated**

**-- STRING\_AGG(column, separator): takes all the values of a column and concatenates them, with ‘separator’ between each value.**

**Graphical user interface, application

Description automatically generated**

**FUNCTIONS FOR MANIPULATING DATA IN POSTGRE SQL**

**Ch1: Overview of Common Data Types**

**-- DATABASE: Sakila, models a DVD rental store.**

**-- COMMON DATA TYPES:**

**-- - Text: CHAR, VARCHAR (‘varying character’), TEXT**

**-- - Numeric: INT, DECIMAL**

**-- - Date/time: DATE, TIME, TIMESTAMP, INTERVAL**

**-- - Arrays**

**-- TEXT DATA TYPES: allow for a fixed or varying number of characters and string data**

**-- - TEXT type: unlimited length**

**-- Determining data types from existing tables / get the data types of your columns: PostgreSQL stores information about all database objects in a system database called “Information Schema”. By querying certain tables in this database you can determine information about it.**

*SELECT*

*column\_ name, data\_type*

*FROM INFORMATION\_SCHEMA:COLUMNS*

*Where column\_name in (‘col1’, ‘col2’) AND table\_name = ‘table1’;*

**-- Get information about all the TABLES from your database using the INFORMATION.SCHEMA**

 -- Select all columns from the TABLES system database

 SELECT \*

 FROM INFORMATION\_SCHEMA.TABLES

 -- Filter by schema (only public)

 WHERE table\_schema = 'public';

**-- DATE DATA TYPES:**

**-- - TIMESTAMP: contains both a date and a time value, with microsecond precision, yyyy-mm-dd format (ISO 8601)**

**-- - INTERVAL: stores date and time data as a period of time in year, month, days, hours, etc.**

**-- Add days (or any interval) to a date (timestamp)**

*SELECT rental\_date + INTERVAL ‘3 days’ AS expected\_return*

*FROM rental;*

**-- TIMEZONES: PostgreSQL lets you store timestamps with or without time zones.**

**-- CREATE A NEW TABLE in a database:**

*CREATE TABLE my\_first\_table(*

*first\_column text***,**

*second\_column integer*

*);*

**-- INSERT NEW ROWS into a table**

*INSERT INTO my\_first\_table*

*(first\_column, second\_column) VALUES (‘tex value’, 12);*

**-- ARRAY data type: very similar to most arrays in other programming languages.**

**-- Create an ARRAY type: add square brackets to the end of the data type**

*CREATE TABLE grades (*

*stundet\_id int,*

*email text[][],* **-- a nested array of text data to store email type and address**

*test\_scores int[]* **– an array of integer values**

*);*

**-- INSERT statements with ARRAYS**

*INSERT INTO grades*

*VALUES (1,*

*‘{{“work”, “work1@datacamp.com”}, {“other”, “other1@datacamp.com”}}’,*

*‘{94, 95, 90, 89}’);*

**-- ACCESING ARRAY data in a SELECT statement, and filter using WHERE statements**

**-- (SQL uses 1-INDEXING, not zero-indexing like Python)**

*SELECT*

*email[1]* *[1] AS type,*

*email[1]* *[2] AS address,*

*test\_scores* *[1] ,*

*FROM grades*

*WHERE email[1]* *[1] = ’work’;*

**-- ANY function to select all arrays that meet certain condition**

*SELECT*

*email[1]* *[1] AS type,*

*email[1]* *[2] AS address,*

*test\_scores* *[1] ,*

*FROM grades*

*WHERE ‘other’ = ANY(email);* **-- All emails that have ‘other’ in the text**

**-- Alternatively, you could do this with the CONTAINS operator**

*SELECT*

*email[1]* *[1] AS type,*

*email[1]* *[2] AS address,*

*test\_scores* *[1] ,*

*FROM grades*

*WHERE email @> ARRAY ['other'];*

**Ch2: Working with DATE/TIME Functions and Operators**

**-- Basic operations with dates / adding and subtracting date time data**

*SELECT date ‘2005-09-11’ - date ‘2005-09-11’;* **-- returns integer 1**

*SELECT date ‘2005-09-11’ + integer ‘3’;* **-- returns integer ‘2005-09-14’**

**-- !!! When you add integers to date objects, the implied precision is days. But if you perform the same operation on a timestamp, you get an interval as a result.**

*SELECT date ‘2005-09-11 00:00:00’ - date ‘2005-09-11 12:00:00’;* ***--* returns the interval ‘1 day 12:00:00’**

**­­-- AGE(ts1, ts2): calculate the difference between two timestamps (it subtracts ts2 from ts1)**

**-- Example:** calculate the number of days the movie was rented

SELECT f.title, f.rental\_duration,

    -- Calculate the number of days rented

    AGE(r.return\_date, r.rental\_date) AS days\_rented

FROM film AS f

    INNER JOIN inventory AS i ON f.film\_id = i.film\_id

    INNER JOIN rental AS r ON i.inventory\_id = r.inventory\_id

ORDER BY f.title;

**-- Multiplication and division using INTERVALS. Eg:** add 21 days

*SELECT timestamp ‘2005-09-01’ + 21 \* INTERVAL ‘1 day’;*

**-- Convert a DATE value to an INTERVAL**

*SELECT INTERVAL '1' day \* rental\_duration;*

­ **-- How to retrieve date-time values in your query / retrieving the current timestamp**

**-- - NOW() function**

*SELECT NOW();*

**-- - Select it without the time zone**

*SELECT NOW()::timestamp;* -- This is an example of “casting”, with a syntax exclusive to Postgre

**-- - PostgreSQL option:**

*SELECT CURRENT\_TIMESTAMP(2);* -- Where the argument specifies the digits for the seconds precision.

*SELECT CURRENT\_DATE;*

*SELECT CURRENT\_TIME;*

**-- CASTING allows you to convert one date type to another such that columns stored in your data base can be retrieved and outputted as a different time.**

**-- The generic sintax is the CAST() function:**

*SELECT CAST(NOW() AS timestamp);*

*SELECT CAST(NOW() AS date);*

SELECT

    CURRENT\_TIMESTAMP(0)::timestamp AS right\_now,

    interval '5 days' + CURRENT\_TIMESTAMP(0) AS five\_days\_from\_now;

**-- Extract a date from a timestamp, or part of a timestamp**

**-- - EXTRACT(field FROM source)**

*SELECT EXTRACT(quarter FROM timestamp ‘2005-01-24 05:12:00’) AS quarter;*

-- You could use year, month, dow (day of week), etc. as field.

**-- This is identical to**

*SELECT DATE\_PART(‘quarter’, timestamp ‘2005-01-24 05:12:00’) AS quarter;*

**-- Example:** get the total payment by quarter of each year

*SELECT*

*EXTRACT(quarter FROM payment\_date) AS quarter,*

*EXTRACT(year FROM payment\_Date) AS year,*

*SUM(amount) AS total\_paments*

*FROM payment*

*GROUP BY 1, 2;*

**-- DATE\_TRUNC(): truncate a timestamp**

*SELECT DATE\_TRUNC(‘year’, ‘2005-01-24 05:12:00’)*

-- Returns *2005-01-01 00:00:00*

**-- Exercise:** In this exercise, you are going to extract a list of customers and their rental history over 90 days. You will be using the EXTRACT(), DATE\_TRUNC(), and AGE() functions that you learned about during this chapter along with some general SQL skills from the prerequisites to extract a data set that could be used to determine what day of the week customers are most likely to rent a DVD and the likelihood that they will return the DVD late.

SELECT

  c.first\_name || ' ' || c.last\_name AS customer\_name,

  f.title,

  r.rental\_date,

  -- Extract the day of week date part from the rental\_date

  EXTRACT(dow FROM r.rental\_date) AS dayofweek,

  AGE(r.return\_date, r.rental\_date) AS rental\_days,

  -- Use DATE\_TRUNC to get days from the AGE function

  CASE WHEN DATE\_TRUNC('day', AGE(r.return\_date, r.rental\_date)) >

  -- Calculate number of d

    f.rental\_duration \* INTERVAL '1' day

  THEN TRUE

  ELSE FALSE END AS past\_due

FROM

  film AS f

  INNER JOIN inventory AS i

    ON f.film\_id = i.film\_id

  INNER JOIN rental AS r

    ON i.inventory\_id = r.inventory\_id

  INNER JOIN customer AS c

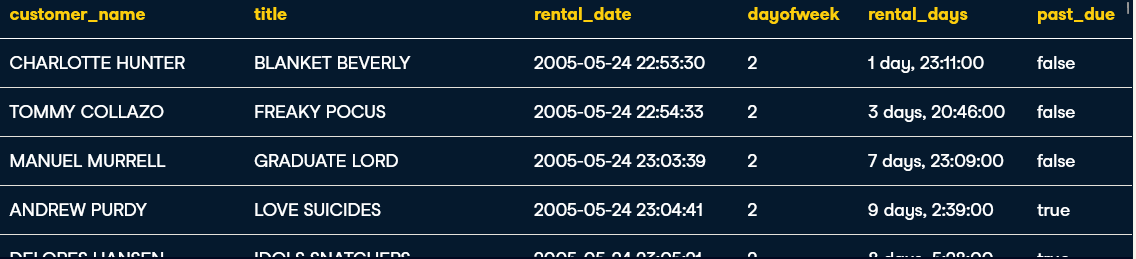
    ON c.customer\_id = r.customer\_id

WHERE

  -- Use an INTERVAL for the upper bound of the rental\_date

  r.rental\_date BETWEEN CAST('2005-05-01' AS DATE)

  AND CAST('2005-05-01' AS DATE) + INTERVAL '90 day';

****

**Ch3: Parsing and Manipulating Text**

**-- Concatenate two strings:**

*SELECT*

*first\_name, last\_name,*

*first\_name || ‘ ‘ || last\_name AS full\_name*

*FROM customer;*

**-- Another example (using a non-string column)**

*SELECT*

*customer id || ‘ : ‘ || first\_name || ‘ ‘ || last\_name AS full\_name*

*FROM customer;*

**-- Or using PostgreSQL’s concat:**

*SELECT*

*CONCAT(first\_name, ‘ ‘, last\_name) AS full\_name*

*FROM customers;*

**-- Change string to upper case or lower case**

*SELECT*

*UPPER(email),*

*LOWER(email)*

*FROM customer;*

**-- Change string to ‘title’ case (upper case first letters)**

*SELECT INITCAP(title)*

*FROM film;*

**-- Replace a character inside a string**

*SELECT*

*REPLACE(column1, ‘old string to replace’, ‘new string’)*

*FROM table1;*

**-- Reverse a string / return a string in reversed order**

*SELECT REVERSE(col1) FROM table1;*

**-- Determine the number of characters in a string**

*SELECT CHAR\_LENGTH(title) FROM film;* **-- equivalent to LENGTH(title)**

**-- Find the position of a character inside a string**

*SELECT*

*email,*

*POSITION(‘@’ IN email)* **– Equivalent to STRPOS(email, ‘@’)**

*FROM customer;*

**-- Extract the first N characters of a string**

*SELECT LEFT(‘Hello’, 2);* -- returns ‘He’

**-- Extract the last N characters of a string**

*SELECT RIGHT(‘Hello’, 2);* -- returns ‘lo’

**-- SUBSTIRNG: extract a substring from text data**

*SELECT*

*SUBSTRING(col1, 10, 15)* – from position 10 to position 15

*FROM table1;*

**-- Example: extract the left part of an email before the arroba**

*SELECT*

*SUBSTRING(email FROM 0 FOR POSITION(‘@’ IN email))*

*FROM customer;*

**-- Example: extract the right part of an email before the arroba**

*SELECT*

*SUBSTRING(email FROM POSITION(‘@’ IN email+1 FOR CHAR\_LENGTH(email))*

*FROM customer;*

**-- TRIM(): remove characters from either the start, end or both from a string. SINTAX:**

**

-- The first parameter is optional, and specifies whether you want to remove characetrs from the beginning/end/both of a string(default is both)

-- Second parameter (also optional), specifies the characters to be removed from the string. The default is a blank space.

-- Third parameter: the string

**-- Append a character or string to another string by a specified number of characters.**

**-- It is useful when you want a minimum number of characters in a column, so you fill the missing length with a specific character (default is a whitespace)**

*SELECT LPAD(‘padded’, 10, ‘#’);* -- Returns ‘###padded’

*SELECT RPAD(‘padded’, 10, ‘#’);* -- Returns ‘padded###’

**-- Exercise:** truncate text fields like the film table's description column without cutting off a word.

SELECT

  f.title,

  -- Truncate the description without cutting off a word

  LEFT(description, 50 -

    -- Subtract the position of the first whitespace character

    POSITION(

      ' ' IN REVERSE(LEFT(description, 50))

    )

  )

FROM

  film AS f

  INNER JOIN film\_category AS fc

    ON f.film\_id = fc.film\_id

  INNER JOIN category AS c

    ON fc.category\_id = c.category\_id;

**Ch4: Full-text Search and PostgreSQL Extensions**

**-- The LIKE operator can be used in the WHERE clause to search for a pattern in a column. To accomplish this, you use a WILDCARD. There are two types of wildcards:**

**-- ‘\_’ wildcard: used to match exactly one character**

**-- ‘%’ wildcard: used to match zero or more characters**

*SELECT title FROM film WHERE title LIKE ‘Elf%’;* – Returns ‘ELF PARTY’

*SELECT title FROM film WHERE title LIKE ‘%Elf’;* – Returns ‘ENCINO ELF’

**-- The LIKE operator is CASE SENSITIVE**

**-- FULL\_TEXT SEARCH provides a means for performing natural language queries of text data in your database, that uses:**

**-- - STEMMING**

**-- - FUZZY-STRING MATCHING to handle spelling mistakes**

**-- - A mechanism to RANK results by similarity to the searched string**

**-- FULL-TEXT SERACH is not case sensitive, it accounts for variations of the search string.**

*SELECT title, description FROM film*

*WHERE to\_tsvector(title) @@ to\_tsquery(‘elf’);* -- Returns all movies with ‘elf’ in their name, regardless of cases

**-- to\_tsvector() convert text and string data to a ts\_vector data type, which is a sorted list of words that have been normalized into variants of the same word, called ‘lexims’**

**Graphical user interface, text

Description automatically generated**

**-- EXTENSIONS FOR SQL: create user-defined data types**

**-- ENUMERATED DATA TYPES (‘ENUMs) allow you to define a custom list of values that are never going to change, like the days of the week:**

*CREATE TYPE dayofweek AS ENUM(*

*‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’, ‘Friday’, ‘Saturday’, ‘Sunday’*

*);*

**-- Once the datatype is created, you can query the system table to get information about it**

*SELECT typname, typcategory*

*FROM pg\_type*

*WHERE typname = ‘dayofweek’;*

**-- Retrieve column data with their respective user-defined data type**

*SELECT column\_name, data\_type, udt\_name*

*FROM INFORMATION\_SCHEMA.COLUMNS*

*WHERE table\_name = ‘table1’;*

**-- USER-DEFINED FUNCTIONS: create functions in PostgreSQL**

*CREATE FUNCTION squared(i, integer) RETURNS integer AS $$*

*BEGIN*

*RETURN i \* i*

*END;*

*$$ LANGUAGE plpgsql*

**-- The previous function accepts an integer ‘i’ as an input parameter and returns the square of that parameter as a result. The ‘$$’ syntax specifies specifies that the function will be using SQL as the language.**

**-- PostgreSQL extension framework. Commonly used extensions:**

**-- - PostGIS: location queries**

**-- - PostPIC: image processing**

**-- - fuzzystrmatch and pg\_trgm: extend full-search capabilities by finding similarities between strings**

**-- View AVAILABLE EXTENSIONS**

*SELECT name FROM pg\_available\_extensions;*

**-- View INSTALLED EXTENSIONS/ CURRENTLY AVAILABLE EXTENSIONS**

*SELECT extname FROM pg\_extension;*

**-- LOAD EXTENSIONS into your database:**

*CREATE EXTENSION IF NOT EXISTS fuzzystrmatch;*

**-- Confirm that the extension was enables**

*SELECT extname FROM pg\_extension;*

**-- Use the LEVENSHTEIN() function from fuzzystrmatch to calculate the Levenshtein distance between two strings (the number of edits required for the strings to be a perfect match)**

*SELECT levenshtein(‘GUMBO’, ‘GAMBOL’);* -- Returns 2 (switch ‘U’ with ‘A’, and add ‘L’)

**-- Determine the SIMILARITY BETWEEN TWO STRINGS using “Try Gram matching”. Try grams are groups of three consecutive characters in a string.**

*SELECT similarity(‘GUMBO’, ‘GAMBOL’);* -- 1 is perfect match