**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;*

**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**

**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*

**-- 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;*