DATA SCIENCE RETREAT

SQL

A crash course

Batch 13
January 21st 2018
Berlin, Germany

Agenda

- **Introduction**
- **SQL Concepts and Basic Operations**
- **More Operations: Joins, Aggregations, Subqueries**
- **SQL Functions: Window, Analytic**
- **Tricks and Tips**

Introduction

What is SQL?

Why SQL?

The RDBMS Landscape

NoSQL?

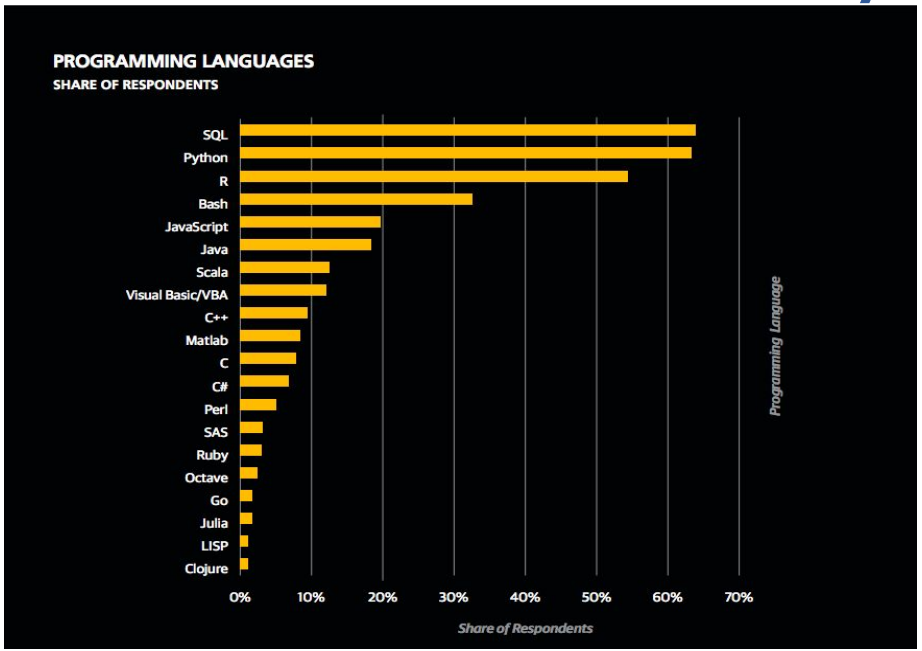
What is SQL?

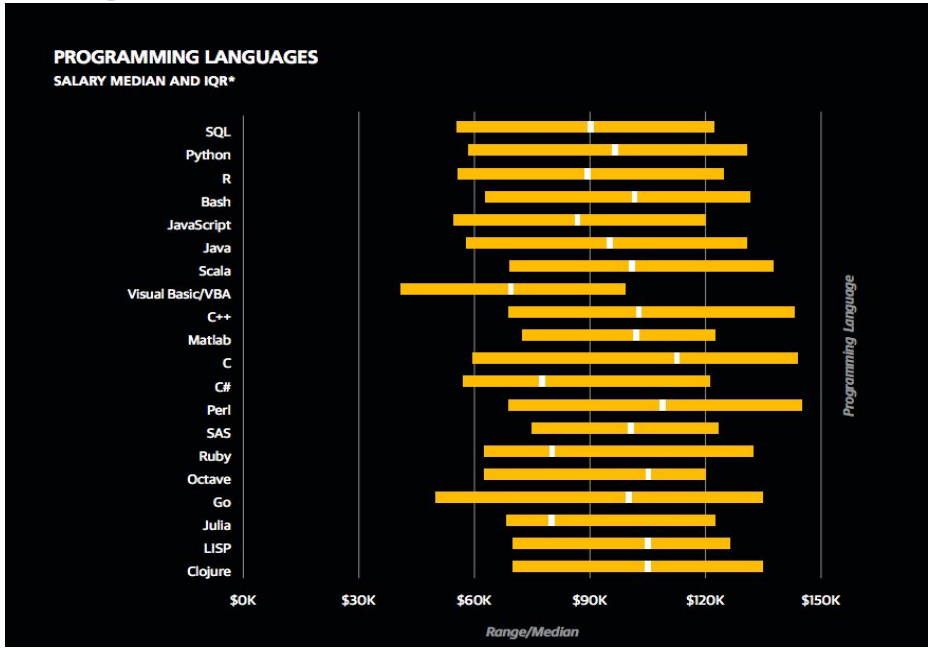
- SQL = Structured Query Language
- An application of the *relational model* (1970)
- Used with commercial databases since 1979
- A standard: ANSI (1986) and ISO (1987)

Why SQL?

- Different domains:
  - Transactional applications such as e-commerce
  - Data warehousing
  - Reporting and Analytics
- Different roles:
  - Software/Data engineers
  - Business analysts
  - Data scientists

Why SQL?





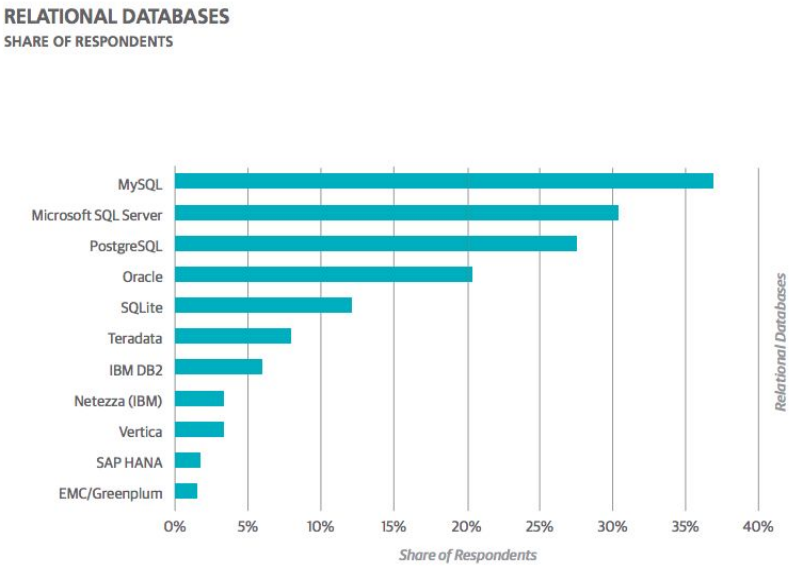
It is a **must-have skill** when you're working towards getting a job in the data science industry.

Source: [2017 O'Reilly Data Science Salary Survey]

The RDBMS Landscape

- RDBMS = Relational Database Management System
- Some relational DBs:
  - SQLite
  - Oracle
  - SQL Server
  - IBM DB2
  - MySQL/MariaDB
  - PostgreSQL
  - Amazon RDS
  - Amazon Redshift
  - SAP Hana
  - Hive
  - SparkSQL

The RDBMS Landscape



Source: [2017 O'Reilly Data Science Salary Survey]

NoSQL?

- Motivation: better scaling
- Specific databases:
  - Elasticsearch (document-oriented)
  - Cassandra
  - MongoDB
  - Neo4J (graph database)

SQL Basics

Concepts

Database Structures

Database Objects

Data Types

Types of SQL Statements

The Basic SQL Query

Concepts

- CAP Theorem: in distributed systems, you can provide two of the three:
  - **Consistency**: Every read receives the most recent write or an error
  - **Availability**: Every request receives a (non-error) response – without guarantee that it contains the most recent write
  - **Partition tolerance**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes
- ACID:
  - **Atomicity**: indivisible (all or nothing) transactions
  - **Consistency**: you always get to either before or after the transaction
  - **Isolation**: each transaction sees things as they were before until it is finished
  - **Durability**: if your transaction finished, it is safe

Database Structures (1/2)

- Storage
  - HDs, SSDs, memory
- Files
  - or memory mapped also possible.
- Tablespace
  - Logical allocation of space in files for database objects.
- Security
  - Users, roles, privileges.

Database Structures (2/2)

- Database
  - Essentially a larger collection of related data (i.e., for an application).
- Schema
  - Logical organization of tables by user or subject within a database.
- Database Objects
  - Tables, views, indexes, keys, functions and more.

Database Objects

- **Table**: the basic data storage type
- **View**: an alias for a select statement
- **Index**: (sometimes) accelerates searches
  - Don’t index everything: index cost performance and space.
  - Some databases offer special index types (like bitmaps).
- **Key**: column(s) used as a unique identifier for rows in the table
- And some more: trigger, function/procedure, sequence, partition, cluster, database link (not in Postgres)

Data Types (1/2)

- NULL
- Numbers (link is for Postgres only)
  - Integers and floating point
  - Numeric (arbitrary precision, decimal exact and slow!)
  - Money
- Text
  - char(n), varchar(n), text (check your database for encoding support and configuration)
  - bytea (raw byte strings, can store blobs – but not really large objects)
- Date/Time
  - date, time, timestamp, interval

Data Types (2/2)

- Boolean
  - TRUE/FALSE
- Enumerated
  - Similar to ordered factors, have to be defined by CREATE TYPE xxx AS ENUM
  - Careful, converting to integer is not easy
- Text Search
  - More on that later
- Others
  - Geometric, network addresses, bit strings, UUID, XML, JSON, arrays, composite

Types of SQL Statements

- Data Definition Language (**DDL**) – manipulate DB objects (tables, etc)
  - CREATE
  - ALTER
  - DROP
- Data Manipulation Language (**DML**) – manipulate the data
  - SELECT
  - UPDATE
  - INSERT
  - DELETE
- Data Control Language (**DCL**) – access control
  - GRANT
  - REVOKE

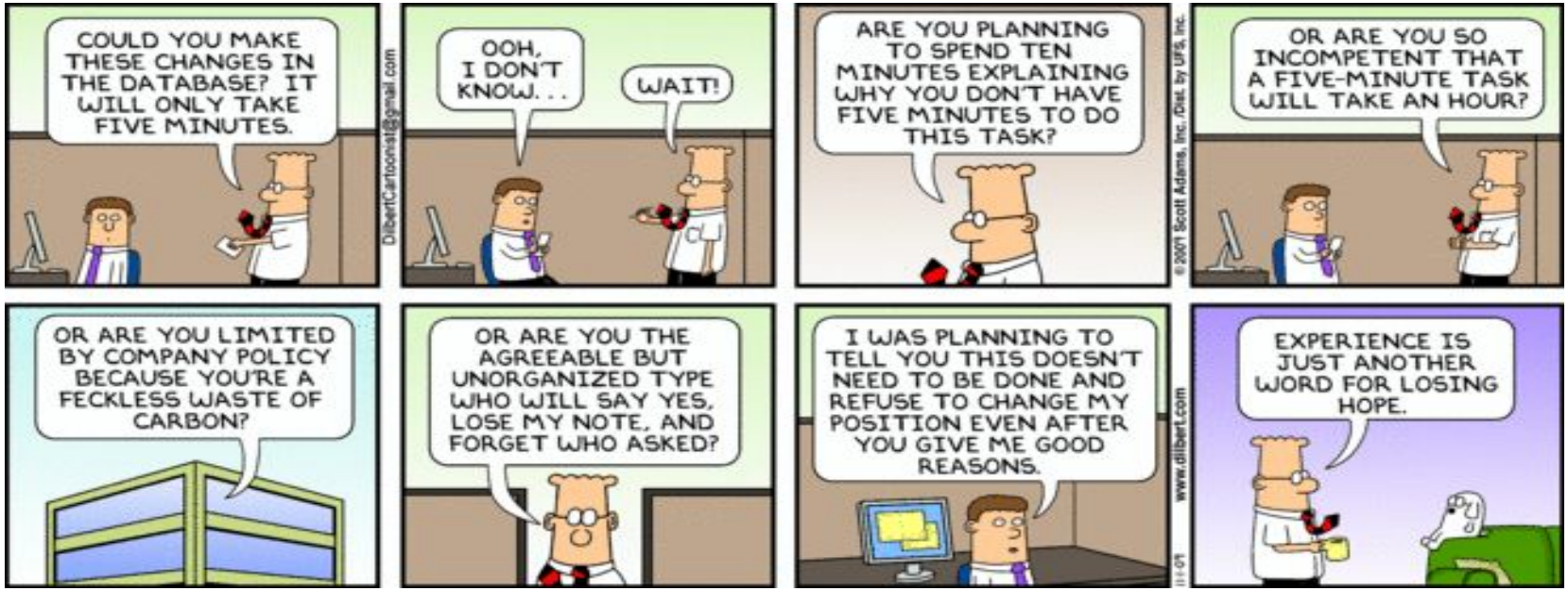
DDL Example - Create

```
CREATE TABLE film (
    film_id integer DEFAULT nextval('film_film_id_seq'::regclass) NOT NULL,
    title character varying(255) NOT NULL,
    description text,
    release_year year,
    language_id smallint NOT NULL,
    rental_duration smallint DEFAULT 3 NOT NULL,
    rental_rate numeric(4,2) DEFAULT 4.99 NOT NULL,
    length smallint,
    replacement_cost numeric(5,2) DEFAULT 19.99 NOT NULL,
    rating mpaa_rating DEFAULT 'G'::mpaa_rating,
    last_update timestamp without time zone DEFAULT now() NOT NULL,
    special_features text[],
    fulltext tsvector NOT NULL
);
```

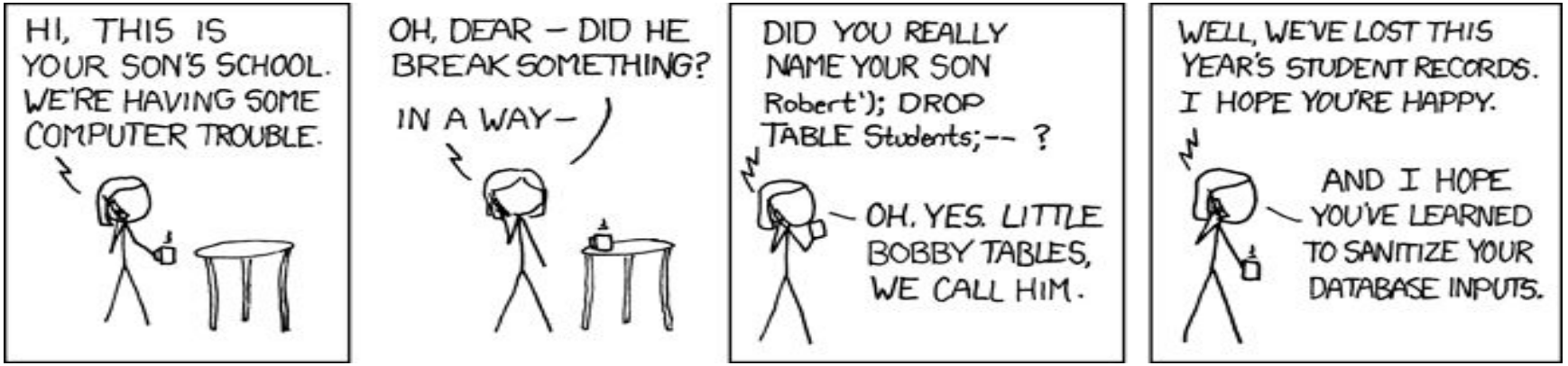
DDL Examples - Create, Alter

- Create a table named "**user**" with columns "first_name" and "last_name"
  - CREATE TABLE user (first_name varchar(20), last_name varchar(20));
- Add a new column "birthdate" to "**user**" table:
  - ALTER TABLE user ADD COLUMN birthdate date;

You'll probably have to go through a DBA for DDL privileges



Little Bobby Tables revisited



DML Examples - Insert, Update, Select

- Insert data into the "**user**" table:
  - INSERT INTO user (first_name, last_name, birthdate) VALUES ('Freddie', 'Flintstone', '1960-09-30');
- Change the first_name of user Freddie to Fred:
  - UPDATE user SET first_name = 'Fred' WHERE first_name='Freddie'
- Check content of "**user**" table:
  - SELECT * FROM user

The Basic SQL Query

```
SELECT  <attributes>
FROM    <one or more relations>
WHERE   <conditions>
```

Inside the SELECT Statement

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]

where from_item can be one of:

    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...] | column_definition [, ...] ) ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
    from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ]

and with_query is:

    with_query_name [ ( column_name [, ...] ) ] AS ( select | values | insert | update | delete )

TABLE [ ONLY ] table_name [ * ]
```

The WHERE Clause

- Select only a subset of rows of the table.
- Operators:

Operator	Description
=	Equal
<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

- More about Operators: https://www.w3schools.com/sql/sql_and_or.asp

Logic question

```
SELECT *
FROM   Person
WHERE  age < 25  OR  age >= 25
```

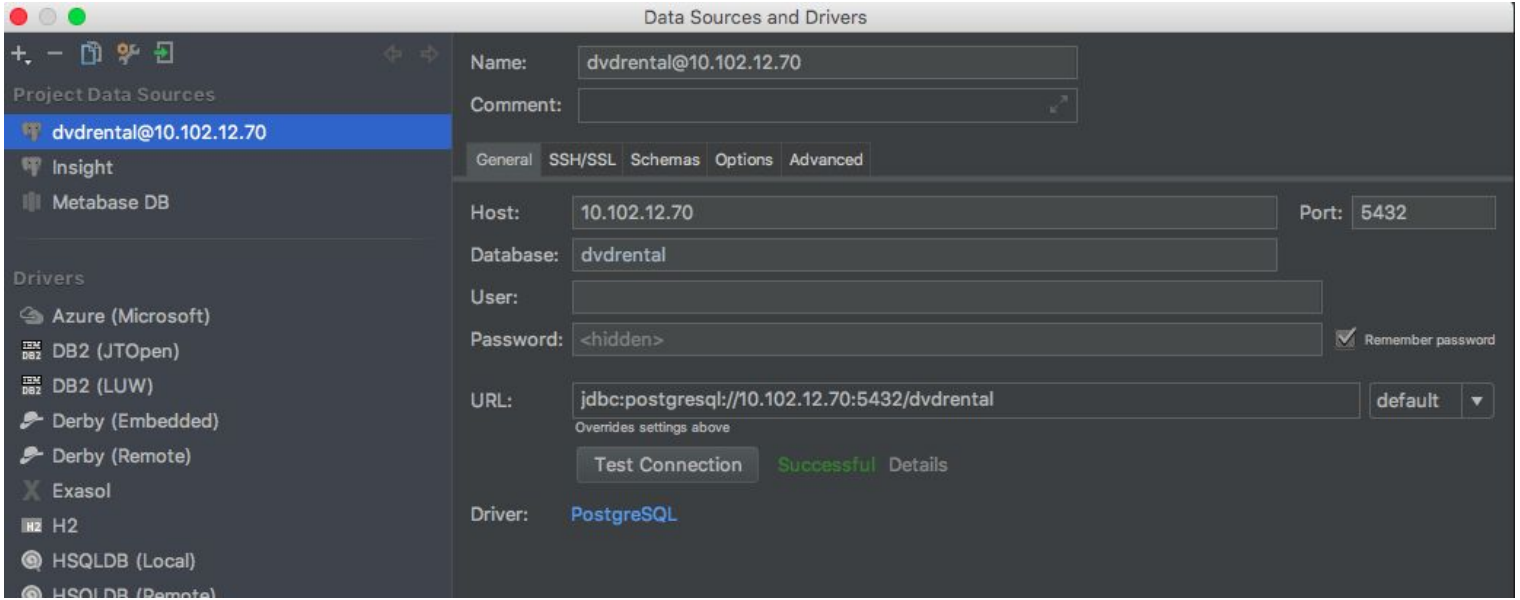
Can it be that some Persons are not included?

WARNING

- SQL has a 3-state boolean logic system. WHERE only accepts rows where the condition is TRUE (i.e. not FALSE, but also not UNKNOWN).

Enough Theory, let's try it!

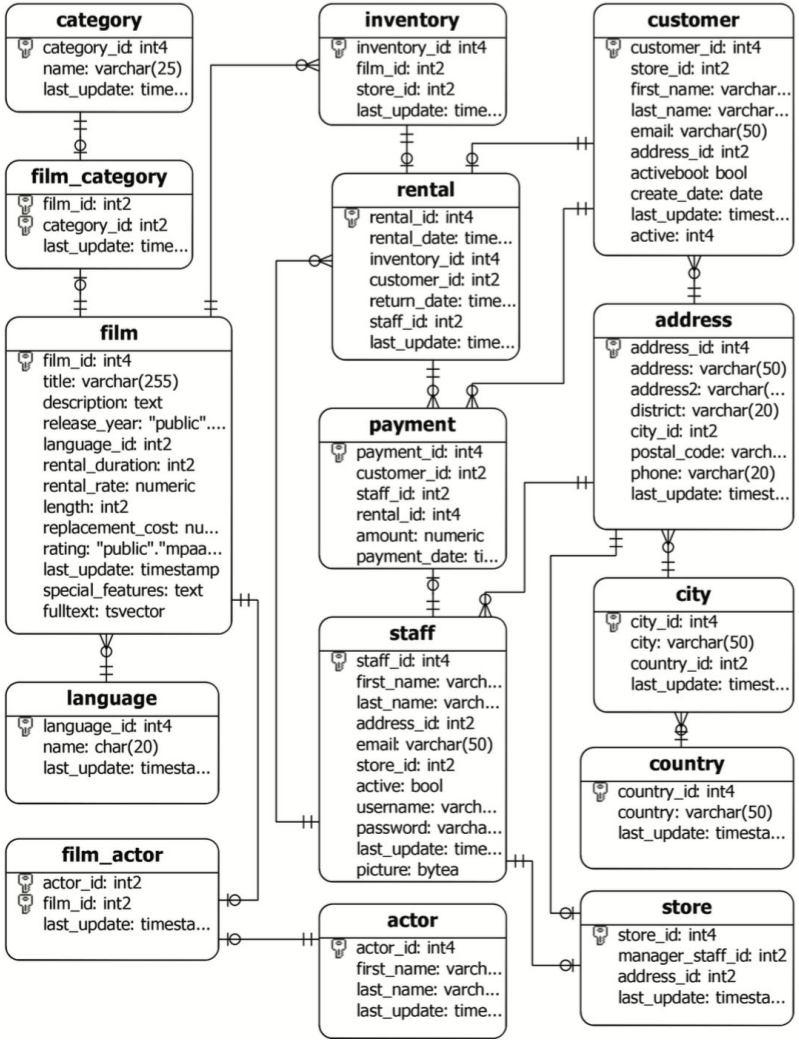
DataGrip



Download at https://www.jetbrains.com/datagrip/

DB Schema

dvdrental



Example

- List all films with their title and language

```
SELECT
    film.title,
    language.name
FROM
    film
    JOIN language
    ON film.language_id = language.language_id
```

Operations

Join

Union

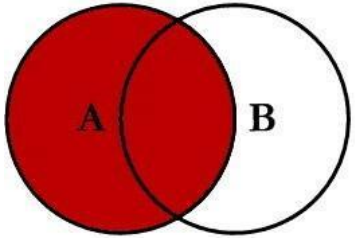
Subqueries

The WITH Clause

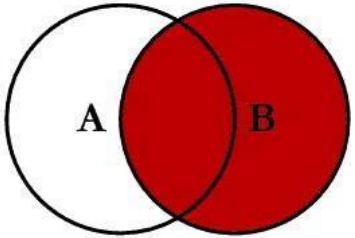
SQL Operations Overview

- One Table Basics
  - Projection: selecting columns
  - Selection: selecting rows
- To see more columns: Join tables
  - Inner Join
  - Outer Join (left, right, full)
  - Cross Join: full cartesian product
  - Semijoin and antijoin: filtering (can also be done with outer joins)
- To see more rows: Union tables
  - Union / Union Distinct / Union Al

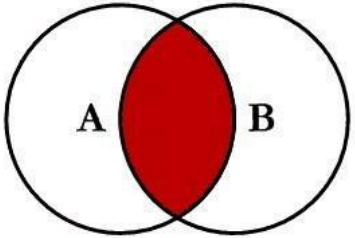
SQL JOINS



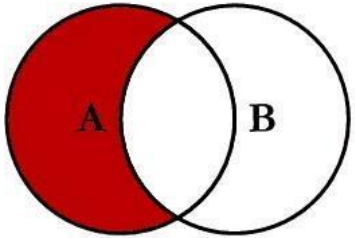

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



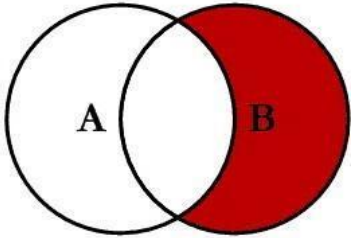

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



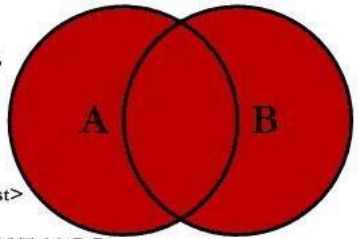

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



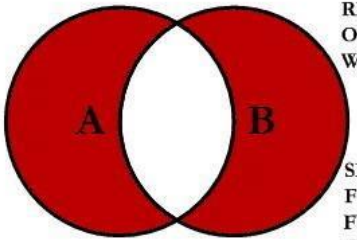

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Join Exercises

1. List all film titles with their actors' names.

2. List titles of films that are not in the inventory.

3. List *distinct* titles of all films returned on 2005-05-27
   a. I haven't showed you how to work with dates; there are many ways to deal with this – can you find one?
   b. *Distinct* titles because maybe the same title was returned by different users.

Union

- Combine the results of two or more SELECT statements.
- Each SELECT statement must have the same number of columns and the columns must:
  - have similar data types
  - be in the same order
- The UNION operator selects only distinct values by default.
- UNION ALL: to allow duplicate values.

Union Syntax

```
SELECT column_name(s) FROM table1
UNION (ALL)
SELECT column_name(s) FROM table2;
```

- Examples: https://www.w3schools.com/sql/sql_union.asp

Subqueries

- IN / NOT IN
  - (column_list) IN (list)
  - Can be a list of values
  - Can be another select
  - You can actually check for tuples, like (first_name, last_name) in (select first_name, last_name from...)
- EXISTS
  - Will be true if at least one comparison satisfies the condition
  - Good for checking if something is on a list (correlated subquery)
- ANY
  - Can check for more than IN. Any comparison operator goes
  - IN is the same as =ANY
- ALL
  - Same as ANY, but will be true if the condition holds for all cases.
  - NOT IN is the same as <> ALL

Subquery Example

```
SELECT
    SUM(Sales)
FROM
    Store_Information
WHERE
    Store_Name IN
    (SELECT Store_Name
    FROM Geography
    WHERE Region_Name = ‘West');
```

Subquery Exercises

1. names of all customers who returned a rental on 2005-05-27

2. names of customers who have made a payment
   a. with a subquery
   b. with a JOIN

The WITH Clause

- Create temporary tables: available during query execution time only.
- Example: List all rented movies titles with the customer names

```
WITH rentals AS (
     SELECT c.first_name, c.last_name, r.rental_id, i.film_id
     FROM customer c, rental r, inventory i
     WHERE c.customer_id = r.customer_id AND r.inventory_id = i.inventory_id
)
SELECT f.title, r.first_name, r.last_name
FROM film f, rentals r
WHERE f.film_id = r.film_id
ORDER BY title
```

- Exercises:
  1. Re-do Subquery exercise 1) using WITH.

Functions

Aggregate Functions

Window Functions

Date Functions

String Functions

Pattern Matching

Sampling

SQL Functions Overview (1/2)

- Aggregate/Statistics
  - Compute a single result from a set of input values.
  - Very useful, especially when combined with GROUP BY/HAVING.
  - Most commonly used: COUNT, SUM, AVG, MIN, MAX, STRING_AGG.
- Window Functions
  - Perform calculations across sets of rows that are related to the current row.
  - Most commonly used: RANK, ROW_NUMBER, NTILE.
- Date/time Functions
  - Manipulating and handling dates and timestamps.
  - Most commonly used: CURRENT_DATE, DATE_TRUNC, DATE_PART.

SQL Functions Overview (2/2)

- String Functions
  - Manipulate values of types: character, character varying and text: CONCAT, LOWER, UPPER, TRIM, SUBSTRING.
  - Pattern matching: LIKE and regular expressions.
  - Conversions/formatting: TO_CHAR, TO_DATE, TO_NUMBER.
- Others:
  - Pivoting;
  - Sampling;
  - Conditional: CASE WHEN, COALESCE;
  - Mathematical: ROUND, CEILING, FLOOR, LOG, SQRT, POWER;
  - Sequence manipulation.

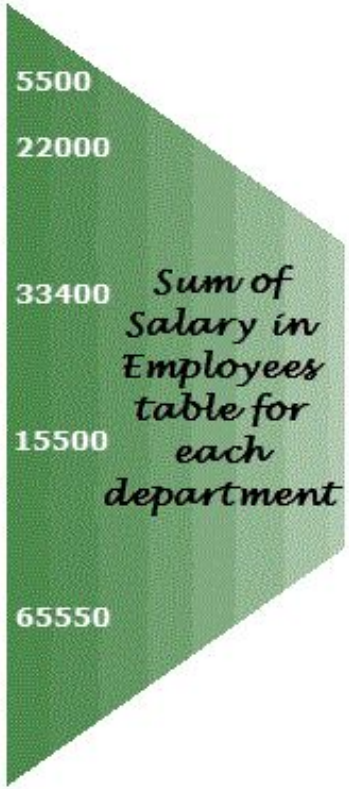
Aggregate Functions

- Compute a single result from a set of input values.
  - Perform calculation:
    - COUNT()
    - SUM()
    - MIN()
    - MAX()
    - AVG()
  - Over all rows, or per group:
    - GROUP BY/HAVING

Aggregate Example

Employees

DEPARTMENT_ID	SALARY
10	5500
20	15000
20	7000
30	12000
30	5100
30	4900
30	5800
30	5600
40	7500
40	8000
50	9000
50	8500
50	9500
50	8500
50	10500
50	10000
50	9500



DEPARTMENT_ID	SUM(SALARY)
10	5500
20	22000
30	33400
40	15500
50	65550

Aggregate Exercises

1. customers ordered by how much they've spent
2. customers who have spent more than 200
3. stores with more than 200 customers
4. the number of rentals from each category

EXTRA: films whose rental_rate is higher than the average rental_rate

Window Functions

- Perform calculations across sets of rows that are related to the current row.
  - ROW_NUMBER *OVER (PARTITION BY … ORDER BY ...)*
    ⟹ Unique number to each row within its partition, counting from 1.
  - RANK *OVER (PARTITION BY … ORDER BY ...)*
    ⟹ Rank of current each row within its partition, with gaps.
  - DENSE_RANK *OVER (PARTITION BY … ORDER BY ...)*
    ⟹ Rank of current each row within its partition, without gaps.
  - NTILE(num_buckets) *OVER (PARTITION BY … ORDER BY ...)*
    ⟹ Distributes the rows in buckets of equal size, that is, percentiles (quartile = 4, decile = 10, …)
- In blue the optional arguments, to apply in a group and a specific sort ordering.

Window Example

```
SELECT
  payment.customer_id,
  customer.first_name,
  customer.last_name,
  payment_date,
  row_number() OVER (ORDER BY payment_date DESC ),
  rank() OVER (ORDER BY payment_date DESC ),
  dense_rank() OVER (ORDER BY payment_date DESC )
FROM
  payment
  JOIN customer ON payment.customer_id = customer.customer_id
ORDER BY payment_date DESC
```

- Exercises:
  1. Find the most rented film in each category.
  2. Find the most recent returned movie title + customer name + date.
  3. Find the 10% most profitable customers (top 10%).

Date Functions

- CURRENT_DATE
- DATE_TRUNC(field, timestamp_column)
- DATE_PART(field, timestamp_column)
  - Allowed field values are: microseconds, milliseconds, second, minute, hour, day, week, month, quarter, year, decade, century, millennium
- Example:

```
SELECT
  DATE_PART('year', rental_date) year_of_rental,
  COUNT(customer_id) customers
FROM rental
GROUP BY 1
```

String Functions

- LOWER(string), UPPER(string)
- CONCAT(string_1, string_2, …, string_n)
  - ```
    SELECT
      CONCAT(first_name, ' ', last_name) AS full_name
    FROM customer
    ```
- TRIM([leading | trailing | both] [characters] from string)
  - TRIM(both 'x' from 'xTomxx') => Tom
- SPLIT_PART(string, delimiter, field)
  - SPLIT_PART('dania@gmail.com', '@', 1) => 'dania'
- Pattern Matching: LIKE
  - string (NOT) LIKE pattern
    - An underscore (_) matches any single character.
    - A percent sign (%) matches any sequence of zero or more characters.

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```
```

String Functions

- Regular expression functions, considering the example string:
  `'http://www.example.com/?utm_source=facebook&utm_medium=social&utm_campaign=black-friday'`
  - SUBSTRING(string from pattern) - extract substring.
    - SUBSTRING(`example_str`, 'utm_campaign=(.*)$') => 'black-friday'
  - REGEXP_MATCHES(source, pattern, replacement [, flags]) - extract pattern.
    - REGEXP_MATCHES(`example_str`,'facebok') => '{facebook}'
  - REGEXP_REPLACE(source, pattern, replacement [, flags]) - replace pattern.
    - REGEXP_REPLACE(`example_str`,'^http://(.*).com', '') => '/?utm_source=facebook&utm_medium=social&utm_campaign=black-friday'
  - regexp_split_to_table(subject, pattern[, flags]) - returns the split string as a new table.
  - regexp_split_to_array(subject, pattern[, flags]) - returns the split string as an array of text.

Other functions

- Pivoting and reshaping
  - mySQL Example (does not work for PostgreSQL...)
- Sampling
  - `SELECT ... ORDER BY random() LIMIT sample_size`
- Generating sequences on the fly
  - Use generate_series() to create a list of dates as a subquery, then outer join to your data and you get evenly distributed observations from sparse actual cases.
    - In this case, you will have to impute missing values.
- Conditional: CASE WHEN
  - The same as IF/ELSE statement in other programing languages.
- Conditional: COALESCE
  - Returns the first non-null argument. You can use it to substitute NULL by a default value.

Tips & Tricks

Database Index & When indexes don't matter

Optimization Examples & Tips

Connecting to Python

User Segmentation with SQL

Database Index

- An index is a data structure that improves the speed of the data retrieval in your database table.
- Indexes can be created by using one or more columns in a database table.
- Pro: allows for quick look up without having to search every row in a database every time the database table is accessed.
- It comes at a cost: there will be additional writes and additional storage space is needed to maintain the index data structure.

When Indexes DON'T Matter

- HAVING Clause: Prevents the database from using any existing index.
  - Alternative: the WHERE clause
    - WHERE clause introduces a condition on individual rows
    - HAVING clause introduces a condition on aggregations or results
    - This is not about limiting the result set, rather about limiting the intermediate number of records within a query.
- The OR Operator
  - Alternative: replace it by a condition with IN
- The NOT Operator
  - Alternative: replacing NOT by comparison operators, such as >, <> or !>

When Indexes DON'T Matter

- The ANY and ALL Operators
  - Alternatives: aggregation functions like MIN or MAX.
    - Be aware of the fact that all aggregation functions like SUM, AVG, MIN, MAX over many rows can result in a long-running query.
    - In such cases, you can try to either minimize the amount of rows to handle or pre-calculate these values.
- Column is used in a calculation or function
  - Alternative: isolate the specific column so that it no longer is a part of the calculation/function.
    - Instead of: WHERE year + 10 = 1980;
      Write: WHERE year = 1970;

Optimization

- Garbage In, Garbage Out (GIGO) principle:
  - The one who formulates the query also holds the keys to the performance of your SQL queries.
- Common performance issues occur on:
  - The WHERE clause
  - Any INNER JOIN or LEFT JOIN
  - The HAVING clause

Based on:

http://www.kdnuggets.com/2017/08/write-better-sql-queries-definitive-guide-part-1.html

http://www.kdnuggets.com/2017/08/write-better-sql-queries-definitive-guide-part-2.html

Bad Performance Example: JOIN

```
SELECT
      employees.employee_number,
      employees.name
FROM
      employees
      INNER JOIN
      (SELECT
            department,
            AVG(salary) AS department_average
      FROM employees
      GROUP BY department) AS temp
      ON employees.department = temp.department
WHERE
      employees.salary > temp.department_average;
```

- A **correlated subquery** is a subquery that uses values from the outer query.
- Having a correlated subquery isn't always a good idea.

Bad Performance Example: WHERE

```
SELECT
     employee_number,
     name
FROM
     employees AS emp
WHERE
     salary > (SELECT AVG(salary)
               FROM employees
               WHERE department = emp.department);
```

- This subquery is not correlated with the outer query, and is therefore executed only once, regardless of the number of employees.

BEST Performance Example

```
WITH temp AS (
     SELECT
          department,
          AVG(salary) AS department_average
     FROM employees
     GROUP BY department
)
SELECT
     employees.employee_number,
     employees.name
FROM employees
     INNER JOIN temp ON employees.department = temp.department
WHERE employees.salary > temp.department_average;
```

- A **correlated subquery** is a subquery that uses values from the outer query.
- Having a correlated subquery isn't always a good idea.

Optimization Tip #1

- Only Retrieve The Data You Need
  - SELECT statement: remove unnecessary columns.
  - DISTINCT clause: try to avoid if you can.
    - It is used to return only different values and the execution time only increases if you add this clause to your query.
  - LIKE operator: this type of query potentially leaves the door open to retrieve too many records that don't necessarily satisfy your query goal.
    - It also prevents the database from using an index (if it exists), when the pattern starts with % or _.

Optimization Tip #2

- Limit Your Results
  - TOP and LIMIT Clauses: to set a **maximum number of rows** for the result set.
    - Note that you can further specify the PERCENT, for example, if you change the first line of the query by SELECT TOP 50 PERCENT *.
  - ROWNUM Clause: equivalent to using LIMIT in your query.

Optimization Tip #3

- Data Type Conversions
  - You should always use the most efficient, that is, **smallest**, data types possible.
  - But, **avoid data type conversion** as much as possible: when you add data type conversion to your query, you only increase the execution time.
  - It’s not always possible to remove or omit the data type conversion from your queries, but you should definitely aim to be careful in including them.

Optimization Tip #4

- Don’t Make Queries More Complex Than They Need To Be
  - The UNION Operator: when you use a UNION in your query, the execution time will increase.
    - Alternative 1: reformulating the query in such a way that all conditions are placed in one SELECT instruction,
    - Alternative 2: OUTER JOIN.

Optimization Tip #5

- No Brute Force
  - The order of tables on joins
    - If you notice that one table is considerably larger than the other one, you might want to rewrite your query so that the biggest table is placed last in the join.
  - Redundant conditions on joins
    - When you add too many conditions to your joins, you basically obligate SQL to choose a certain path. It could be though, that this path isn't always the more performant one.

Connecting to Python

```
import psycopg2
import psycopg2.extras


def ResultIter(cursor, arraysize=1000):
    'An iterator that uses fetchmany to keep memory usage down'
    while True:
        results = cursor.fetchmany(arraysize)
        if not results:
            break
        for result in results:
            yield result


conn = psycopg2.connect("dbname=dvdrental user=postgres host=192.168.111.128")
cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
cur.execute("select * from film")
for result in ResultIter(cur):
    print(result)
```

User Segmentation with SQL

- RFV Segmentation
  - R: recency - the last transaction
  - F: frequency - how many transactions
  - V: value - the total value of the transactions
  - Optional: for a determined period
    - e.g. year, quarter, month

User Segmentation with SQL

```
SELECT
   customer.first_name,
   customer.last_name,
   max(rental.rental_date)                    AS last_rental_date,
   count(rental.rental_id)                    AS total_transactions,
   sum(payment.amount)                        AS total_amount,
   NTILE(2) OVER (ORDER BY max(rental.rental_date) DESC ) AS median_r,
   NTILE(2) OVER (ORDER BY count(rental.rental_id) DESC ) AS median_f,
   NTILE(2) OVER (ORDER BY sum(payment.amount) DESC )     AS median_v
 FROM
   rental
   JOIN payment ON rental.rental_id = payment.rental_id
   JOIN customer ON payment.customer_id = customer.customer_id
 WHERE date_part('year',rental.rental_date) = 2005
 GROUP BY 1, 2
 ORDER BY 3 DESC, 4 DESC, 5 DESC
```

Questions ?

Thank you!

Dania Meira
meira.dania@gmail.com
https://www.linkedin.com/in/daniameira/