DATA SCIENCE RETREAT

SQL

A crash course

https://github.com/meiradania/data-science-retreat-sql/

Batch 20

September 26th 2019

Berlin, Germany

About Me

- Data Scientist in Berlin since Sep-2015
- Developing machine learning algorithms for predictive analytics

  Making use of SQL and Python daily
- Data Ambassador & Volunteer Assessment @ Data Science for Social Good Berlin - https://dssg-berlin.org/
- Founding member of The AI Guild - https://www.theguild.ai/

The AI Guild is the go-to-community for Data Scientists, Data Engineers, Machine Learners, and Deep Learners accelerating the adoption of AI.

Agenda

- **Introduction**
- **SQL Concepts and Basic Operations**
- **More Operations: Joins, Aggregations, Subqueries**
- **SQL Functions: Window, Analytic**
- **Data Science Interview Questions**
- **Extra: Tricks and Tips**

Introduction

What is SQL?

Why SQL?

The RDBMS Landscape

NoSQL?

What is SQL?

- SQL = Structured Query Language
- Language used to interact with a database
  - Data stored in a relational database is **dynamic**: it can be queried, modified, and manipulated with basic SQL queries.
- Based on relational algebra and tuple relational calculus
- SQL was designed specifically for data
  - No surprise it excels at accessing & organizing data
- Invented in the 1970s and used with commercially since 1979 ...

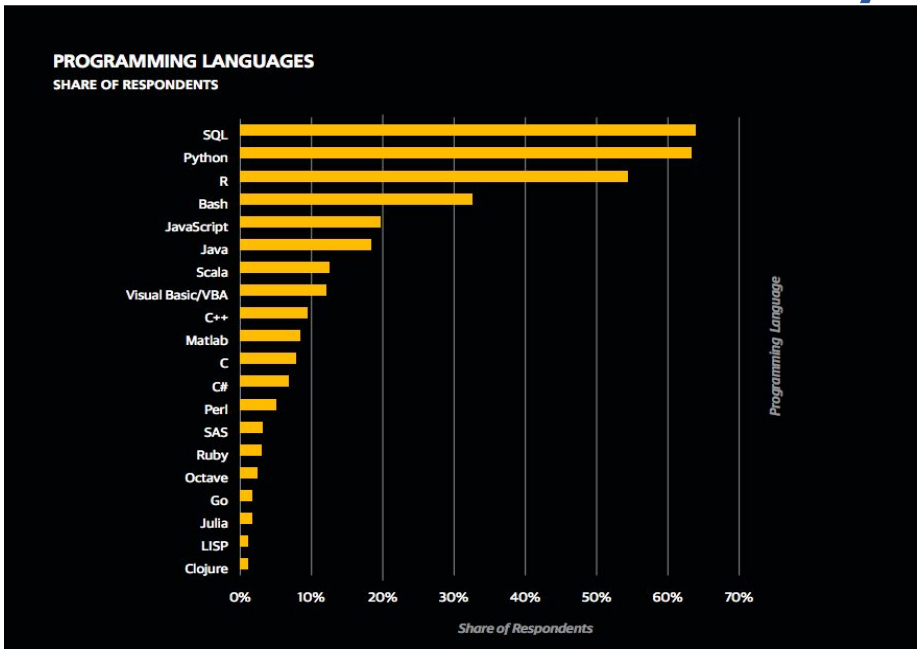
Why SQL?

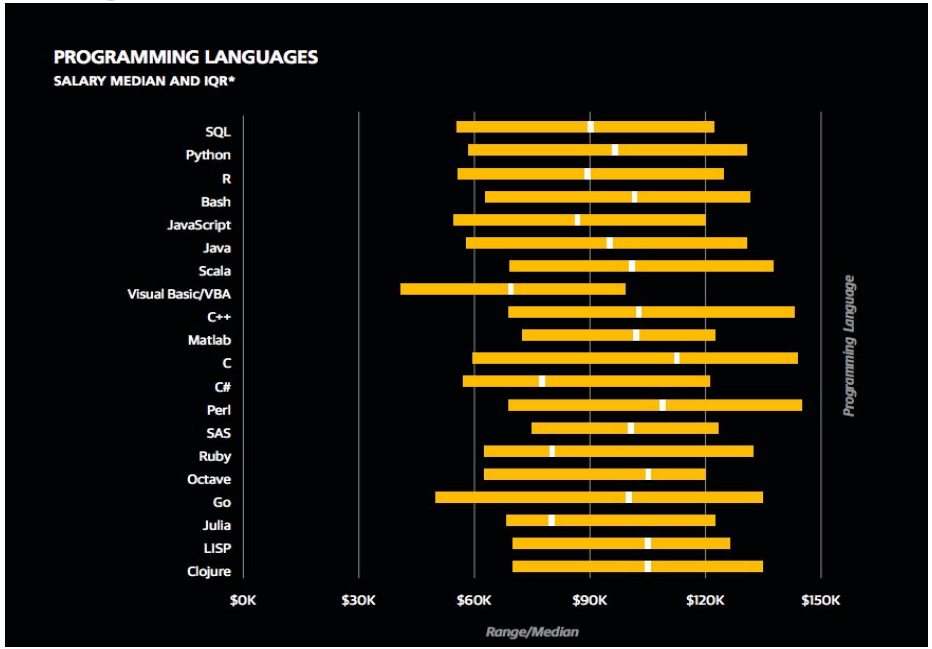
- Effective at data manipulation by design
  - It's more efficient to bring the computation to the data, rather than bringing the data to the computation.
- Battle-tested and proved to be robust and reliable
  - Used in many different scenarios from pre-web offline databases to global apps like Facebook.
  - Numerous plans patterns for backups, change management and operational rigor.
- Extensible
  - Adapts to new requirements, new processing techniques and calculations have been added over the years.

Why SQL?

- Open Source and Interoperability
  - SQL syntax varies only slightly between vendors, making it possible to reuse it with some modification.
- Ubiquity and Simplicity
  - Easy to learn, SQL is almost common knowledge.
  - Shared knowledge in the form of documentation and active community
  - Skill sets transfer between companies and industries
    - Deeply understanding the relational database systems that SQL runs on is another thing!
- Databases have become a ‘solved problem’ you no longer need to think about if you want to work with data.

Why SQL?



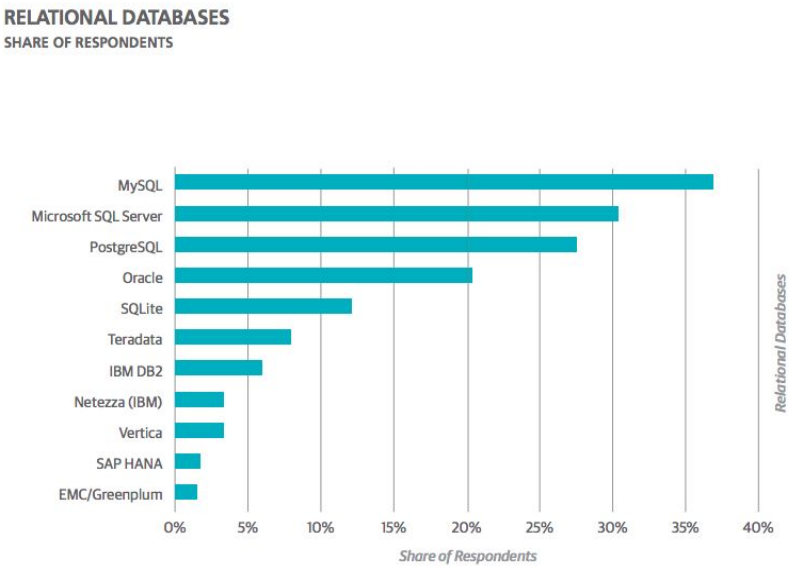


It is a **must-have skill** when you're working towards getting a job in the data science industry.

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

The RDBMS Landscape

- RDBMS = Relational Database Management System



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

NoSQL?

- SQL and NoSQL databases play different roles
  - SQL databases have their drawbacks and aren't the best choice for certain jobs. They are the best when consistent data integrity is essential.
- NoSQL motivation: better scaling
  - You can still have millions of users with a SQL DB with no issues.
  - It's possible to scale RDBMS, just requires knowing the tradeoffs.
- Specific databases:
  - Elasticsearch (document-oriented)
  - Cassandra
  - MongoDB
  - Neo4J (graph database)

SQL Basics

Database Structures

Database Objects

Data Types

Types of SQL Statements

The Basic SQL Query

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

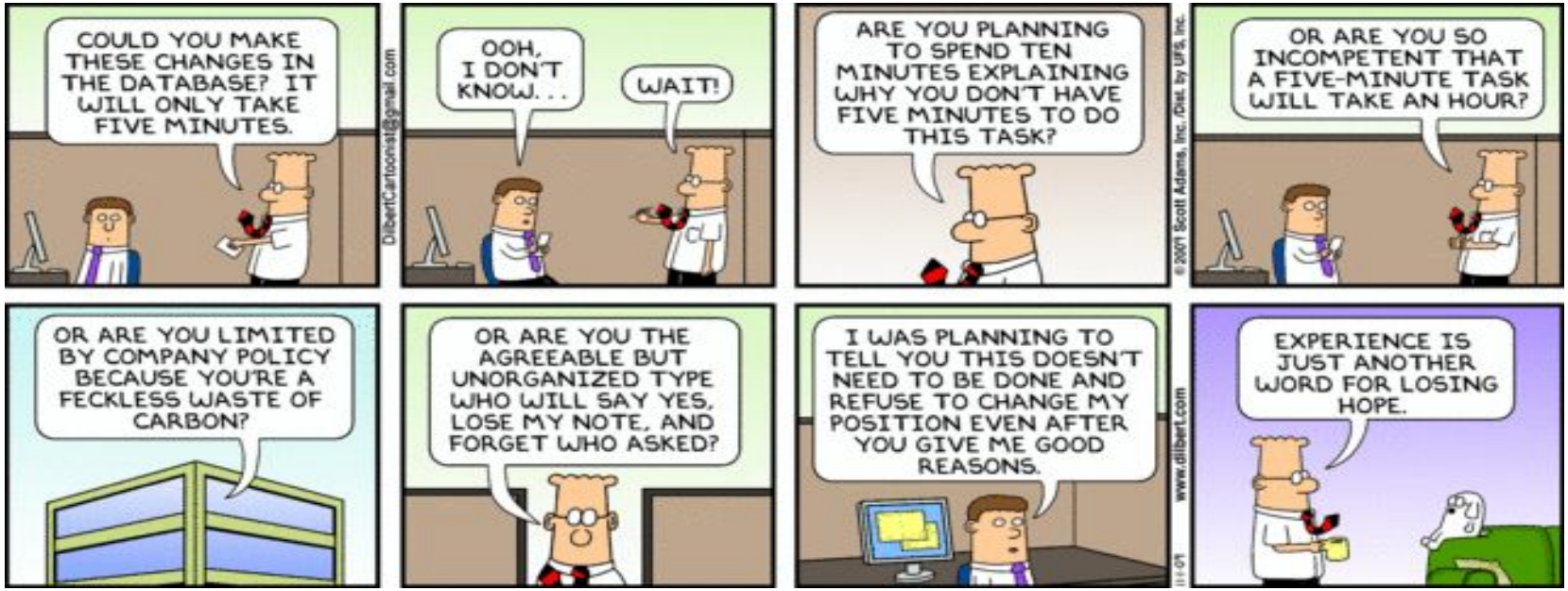
DDL Example - Create

```
CREATE TABLE category
(
 category_id INTEGER DEFAULT nextval('category_category_id_seq' ::
REGCLASS) NOT NULL
   CONSTRAINT category_pkey
   PRIMARY KEY,
 name        VARCHAR(25)   NOT NULL,
 last_update TIMESTAMP DEFAULT now()   NOT NULL
);
```

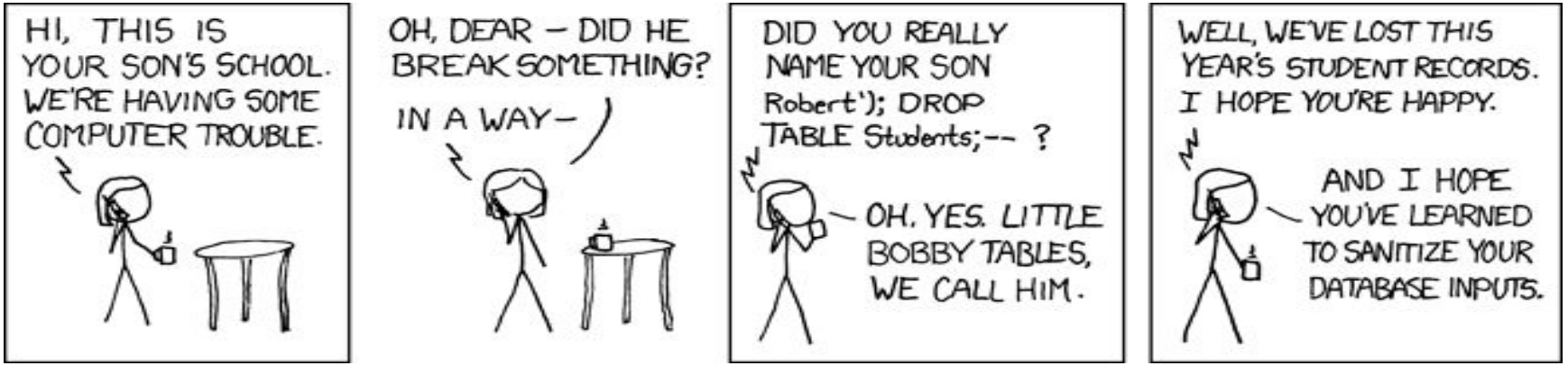
DDL Examples - Create, Alter

- 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?

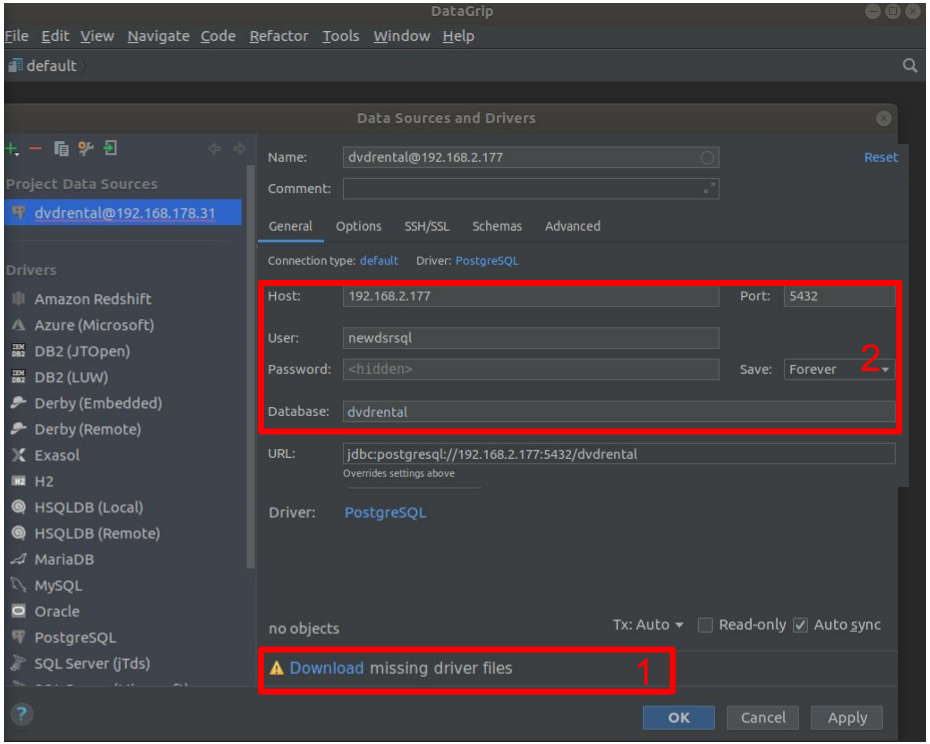
WARNING

- 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

- ➔ File
- ➔ Data Sources
- ➔ “+”
- ➔ PostgreSQL

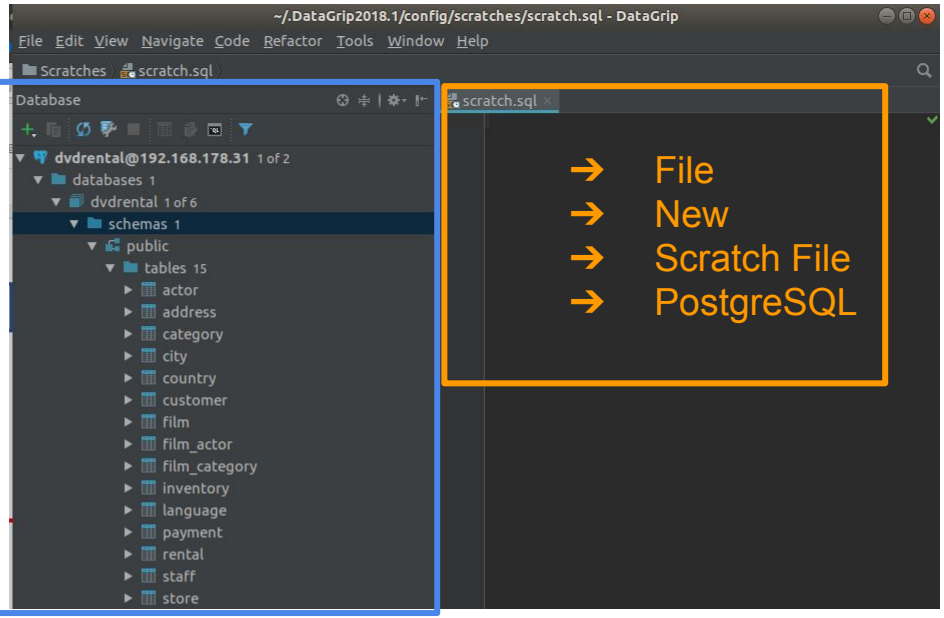


Download at https://www.jetbrains.com/datagrip/

Enough Theory, let's try it!

DataGrip

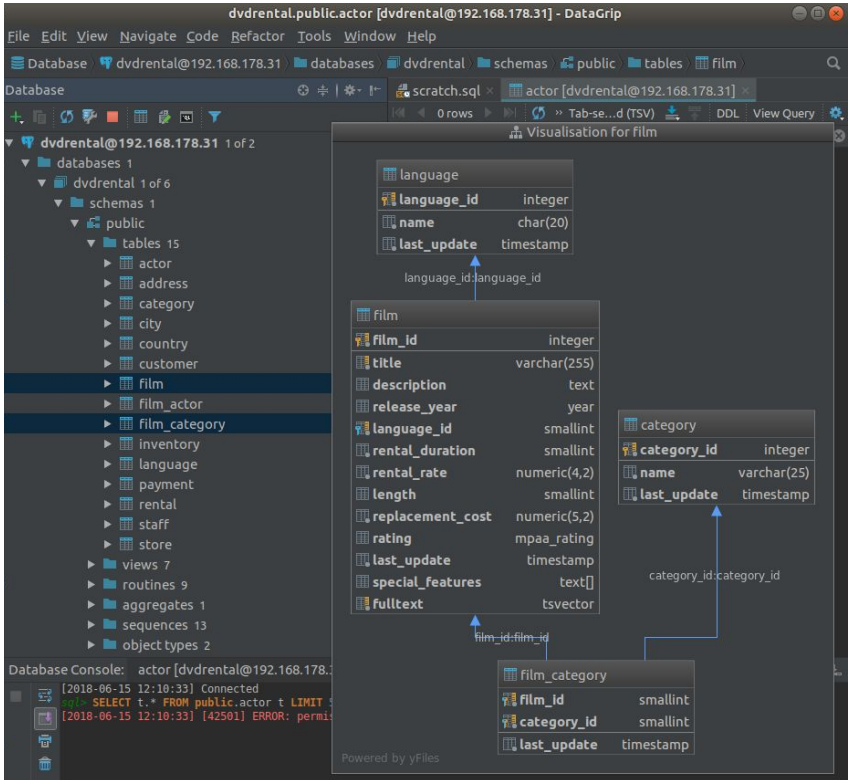
- View
- Tool Windows
- Database



DVDRental Schema

DataGrip

- ➔ Select table
- ➔ Right-click
- ➔ Diagrams
- ➔ Show Visualisation Popup ...



Example

- List all films with their title, rating and length

```
SELECT
    title,
    rating,
    length
FROM
    film
```

- **Exercise**:

    1. Find the film titles that are R rated and have less than 1 hour of length.

    EXTRA: Order the list of films above by length - from longer to shorter.

Operations

Join

Union

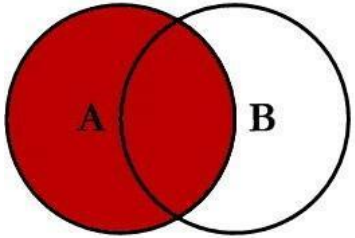
Subqueries

The WITH Clause

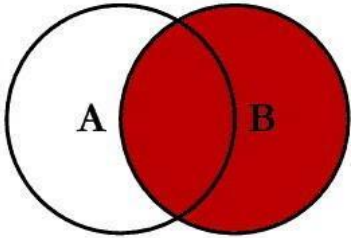
SQL Operations Overview

- 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
- To see more rows: Union tables
  - Union / Union Distinct / Union All

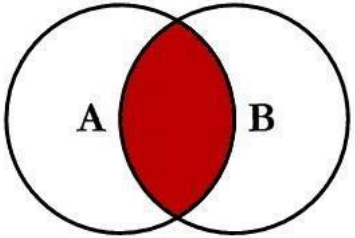
SQL JOINS



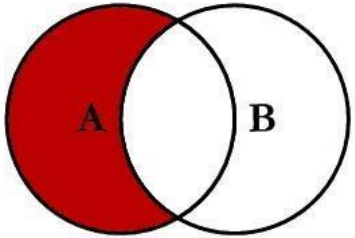

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



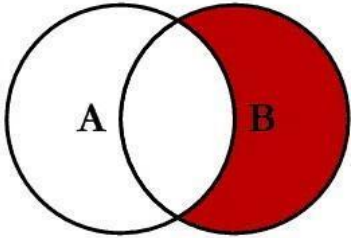

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



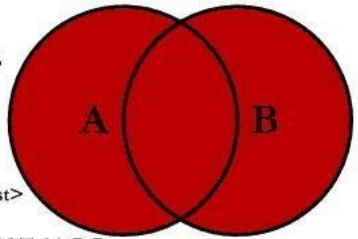

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



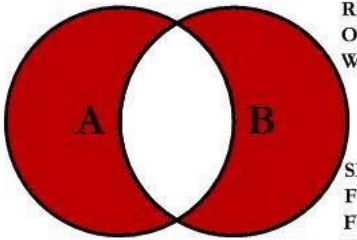

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
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                1
    FROM Geography
    WHERE Region_Name = ‘West'
    );
```

PostgreSQL executes the query that contains a subquery in the following sequence:

1. Executes the subquery.
2. Gets the result and passes it to the outer query.
3. Executes the outer query.

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 film titles with the customer names

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

- **Exercise**:
  1. Re-do Subquery exercise 1) using WITH.

Functions

Aggregate Functions

Window Functions

Date Functions

SQL Functions Overview (1/2)

- Aggregate Statistics
  - Compute a single result from a set of input values: COUNT, SUM, AVG, MIN, MAX, STRING_AGG.
  - Used with GROUP BY and/or HAVING.
- Window Functions
  - Perform calculations across sets of rows that are related to the current row: RANK, ROW_NUMBER, NTILE.

SQL Functions Overview (2/2)

- Date/time Functions
  - Manipulating and handling dates and timestamps: CURRENT_DATE, DATE_TRUNC, DATE_PART.
- Others:
  - 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.
  - Conditional: CASE WHEN, COALESCE;
  - Mathematical: ROUND, CEILING, FLOOR, LOG, SQRT, POWER;

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

```
SELECT COUNT(column_name)
FROM table_name
```

GROUP BY Syntax

```
SELECT
    COUNT(column_name1),
    column_name2
FROM
    table_name
GROUP BY
    column_name2
< optional: WHERE column_name = ... >
```

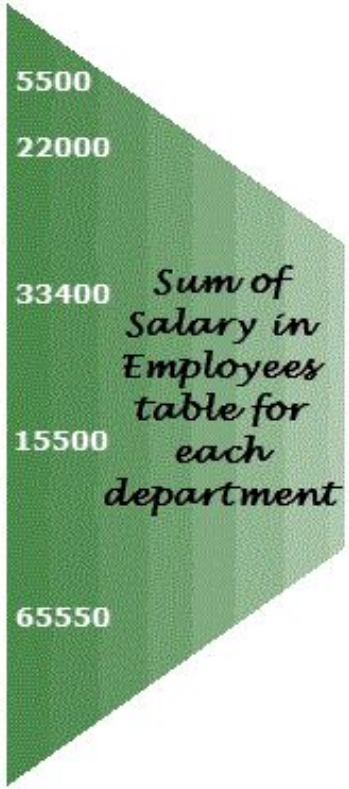
HAVING Syntax

```
SELECT
   COUNT(column_name1),
   column_name2
FROM
   table_name
GROUP BY
   column_name2
HAVING
   COUNT(column_name1) = ...
```

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 (payment.amount).
2. customers who have spent more than $200.
3. the number of rentals for each category.
4. the number of rentals for each film with its category.

EXTRA: films which have *film.rental_rate* higher than the average *film.rental_rate* between all films in the DB.

Window Functions

- Perform calculations across sets of rows that are related to the current row.
- `ROW_NUMBER() OVER (PARTITION BY <column> ORDER BY <column>)`
  - Unique number to each row within its partition, counting from 1.
- `RANK() OVER (PARTITION BY <column> ORDER BY <column>)`
  - Rank of current each row within its partition, with gaps.
- `DENSE_RANK() OVER (PARTITION BY <column> ORDER BY <column>)`
  - Rank of current each row within its partition, without gaps.
- `NTILE(num_buckets) OVER (PARTITION BY <column> ORDER BY <column>)`
  - 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 last returned film title - show customer name and return date.
  2. Find the 10% most profitable customers (top 10%).
  3. Find the most rented film for each category (start from aggregate exercise #4).

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.

Data Science Interview Questions

Data Science Interview Questions

From Data Science Dojo Blog - August 2019

https://blog.datasciencedojo.com/data-science-interview-questions

- List of 101 actual data science interview questions
- Asked between 2016-2019
- Amazon, Microsoft, Facebook, Google, Netflix, Expedia, etc
- 6 of them are SQL questions

Data Science Interview Questions

1. How would you handle NULLs when querying a data set?
2. How will you explain JOIN in the simplest possible way?
3. Select all customers who purchased at least two items on two separate days.
4. What is the difference between DDL, DML and DCL?
5. Why is database normalization important?
6. What is the difference between clustered and non-clustered index?

Data Science Interview Questions

1. How would you handle NULLs when querying a data set?

- A null is not the same as 0 or blank, it represents unknown or missing value
- if NULL exists during some arithmetic operations, then the answer always remains null

Data Science Interview Questions

3. Select all customers who purchased at least two items on two separate days.

```
SELECT Customer_ID, COUNT(DISTINCT Item_ID) as ‘item’, COUNT(DISTINCT
Purchase_Date) as ‘date’
FROM Purchase_List
GROUP BY Customer_ID
HAVING ‘date’ >= 2 AND ‘item’ >= 2
```

Data Science Interview Questions

5. Why is database normalization important?

Database normalization is a process used to organize a database into tables and columns. This procedure helps to achieve the following:

- All the data is stored in one place ensuring consistency
- Removes the duplicate records
- Minimizes data modification issues
- Querying the database is simplified

Data Science Interview Questions

6. What is the difference between clustered and non-clustered index?

The purpose of indexes is to **speed-up** query process.

- A **clustered** index defines the order in which data is physically stored in a table. Since the data can be sorted in only one order, there can be only one clustered index per table. It is **faster to read** than non-clustered index as data is physically stored in index order.

Data Science Interview Questions

6. What is the difference between clustered and non-clustered index?

The purpose of indexes is to **speed-up** query process.

- A **non-clustered** index doesn't sort the data inside the table. A non-clustered index is stored at one place and table data is stored in another place. This allows for more than one non-clustered index per table. This method is **quicker to insert and update operations** than a clustered index.

Tips & Tricks

Connecting to Python

EXTRA - User Segmentation with SQL

EXTRA - Database Index & When indexes don't matter

EXTRA - Optimization Examples

Connecting to Python

- python library psycopg : pip install psycopg2

```
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=dania host=192.168.2.174")
cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
cur.execute("select * from film")
for result in ResultIter(cur):
    print(result)
```

Connecting to Python

- python library SQLAlchemy includes dialects for SQLite, Postgresql, MySQL, Oracle, MS-SQL, Firebird, Sybase and others.
- Example:

```
from sqlalchemy import create_engine

eng = create_engine("postgresql://username@host/dbname")
with eng.connect() as con:
    rs = con.execute("SELECT film.title FROM film")
    data = rs.fetchone()[0]
    print "Data: %s" % data
```

- Tutorial with detailed steps: http://zetcode.com/db/sqlalchemy/rawsql/

Connecting to Python

- python library pandas is well suited for working with tabular data with heterogeneously-typed columns, as in an SQL table.
- ***pandas.read_sql*** - Read SQL query or database table into a DataFrame.

```
from sqlalchemy import create_engine

import pandas

eng = create_engine("postgres://user:pass@host/database”)

data = pandas.read_sql("SELECT film.title FROM film", con=eng)

data
```

- Comparing SQL and pandas

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
```

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.

Questions ?

Thank you!

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