SQL and Databases

30. 6. 2022

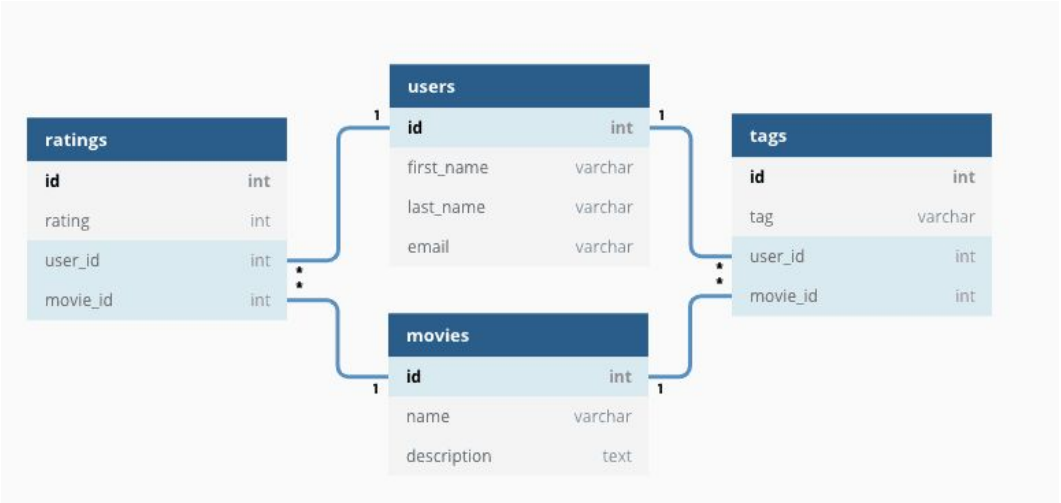
Databases introduction

- Collection of data organized in a certain way
- Classic example - tables with rows containing information and columns maintaining information about the field
- Standard way how to keep information organized

Examples of databases

- sqlite3
- PostgreSQL
- Oracle
- MySQL
- MS Access
- MongoDB
- Apache Ignite
- Neo4J
- Elasticsearch

Relational Database



- (usually) deduplicated
- (usually) normalized data
- rows - actual info
- columns - what and data type
- primary key - unique identifier
- foreign key - for match to the other tables

What is Relational Model

Posts

ID	Title	Date	Likes	Shares	Comments	Fanpage	Fanpage ID
1	The new G-class	01.01.18	6500	36	300	MB HQ	1
2	Life is a journey	02.01.18	5000	59	356	MB HQ	1
3	The legend is back	03.01.18	4264	123	427	MB HQ	1
4	Stronger than Time	03.01.18	5673	34	864	MBD	2
5	Mark your calendars	03.01.18	1348	99	13	MB HQ	1
6	The new G-class	04.01.18	3458	120	1267	MBD	2
7	Modern. Sleek. Elegant	04.01.18	2490	58	458	MB HQ	1

Fanpages

ID	Fanpage	Total Posts	Total Followers	Total Likes
1	MB HQ	34.456	2034568	41347200
2	MBD	12.567	1023461	15080400

Primary keys define relationships within relational database

The relational model

1. Organizes data into tables of:

- COLUMNS (**attributes**)
- ROWS (**records**)

2. Assigns:

- Unique ID (**primary key**) for each row

What is SQL?

Method of communicating between You and Database

Structured Query Language (prev. SEQUEL)

Standard language for **relational database management systems**

Non-procedural language - you cannot write complete applications (programs) with SQL

SQL has **four main functions**:

•CREATE Data

•**READ Data**

•UPDATE Data

•DELETE Data

NoSQL

- Key value: data is stored as attribute names or keys with values
- Document: contains many different key value pairs
- Graph: used to store data related to connections or networks
- Column: data is stored as columns instead of rows

SQL vs NoSQL

SQL	**NoSQL**
Schema designed at beginning	Flexible schema design
SQL	Many different languages
Vertically Scalable	Horizontally scalable
Best for complex queries	Best for complex, unstructured data

SQL: Data Model

Entity

- Facebook post, facebook page
- Unique
- “row in a table”

Attributes

- Number of likes, date, id, author
- Characteristics defining the entity
- “column in a table”

Relationships

- Association between entities
- One-to-many (page to post)
- Many-to-many (worker to projects)
- One-to-one (office manager to office)

Entity Relationship

- Author to song
- Husband to wife
- Parents to children
- Cities in the country
- Book to writer
- Book edition to publisher

one-to-one
one-to-many
many-to-many

Unique Key/Primary Key

- Unique identifier of a row (entity)
- Primary key can be combined of multiple columns
- Unique key should not be null (empty) vs Primary key can’t be null

Foreign key

- A Foreign Key is a field (or collection of fields) in one table, that refers to the Primary Key in another table.
- The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

Persons Table

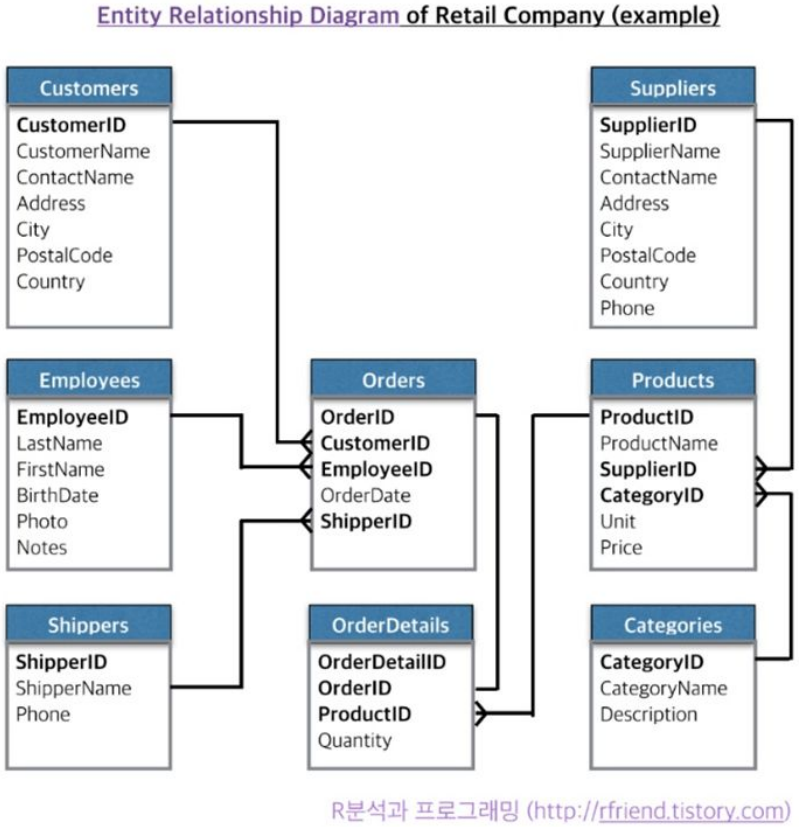
PersonID	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

Orders Table

OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3
3	22456	2
4	24562	1

Practice database

W3School provides testing database to work on: https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_all



SELECT

```
SELECT something FROM somewhere;

SELECT * FROM customers

SELECT city FROM customers;

SELECT DISTINCT city FROM customers;
```

Task

In how many different countries are customers located?

Answer

```
SELECT DISTINCT country FROM Customers;
```

→ 21

WHERE

- WHERE clause is used to filter records with condition.
- SELECT * FROM Customers WHERE Country == 'Mexico';

Task:

In how many cities in UK do we have customers located?

Answer

```
SELECT distinct city FROM Customers where country == "UK";
```

→ 2

AND, OR, NOT

- SHOW ME CUSTOMERS THAT ARE NEITHER FROM THE UK NOR FROM GERMANY

```
SELECT * FROM Customers WHERE NOT Country= 'UK' AND NOT Country =
'Germany';
```

- SHOW ME CUSTOMERS FROM US CITIES Eugene or Portland

```
SELECT * FROM Customers
WHERE Country='USA' AND (City='Eugene' OR City='Portland');
```

ORDER BY

- Ordering your results
- Must be placed after WHERE clause in the query
- ASC default, DESC for descending order
- SELECT * FROM Customers ORDER BY Country ASC

Task:

Sort the ContactName column in Suppliers table in descending order. What is the supplier name of the first record?

Answer

```
SELECT * FROM Suppliers ORDER BY ContactName DESC;
```

→ Tokyo Traders

Aggregation functions

- `Count()`
- `Sum()`
- `Min()`
- `Max()`
- `Avg()`
- `SELECT sum(price) FROM Products where SupplierID == 5`
- `SELECT min(price) FROM Products where SupplierID == 5`

Task

What is the total quantity of items that have been ordered, ever? Hint: use OrderDetails table

Answer

```
SELECT sum(Quantity) FROM OrderDetails
```

→ 12743

GROUP BY

```
SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country;
```

- Grouped information directly based on column, usually with combination with aggregation functions
- DATA -> WHERE -> GROUP BY -> HAVING -> ORDER BY -> LIMIT.

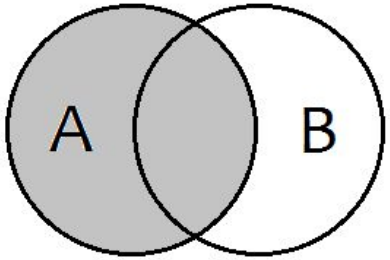
JOIN

- Two tables with different information

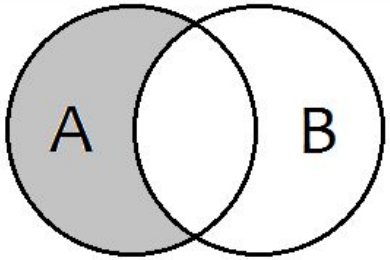
```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

- Both tables have to share one column (not necessary same name) - links them

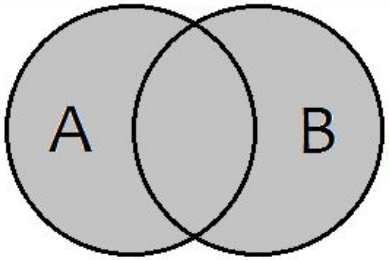
SQL JOINS



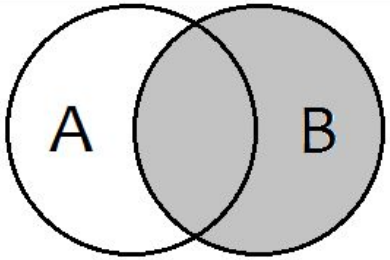

```
SELECT *
FROM TableA a
LEFT JOIN TableB b
ON a.Key = b.Key
```



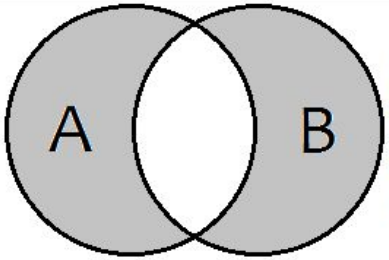

```
SELECT *
FROM TableA a
LEFT JOIN TableB b
ON a.Key = b.Key
WHERE b.Key IS NULL
```



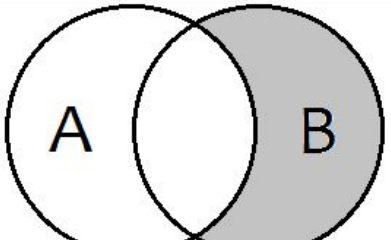

```
SELECT *
FROM TableA a
FULL OUTER JOIN TableB b
ON a.Key = b.Key
```



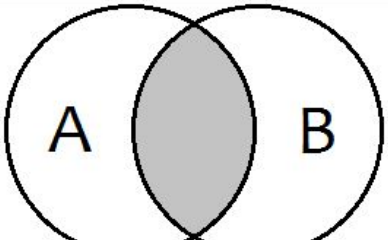

```
SELECT *
FROM TableA a
RIGHT JOIN TableB b
ON a.Key = b.Key
```




```
SELECT *
FROM TableA a
FULL OUTER JOIN TableB b
ON a.Key = b.Key
WHERE a.Key IS NULL
OR b.Key IS NULL
```




```
SELECT *
FROM TableA a
RIGHT JOIN TableB b
ON a.Key = b.Key
WHERE a.Key IS NULL
```




```
SELECT *
FROM TableA a
INNER JOIN TableB b
ON a.Key = b.Key
```

UPDATE, DELETE

- `UPDATE students SET age = 35 WHERE id = 1 LIMIT 1;`

- `DELETE FROM students WHERE id = 1 LIMIT 1;`

ALTER TABLE

- `ALTER TABLE students ADD COLUMN grade INTEGER;`
- `ALTER TABLE students DROP COLUMN grade;`

- `ALTER TABLE distributors ADD CONSTRAINT distaddr FOREIGN KEY (address_id) REFERENCES addresses (id);`

Advanced SQL

- Many other clauses to help you get correct results
- Indexes - to help you speed up your searches
- Triggers in PostgreSQL
- Insert to add data

SQL Dialects

- not every Relational database speak same dialect:
- Oracle has strong dialect - many of the functions there are added only for Oracle and do not exists in plain SQL.
- PostgreSQL has extensions for spatial data, images, which does not exist elsewhere.

Best practices

- SELECT * FROM TABLE can take very long
- SQL commands written in CAPITAL LETTERS
- comments with **--comment line** or ***/ comment block */**
- UPDATE and DELETE LIMIT 1 for safety

Install sqlite3

- In-process library that implements a serverless, zero configuration, self contained SQL database engine
- https://www.sqlite.org/download.html
- download DLL + tools, extract to new folder (eg C:\sqlite) and add the folder to PATH (environment variables)
- - test by typing **sqlite3** to cmd line
- Windows - needs download
- Mac + Ubuntu usually preinstalled

Import database

- Download database from https://www.sqlitetutorial.net/sqlite-sample-database/
- To import database into sqlite3 do:

extract .zip archive

type into terminal:
sqlite3 C:\Users\tyna\Desktop\chinook.db
.tables
.schema table_name
When using SQL commands, end them with ; SELECT title FROM albums;

DATA TYPES IN SQLITE3

- **NULL.** The value is a NULL value.
- **INTEGER.** The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
- **REAL.** The value is a floating point value, stored as an 8-byte IEEE floating point number.
- **TEXT.** The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- **BLOB.** The value is a blob of data, stored exactly as it was input (image for example) - binary large object.

Connecting to SQLite via Python

- **sqlite3** module included in standard library
- standardized DB-API (PEP 249), all other major database clients in Python use it (PostgreSQL, MySQL)

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute('SELECT name FROM books WHERE pages>400')
print(c.fetchall())
conn.commit()
```

Connecting to SQLite via Python DB-API

- Connection - should have **commit()** and **rollback()** for transactions - (single unit of work, can consist of many operations)
- Cursor - does actual traversing over DB content - should have **execute(operation, params)**, **fetchone(), fetchmany(rowcount), fetchall(), rowcount**
- set of always present Exceptions (for example **DatabaseError**, **OperationalError**, **ProgrammingError**)
- you should do cursor.close(), connection.close()
- good idea to use **try/except** block for DB operations and closing connection in **finally** block

Connecting to PostgreSQL

- **psycopg2** package using same syntax as **sqlite3** (DB-API)
- not part of standard library - pip install psycopg2
- extra connection parameters need to be set up
- host (url), port (5432), database name, user, password

PostgreSQL extensions

- PostgreSQL is supporting usage of external extensions which helps with specific tasks
- PostGIS - enables spatial data usage and fast operations, usage of geometry type
- PostPic - new Image type and image usage directly in database

SQL Alchemy and ORM - object relational mapping

- SQL Alchemy - standard interface over different database engines, to focus on implementing actual program logic instead of handling connections, syntax etc.
- most of times used for its optional ORM part
- ORM translated Python classes to tables and converts function calls to SQL statements, handles data types

```
class Book(Base):
    __tablename__ = 'books'
    id=Column(Integer, primary_key=True)
    title=Column('title', String(500))
    author=Column('author', String(500))
    in_stock=Column('in_stock', Boolean)
    quantity=Column('quantity', Integer)
    price=Column('price', Numeric)
```

SQL Alchemy ORM

- Declarative base Class - you describe actual database tables, that the classes will be mapped to
- After creating a session, you can add/query/delete instances of your Class from real database

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, Sequence('user_id_seq'), primary_key=True)
    name = Column(String(50))
ed_user = User(name='ed')
session = Session()
session.add(ed_user) # not created yet
our_user = session.query(User).filter_by(name='ed') # he was created, when we queried him
ed_user is our_user # returns True
session.commit() # to save changes manually
```

https://docs.sqlalchemy.org/en/13/orm/tutorial.html

SQL Alchemy ORM

- Adding more User objects via session.add_all([])

```
session.add_all([User(name='wendy'), User(name='fred')])
```

- Updating data ed_user.name='' # still not updated to DB
- List out non-commited updates via session.dirty or or non-commited added ones via session.new

SQL Alchemy ORM - rolling back

- Possible to revert wrong commits via **.rollback()**

```
ed_user.name = 'Edwardo'
fake_user = User(name='fakeuser')
session.add(fake_user)
session.query(User).filter(User.name.in_(['Edwardo', 'fakeuser'])).all() # gets flushed
when queried
session.rollback()
ed_user.name # 'ed'
fake_user in session # False
```

SQL Alchemy ORM

```
# iterating over query results
for name in session.query(User.name):
    print(name)
```

Filtering - LIKE vs ILIKE (case insensitive), to be sure, as different DBs have different implementations of LIKE

```
query.filter(User.name.ilike('%ed%')) # get me all users containing ed in name
query.filter(~User.name.in_(['jane', 'tom'])) # ~ is negation, so it gets me users, which dont have name jane or tom
```

counting - with .count() in the end

Mongodb, example of NoSQL DB

- data are stored in documents (JSON-like), ids created by mongodb itself
- PyMongo - pip3 install PyMongo
- instead of tables, collections

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]


myquery = { "address": "Park Lane 38" }
mydoc = mycol.find(myquery)
for x in mydoc:
     print(x)
```

Project to practice **Python** & databases

- 3 .csv files to import into database provided in the github repository in https://github.com/UndeadFairy/pyladies_vienna/tree/master/databases
- Read CSV
- Cleanup data (revenue $, rename columns, correct No Data values)
- Connect to sqlite3 database
- Create tables
- Import data
- Answer some questions about data

Project to practice **Python** & databases QUESTIONS

- Which series has got the most episodes (return name and number)
- Return average rating per series
- Which viewer watched for longest time on each day (10, 11, 12) (return name and how many minutes)
- Return the users which are registered on the platform the longest and shortest based on logins
- Order series by total minutes streamed descending
- Which series are most liked by men and women respectively based on minutes streamed
- ...

Sum it up

- Databases are useful for storing data in organized way
- Many many many types of them with different strengths
- For incrementally growing data in terms of structure, use NoSQL
- Python has great interface for communication with your data