DA1: DESIGNING APPLICATIONS IN PYTHON
LECTURE
DATABASES

Aidos Sarsembayev

CONTENT

- DBs types
- Relational DBs
- Non-Relational DBs

TYPES OF THE DATABASES

- Relational
  - **SQL / RDBMS**
- Non-relational
  - **NoSQL**

RELATIONAL DATABASE MANAGEMENT SYSTEMS

- are more widely known and understood than NoSQL
- Is a collection of tables, each with a *schema* that represents the fixed attributes and data types that the items in the table will have
- provides functionality for reading, creating, updating, and deleting data (using SQL statements)

RELATIONAL DATABASE MANAGEMENT SYSTEMS

- The tables in a relational database have ***keys*** associated with them, which are used to identify specific columns or rows of a table and facilitate faster access to a particular table, row, or column of interest.

SOME OF THE MOST POPULAR **RDBMS**

- **Oracle** - Oracle Database (commonly referred to as Oracle RDBMS or simply as Oracle) is a multi-model database management system produced and marketed by Oracle Corporation.
- **MySQL** - MySQL is an open source RDBMS based on Structured Query Language (SQL). MySQL runs on virtually all platforms, including Linux, UNIX, and Windows.
- **Microsoft SQL Server** - Microsoft SQL Server is an RDBMS, that supports a wide variety of transaction processing, business intelligence and analytics applications in corporate IT environments.
- **PostgreSQL** - PostgreSQL, often simply Postgres, is an object-relational database management system (ORDBMS) with an emphasis on extensibility and standards compliance.
- **DB2** - DB2 is an RDBMS designed to store, analyze and retrieve data efficiently.

RDBMS PRONS

- Relational databases are well-documented and mature technologies, and RDBMS are sold and maintained by a number of established corporations.
- SQL standards are well-defined and commonly accepted.
- A large pool of qualified developers have experience with SQL and RDBMS.
- All RDBMS are ACID-compliant, meaning they satisfy the requirements of Atomicity, Consistency, Isolation, and Durability.

RDBMS CONS

- RDBMSes don't work well — or at all — with unstructured or semi-structured data, due to schema and type constraints. This makes them ill-suited for large analytics or IoT event loads.
- The tables in your relational database will not necessarily map one-to-one with an object or class representing the same data.
- When migrating one RDBMS to another, schemas and types must generally be identical between source and destination tables for migration to work (schema constraint). For many of the same reasons, extremely complex datasets or those containing variable-length records are generally difficult to handle with an RDBMS schema.

NOSQL / NON-RELATIONAL DATABASES

- can take a variety of forms
- the critical difference between NoSQL and relational databases is that RDBMS schemas rigidly define how all data inserted into the database must be typed and composed, whereas NoSQL databases can be schema agnostic, allowing unstructured and semi-structured data to be stored and manipulated.

TYPES OF NOSQL DATABASES

- Key-Value Stores
- Wide Column Stores
- Document Stores
- Graph Databases
- Search Engines

TYPES OF NOSQL DATABASES

- Key-Value Stores | extremely simple key-value paired DBs such as Redis and Amazon DynamoDB
- Wide Column Stores
- Document Stores
- Graph Databases
- Search Engines

TYPES OF NOSQL DATABASES

- Key-Value Stores
- Wide Column Stores | such as Cassandra, Scylla, and HBase, are schema-agnostic systems that enable users to store data in column families or tables, a single row of which can be thought of as a record — a multi-dimensional key-value store.
- Document Stores
- Graph Databases
- Search Engines

TYPES OF NOSQL DATABASES

- Key-Value Stores
- Wide Column Stores |
  These solutions are designed with the goal of scaling well enough to manage petabytes of data across as many as thousands of commodity servers in a massive, distributed system.
- Document Stores
- Graph Databases
- Search Engines

TYPES OF NOSQL DATABASES

- Key-Value Stores
- Wide Column Stores
- Document Stores | including MongoDB and Couchbase, are schema-free systems that store data in the form of JSON documents. Document stores are similar to key-value or wide column stores, but the document name is the key and the contents of the document, whatever they are, are the value.
- Graph Databases
- Search Engines

TYPES OF NOSQL DATABASES

- Key-Value Stores
- Wide Column Stores
- Document Stores |
  In a document store, individual records do not require a uniform structure, can contain many different value types, and can be nested. This flexibility makes them particularly well-suited to manage semi-structured data across distributed systems.
- Graph Databases
- Search Engines

TYPES OF NOSQL DATABASES

- Key-Value Stores
- Wide Column Stores
- Document Stores
- Graph Databases | such as Neo4J and Datastax Enterprise Graph, represent data as a network of related nodes or objects in order to facilitate data visualizations and graph analytics.
- Search Engines

TYPES OF NOSQL DATABASES

- Key-Value Stores
- Wide Column Stores
- Document Stores
- Graph Databases | such as Neo4J and Datastax Enterprise Graph, represent data as a network of related nodes or objects in order to facilitate data visualizations and graph analytics.
- Search Engines

TYPES OF NOSQL DATABASES

- Key-Value Stores
- Wide Column Stores
- Document Stores
- Graph Databases | A node or object in a graph database contains free-form data that is connected by relationships and grouped according to labels. Graph-Oriented Database Management Systems (DBMS) software is designed with an emphasis on illustrating *connections* between data points.
- Search Engines

TYPES OF NOSQL DATABASES

- Key-Value Stores
- Wide Column Stores
- Document Stores
- Graph Databases | As a result, graph databases are typically used when analysis of the relationships between heterogeneous data points is the end goal of the system, such as in fraud prevention, advanced enterprise operations, or Facebook's original friends graph.
- Search Engines

TYPES OF NOSQL DATABASES

- Key-Value Stores
- Wide Column Stores
- Document Stores
- Graph Databases
- Search Engines | such as Elasticsearch, Splunk, and Solr, store data using schema-free JSON documents. They are similar to document stores, but with a greater emphasis on making your unstructured or semi-structured data easily accessible via text-based searches with strings of varying complexity.

NOSQL PRONS

Since there are so many types and varied applications of NoSQL databases, it's hard to nail these down, but generally:

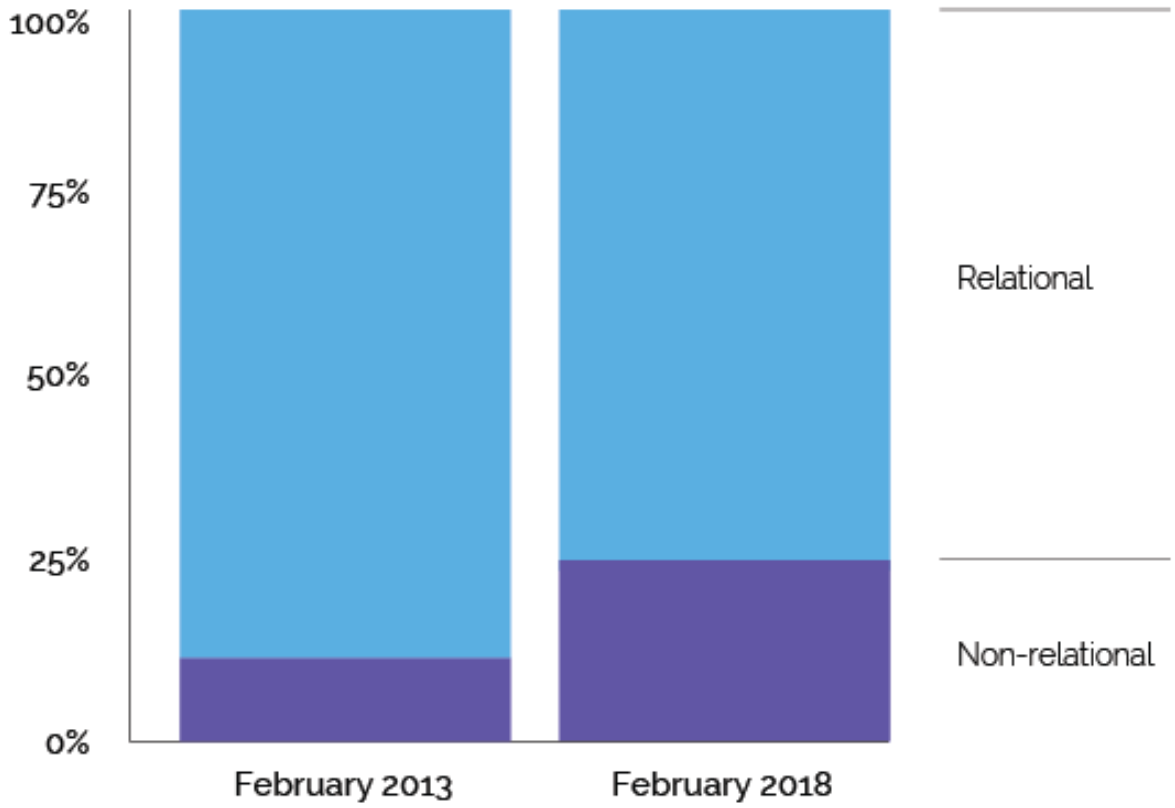
- Schema-free data models are more flexible and easier to administer.
- NoSQL databases are generally more horizontally scalable and fault-tolerant.
- Data can easily be distributed across different nodes. To improve availability and/or partition tolerance, you can choose that data on some nodes be "eventually consistent".

NOSQL CONS

These are also dependent on the database type. Principally:

- NoSQL databases are generally less widely adopted and mature than RDBMS solutions, so specific expertise is often required.
- There are a range of formats and constraints specific to each database type.

Popularity (percentage) Relational Databases vs. Non-Relational Databases



Source: https://db-engines.com/en/ranking_trend

MOST COMMON DATABASES FOR PYTHON WEB APPS

- PostgreSQL and MySQL are two of the most common open source databases for storing Python web applications' data.

- SQLite is a database that is stored in a single file on disk (non **client-server**). SQLite is built into Python but is only built for access by a single connection at a time. Therefore is highly recommended to not run a production web application with SQLite.

POSTGRESQL DATABASE

- PostgreSQL is the recommended relational database for working with Python web applications. PostgreSQL's feature set, active development and stability contribute to its usage as the backend for millions of applications live on the Web today.

MYSQL DATABASE

- MySQL is another viable open source database implementation for Python applications. MySQL has a slightly easier initial learning curve than PostgreSQL but is not as feature rich.

CONNECTING TO A DATABASE WITH PYTHON

To work with a relational database using Python, you need to use a code library. The most common libraries for relational databases are:

- psycopg2 for PostgreSQL.
- MySQLdb for MySQL. Note that this driver's development is mostly frozen so evaluating alternative drivers is wise if you are using MySQL as a backend.
- cx_Oracle for Oracle Database.
- SQLite support is built into Python 2.7+ and therefore a separate library is not necessary. Simply "import sqlite3" to begin interfacing with the single file-based database.

INSTALLING THE DB PACKAGES

- pip install psycopg2 # for Postgre
- import sqlite3 # SQLLite is embedded into Python3 no need to install

SQLITE3

- Is kind of a python wrapper for SQL code
- It is good for relatively small problems (i.g. simple mobile apps), but not suitable for production of big projects

5 TYPICAL STEPS OF WORKING WITH SQLITE

1. Connect to a DB
2. Create a **cursor** object (kind of a pointer to access rows of a table of your DB)
3. Write an SQL query
4. Commit your changes to DB
5. Close the connection with DB

5 TYPICAL STEPS OF WORKING WITH SQLITE

```
import sqlite3


# step 1
# if you don't have a database yet, the connection will create it
# and the connection will be established
conn = sqlite3.connect("lite.db")
```

5 TYPICAL STEPS OF WORKING WITH SQLITE

```
# step 2
# create a cursor and execute the first query for creation of table
cur = conn.cursor()
```

5 TYPICAL STEPS OF WORKING WITH SQLITE

```
# step 3
cur.execute("CREATE TABLE store (item TEXT, quantity INTEGER, price REAL)")
```

5 TYPICAL STEPS OF WORKING WITH SQLITE

```
# step 4
# commit the changes
conn.commit()
```

5 TYPICAL STEPS OF WORKING WITH SQLITE

```
# step 5
conn.close()
```

5 TYPICAL STEPS OF WORKING WITH SQLITE

```
import sqlite3


conn = sqlite3.connect("lite.db")
cur = conn.cursor()
cur.execute("CREATE TABLE store (item TEXT, quantity INTEGER, price REAL)")
conn.commit()
conn.close()
```

5 TYPICAL STEPS OF WORKING WITH SQLITE

```
import sqlite3
conn = sqlite3.connect("lite.db")
cur = conn.cursor()
cur.execute("CREATE TABLE store (item TEXT, quantity INTEGER, price REAL)")
conn.commit()
conn.close()
```

```
Traceback (most recent call last):  File
".\db_exmpl.py", line 8, in <module>
cur.execute("CREATE TABLE store (item TEXT,
quantity INTEGER, price
REAL)")sqlite3.OperationalError: table store already
exists
```

If you execute it twice, you'll get an exception stating that the
DB exists

5 TYPICAL STEPS OF WORKING WITH SQLITE

```
import sqlite3

conn = sqlite3.connect("lite.db")

cur = conn.cursor()

cur.execute("CREATE TABLE  IF NOT EXISTS store (item TEXT, quantity INTEGER, price REAL)")

conn.commit()

conn.close()
```

Traceback (most recent call last): File ".\db_exmpl.py", line 8, in <module> cur.execute("CREATE TABLE store (item TEXT, quantity INTEGER, price REAL)")sqlite3.OperationalError: table store already exists

If you execute it twice, you'll get an exception stating that the
DB exists

INSERT VALUE

```
import sqlite3


conn = sqlite3.connect("lite.db")
cur = conn.cursor()
cur.execute("CREATE TABLE store (item TEXT, quantity INTEGER, price REAL)")
cur.execute("INSERT INTO store VALUES ('Wine Glass', 8, 10.5)")
conn.commit()
conn.close()
```

Not a good approach, since we will duplicate the values.
It's better to wrap into a method

INSERT VALUE

```
import sqlite3

def create_table():
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS store (item TEXT, quantity INTEGER, price REAL)")
    conn.commit()
    conn.close()

def insert(item, quantity, price):
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("INSERT INTO store VALUES(?,?,?)", (item, quantity, price))
    conn.commit()
    conn.close()

insert("Water Glass", 5, 0.5)
```

INSERT VALUE

```
import sqlite3
def create_table():
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS store (item TEXT, quantity INTEGER, price REAL)")
    conn.commit()
    conn.close()
def insert(item, quantity, price):
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("INSERT INTO store VALUES(?,?,?)", (item, quantity, price))
    conn.commit()
    conn.close()
insert("Water Glass", 5, 3)
def view():
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("SELECT * FROM store")
    rows = cur.fetchall()
    conn.close()
    return rows     # rows returned as a python list
print(view())
```

INSERT VALUE

```
def delete(item):
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("DELETE FROM store WHERE item=?",(item,))
    conn.commit()
    conn.close()

def update(quantity, price, item):
    conn = sqlite3.connect("lite.db")
    cur = conn.cursor()
    cur.execute("UPDATE store SET quantity=?, price=? WHERE item=?", (quantity, price, item))
    conn.commit()
    conn.close()
```

POSTGRES CASE

- You can create the DB in pgadmin
- In your python code you have to import psycopg2

```
def create_table():
    conn = sqlite3.connect("dbname = 'database1' user = 'postgres' password='postgres123' host = 'localhost' port = '5432'")
    cur = conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS store (item TEXT, quantity INTEGER, price REAL)")
    conn.commit()
    conn.close()
```

POSTGRES CASE

- cur.execute("INSERT INTO store VALUES('%s','%s','%s')" % (item, quantity, price))

OR

- cur.execute("INSERT INTO store VALUES(%s,%s,%s)", (item, quantity, price))

- cur.execute("DELETE FROM store WHERE item=%s",(item,))

- cur.execute("UPDATE store SET quantity=%s, price=%s WHERE item=%s", (quantity, price, item))