NoSQL Databases

Before we leave the topic of databases, let's talk about another type of databases; NoSQL databases. Nevertheless of today's discussion, we will continue to use PostgreSQL throughout this semester for academic reasons but keep your knowledge about NoSQL databases in mind for possible future opportunities.

Lecture Overview

- NoSQL Database History
- What are NoSQL Databases?
- Differences between NoSQL Databases and Relational Databases
- NoSQL Data Models
- Run BigTable Locally via Docker

Why NoSQL Databases?

1. Wouldn't be more efficient if you query semi-structured data without having to convert them into structured form?

2. Wouldn't be easier to store and query 1,000,000,000 of records without having to buy new database servers?

NoSQL History

- The NoSQL term was first widely used in the late 1990s to refer to open-source relational databases that scripts were used to interact with.
- The modern NoSQL term, referring to a non-relational database management system, started being used in 2009. There are some milestones in the history of NoSQL.
- In 1998, Carlo Strozzi used that term for the open-source database. In 2000, the Graph database Neo4J began with graph data-models coming in the table.
- In 2005, CouchDB project started as document data-store while in 2007, MongoDB, project began.

What is NoSQL Database/Store?

- There is no standard definition for NoSQL databases/stores. However, definitions boil down to one of two choices:
  - Some say it means Not Only SQL. In other words, you can use SQL, and some other non-SQL functionality, as well.
  - Other people say it simply means Not SQL.
- At its heart, NoSQL is usually defined as something called **Polyglot** persistence. Polyglot persistence is defined as using different data stores, in different circumstances. In other words, we have lots of different data models we can work with, and we use different ones in different situations, even within the same solution.
- NoSQL is a much more flexible, or malleable structure, than relational database management systems.

NoSQL Features

A NoSQL Database contains one, or more, of the following features:

- Non-relational
- Distributed
- Open-source
- horizontally scalable.

Although some people call relational databases as SQL databases, NoSQL databases are **NOT** the opposite of relational databases. **The general theme of NoSQL databases/stores is favoring performance over full consistency.**

Issues with Relational Databases (e.g. PostgreSQL)

There are two major issues with relational databases;

- They are not designed to scale (scale-up or scale-out)
- They are expensive when used at corporate scale (e.g. Oracle, SQL Server, etc.).

On the other hand, NoSQL databases/stores are designed for scalability and can handle semi-structured data nicely

How do NoSQL databases/stores differ from relational databases?

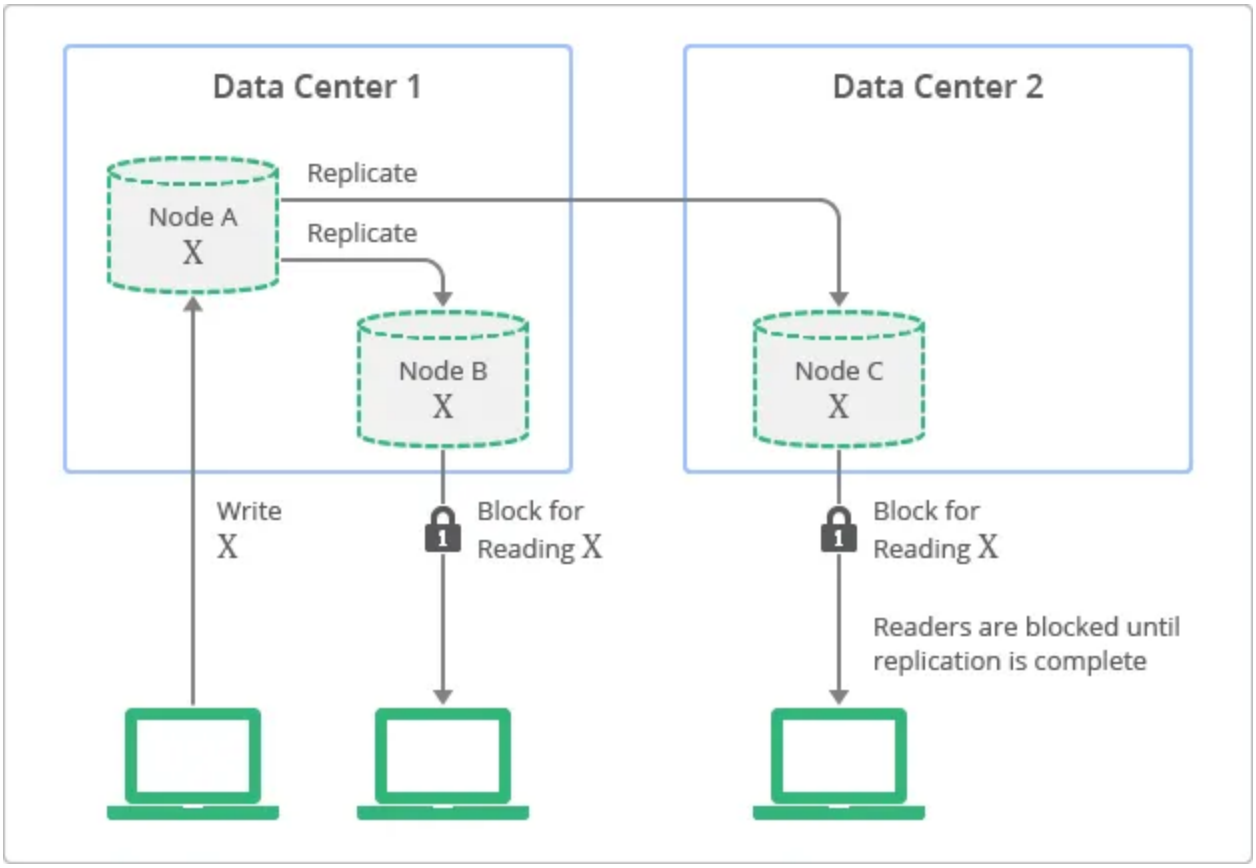
1. NoSQL databases/stores scale-up and scale-out easily since they are not based on the relational data model. There is no standard NoSQL model but the most common ones are **document stores, graph stores, column stores, and key-value stores**.

1. NoSQL has no schema and often uses aggregates, which increases flexibility.

1. Most NoSQL solutions are open-source, and are designed to optimize performance rather than consistency.

1. NoSQL databases/stores don't **fully** support **ACID** (Atomic, Consistent, Isolated, and Durable) semantics. As mentioned in our first lecture, relational databases support **ACID** semantics. SQL statements are executed in chunks, called transactions.
   - A transaction is *atomic* if all of the work in a transaction completes or commits, or none of it completes.
   - A transaction is *consistent* if it transforms the database from one consistent state to another consistent state.
   - A transaction is *isolated* if the results of any changes made during a transaction are not visible until the transaction has committed.
   - A transaction is *durable* if the results of a committed transaction survive failures.

SQL Database Consistency



The majority of NoSQL databases/stores don't support full consistency. Rather, they support *eventual consistency*. To be clearer, With NoSQL, **inserts and updates are asynchronous.** Changes may not be immediately propagated to all nodes and there are no real-time updates. When reading about NoSQL stores, you will see a reference for **BASE** semantics (instead of ACID). **BASE** stands for *Basically Available, Soft State, and Eventually Consistent.* We will talk about BASE semantics later in this lecture.

BASE Transactions

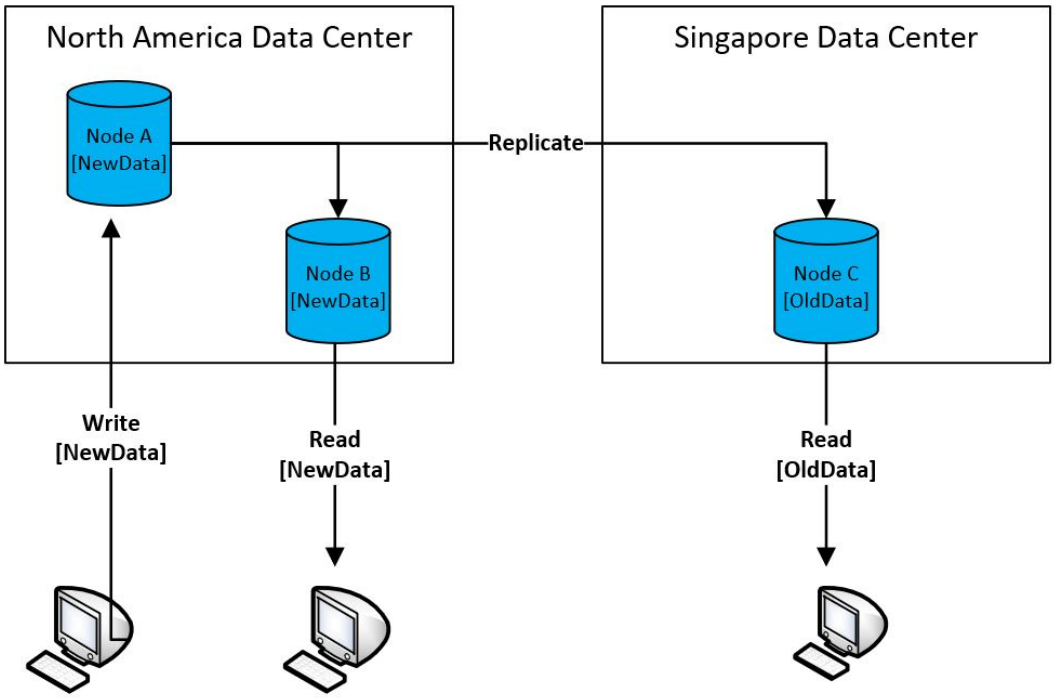
BASE stands for **Basically Available, Soft State, and Eventually Consistent.**

Basically Available data is available on multiple nodes due to data replication. It ensures availability even with multiple failures. Take a distributed system that is composed of multiple machines (or nodes). If one node goes down, we can keep working with the other nodes. So, basically available means the data's available even with multiple failures. I can have more than one node go down, and the data's still available. The only way this can be accomplished is by **data replication** by duplicating the data across multiple nodes.

Soft state means that I may have 20 nodes, and I may have made a change to one node and that change might not have reached all of the other 19 yet. In other words you, as a data scientist, will have to devise some methodology whereby, when you interact with a specific node of a NoSQL data store, that you're aware of the fact that you might be dealing with stale data.

Eventual consistency. With NoSQL, consistency is not always current. With data propagation, all nodes will contain all data eventually. In other words, if we're going to grab data, we hope it's the most consistent. This means that the query results could be approximate and may not be exact.

Eventual Consistency in Action



DIFFERENCES BETWEEN THE ACID AND BASE MODELS

Aspect	ACID	BASE
Atomicity vs. Basically Available	Ensures complete transaction success or failure.	Ensures general database availability.
Consistency vs. Soft State	Maintains strict transaction rules.	Allows data flexibility and adaptability.
Isolation vs. Eventual Consistency	Processes transactions independently.	Promises data synchronization over time.
Durability	Guarantees transaction permanence.	Durability less emphasized.
System Design and Use Cases	Ideal for precise, reliable data transactions.	Suitable for scalable, high-traffic systems.

Now that we talked about some of the differences between relational and NoSQL stores, let's discuss the best applications for NoSQL databases.

When to consider using NoSQL database (NoSQL Store)?

- NoSQL is suitable for big data, for use in distributed environments such as clusters or clouds, and for data analytics. As of now, anything well in excess of a terabyte or two of data, is usually considered big data. So, if you have a set of data you're working with that's likely to exceed 1 or 2 TB, then it's probably considered big data and NoSQL might be a really good solution for that.
- NoSQL is appropriate when the emphasis is more on querying your data rather than inserting/updating your data. In this case, you may give full consistency up for high-performance queries.

- NoSQL is appropriate if it's acceptable for data to be de-normalized and schema-less, and if the data has a simple structure and low volatility.

Exercise: Can you think of your course projects that could be implemented using NoSQL store? and two others that relational database management system is the best fit for them?

At this point, you should have a good theoritical foundation about the basic concepts associated with NoSQL stores. Now, let's explore some common models of NoSQL stores.

Common NoSQL Data Models

The flexibility of NoSQL stores is associated with their various data models. You may combine multiple data models in one NoSQL solution. Common NoSQL data models include column-data, document-based, graph-based, and key-value data models.

1. Column-based Data Model

Think about a scenario where you have different measurements for each obersvation in your experiment. If you were to represent this in relational model, you would have to create your dataset into tabular model with irrelevant measurements populated as **N/A** for corresponding observations. Take a look at the following figures.

	Measurement-1	Measurement-2	Measurement-3	Measurement-4	Measurement-5
Observation-A					N/A
Observation-B					
Observation-C				N/A	
Observation-D				N/A	N/A

Fig. Sample Relational Model for Observations with Common/Special Measurements

Observation-A	Measurement-1 (Value)	Measurement-2 (Value)	Measurement–3 (Value)	Measurement–4 (Value)	
Observation-B	Measurement-1 (Value)	Measurement-2 (Value)	Measurement–3 (Value)	Measurement–4 (Value)	Measurement–5 (Value)
Observation-C	Measurement-1 (Value)	Measurement-2 (Value)	Measurement–3 (Value)	Measurement–5 (Value)	
Observation-D	Measurement-1 (Value)	Measurement-2 (Value)	Measurement–3 (Value)		

Fig. Corresponding Column-data Model for the Relational Model above

In column-data model, each column is essentially an aggregate of various values. Now because of that, each row does not have to have the same number of columns or even the same column types.

For example, I might have a situation where I'm storing employee data. **The first row or the first employee might have an aggregate that includes LinkedIn address. The second employee doesn't have a LinkedIn page so we don't even bother to add that value type to that particular row so that column won't have an aggregate for LinkedIn.**

Examples of data stores that use column-based model include: Accumulo, Cassandra, Druid, Hbase, and BigTable.

Now, if you look at our NSL-KDD dataset, it's structured dataset and the number of records is manageable so NoSQL won't be best solution for it. So, let's take another example where you have employee information. You have DOB for some of them while others you have only the HomeTown. In this case, you would represent it as column-data model in the following manner:

```
// column people {
 // row “smith-john" : {
     firstName: “John",
     lastName: “Smith",
     DoB: “6/1/1990“
 }
 // row “Martinez, juan" : {
     firstName: “Juan",
     lastName: “Martinez",
     HomeTown: “Chicago“
 }
}
```

Creating these records in hbase is as simple as specifying them directly in your insert statement.

```
hbase_table = {            # Table
  'row1': {                # Row key
    'cf1:col1': 'value1', # Column family, column, and value
    'cf1:col2': 'value2',
    'cf2:col1': 'value3'
  },
  'row2': {
    # More row data
  }
}
```

2. Document-based Data Model

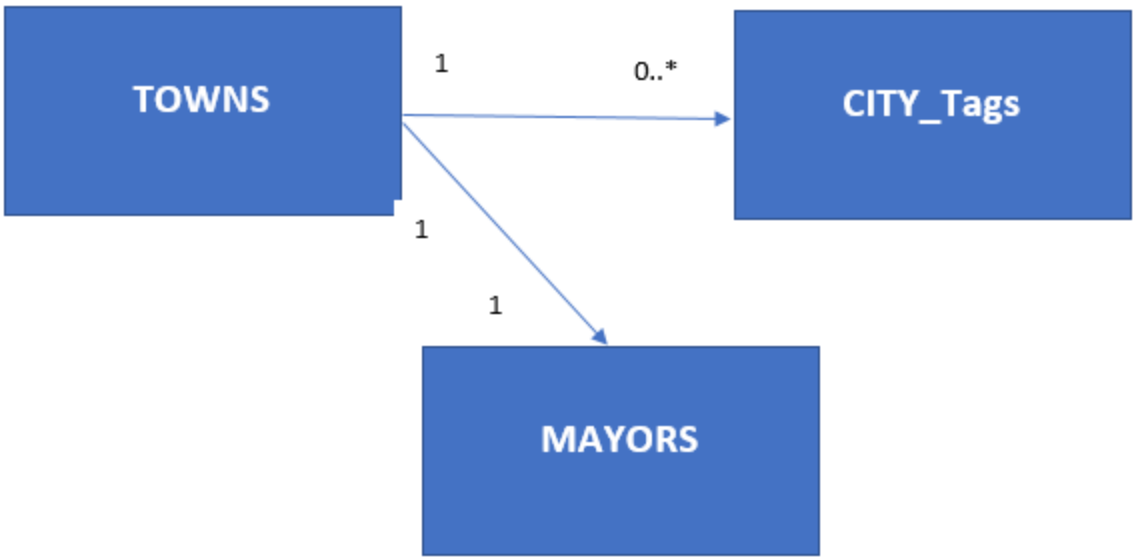
A document store actually has some sort of **document**.

- A document data store could use **any document type**: PDF, .DOC, HTML, or XML. They can also be data formats, like JSON and BSON. JSON is one of the more common used in document data stores because it so readily integrates with various programming environments so that it's easy to interact with the document.
- **For example, if your document data store depends on XML documents, then literally it stores XML files**. In addition, the particular implementation may extract metadata about that document and store that as well.
- Documents have to be addressed in data store via some **unique key** that represents that document. It can be a string, it can be almost anything, but a common mechanism is a **URL** that actually points to where the document is at. You will be able to interact with the files via **API** that each data store provides.

Document data stores can help you save effort in creating tables by creating this mongodb insertion that creates towns object and associate it with its mayor and famous sightseeings

```
> db.towns.insert({
  name: "New York",
  population: 22200000,
  lastCensus: ISODate("2016-07-01"),
  famousFor: [ "the MOMA", "food", "Derek Jeter" ],
  mayor : {
    name : "Bill de Blasio",
    party : "D"
  }
})
```

If we were to store the data from the previous example in a SQL database, we would have needed 3 different tables:

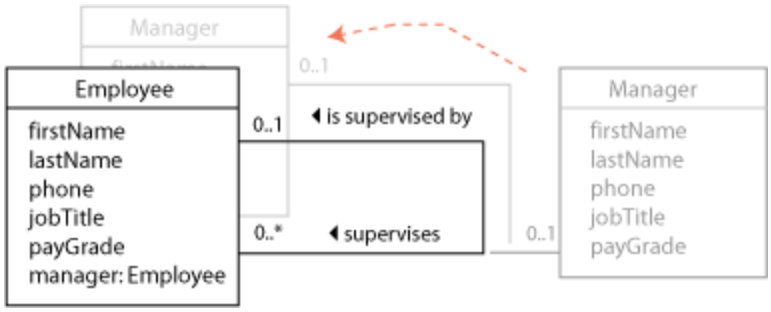


Examples of document data stores include MongoDB and CouchDB.

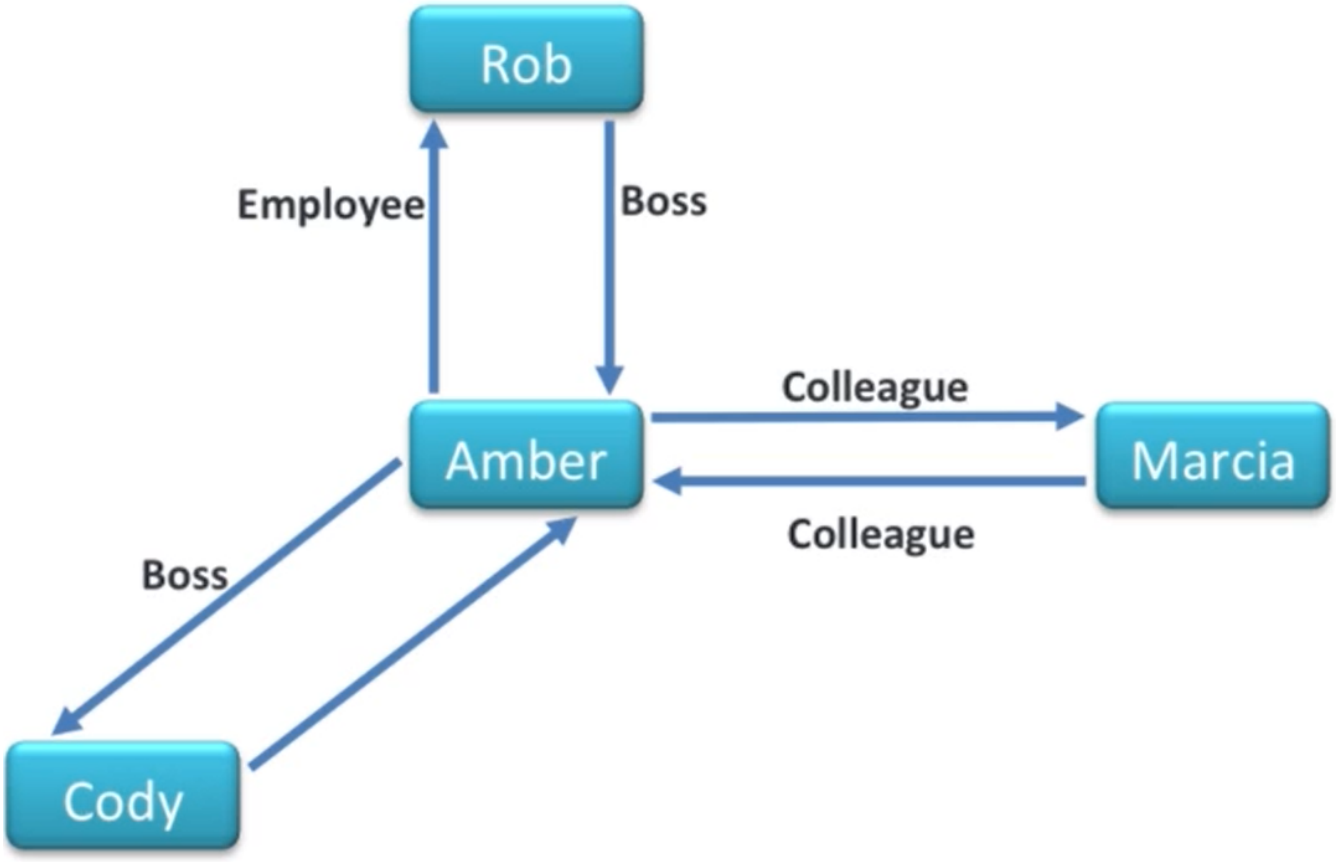
3. Graph-based Data Model

- Graph-based data models are inspired by the mathematical concept of graph theory.
- A **node** is some sort of entity, something in your database/store. For example, if I have an employee database/store, a specific employee would be a node.
- How do these nodes connect to each other? I have lots of employees, how does one employee connect to another? Is employee A, a subordinate of employee B, a supervisor of employee B, or a colleague of employee B? These connections are little lines drawn on a graph.

In PostgreSQL (or any other RDBMS), think about a scenario where you would like to simulate the relationship between employees and their supervisors. You would have had to build a complex model that leverages reflexive (or recursive) relationship like this:



Now, you could have represented this in graph-based model in much simpler way. Look at the following example:



The previous graph can be implemented in Neo4J using the following script:

```
Node Amber= graphDb.createNode();
Amber.setProperty(“name", "Amber");

Node Cody= graphDb.createNode();
Cody.setProperty("name", "Cody");

Amber.createRelationshipTo(Cody, Boss);
Cody.createRelationshipTo(Amber, Assistant);
```

Examples of NoSQL stores that use the graph-based model are Neo4j, FlockDB, Pregel, InfoGrid, and HypergraphDB.

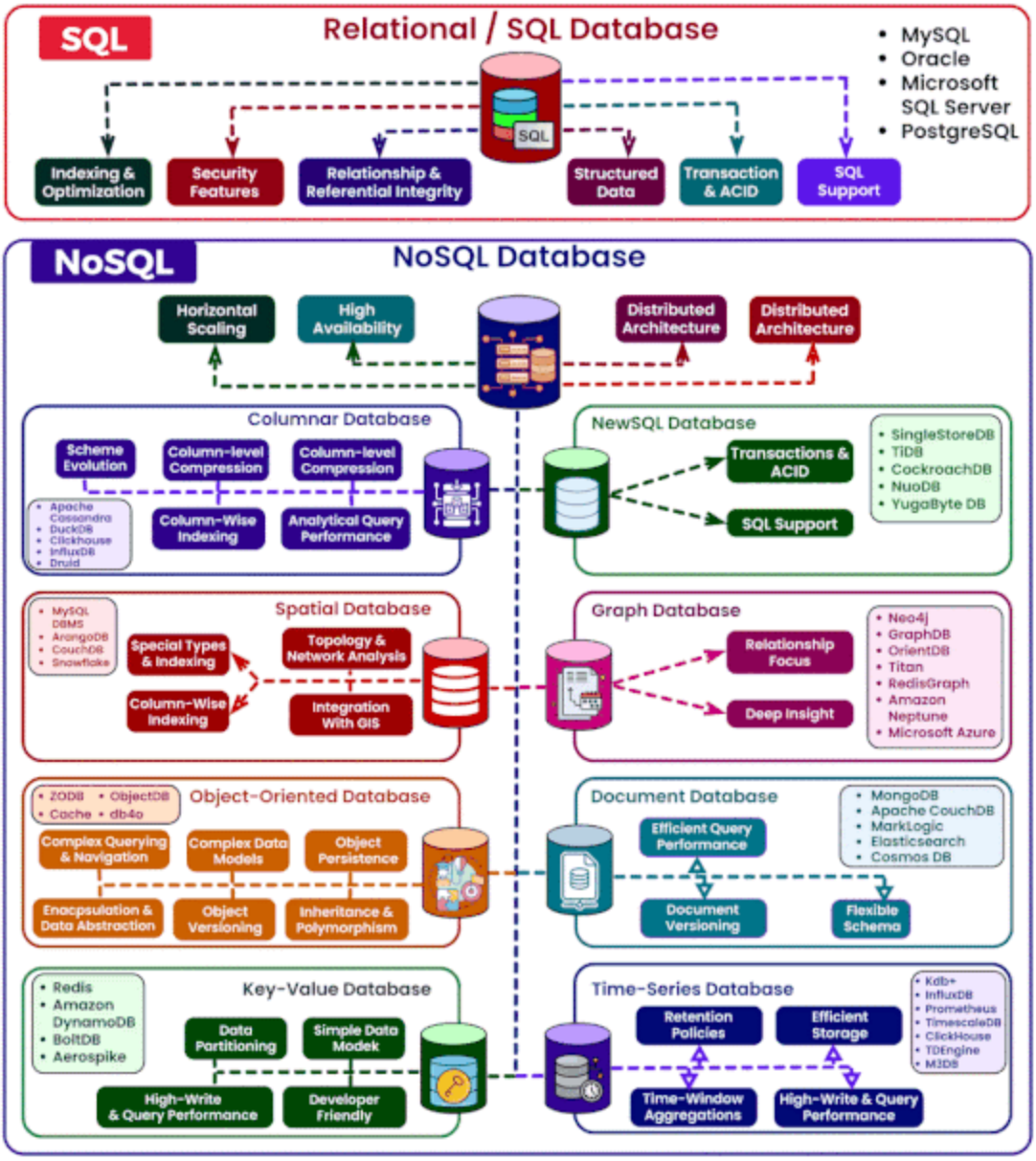
4. Key-Value Data Model

A key-value data store is simply a hash table where we connect values to a key. It uses an associative array, also known as a map or a dictionary, to connect these keys to the values and each key-value appears, at most, once in a data set.

There are a lot of examples out there including DynamoDB, BerkleyDB, BigTable, and Riak.

Example of a Key-Value Store

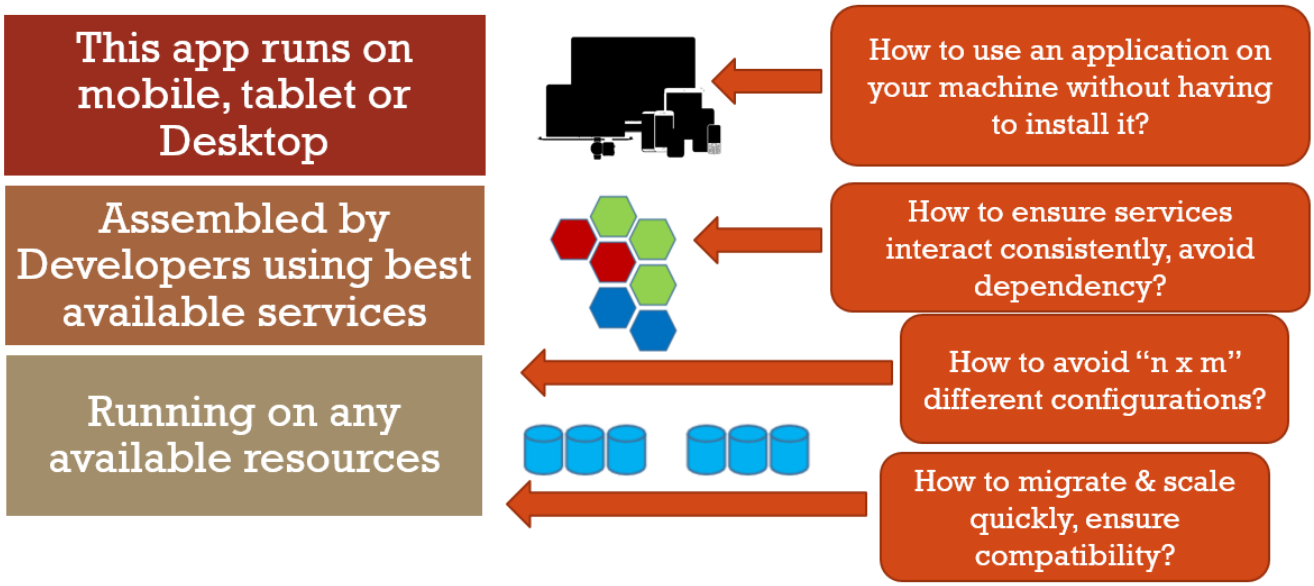
Types of Databases



In the next lecture, we will talk about Neo4J AuraDB and Google BigTable

Excerise: Install Docker to use Google BigTable.

What is Docker? and Why?



You can also explore thousands of applications that you can use via Docker containers rather than installing them on your machine from Docker Hub. Access Docker Hub repositories from here.

Run BigTable Locally via Docker

You can run a BigTable emulator Locally. To do so, you need to pull Google Cloud SDK Container and run it. More information can be found here

- Set the **BIGTABLE_EMULATOR_HOST** environment variable to **localhost:8086**. More information can be found on https://cloud.google.com/bigtable/docs/emulator
- Run the docker pull command for the image: **docker pull google/cloud-sdk**
- Execute the docker run command: **docker run -p 127.0.0.1:8086:8086 --rm -ti google/cloud-sdk gcloud beta emulators bigtable start --host-port=0.0.0.0:8086**
- In your terminal, run **docker container ls** or check your docker desktop to see if the container is running

Install Required Packages

In [ ]:
```
!pip install google-cloud-bigtable
!pip install google-cloud-happybase
```

ⓘ Note that you will have to create free account on Docker Hub in order to be able to create custom Docker images - if interested-.

Readings

- Graph DB vs RDBMS, published on Canvas
- Application of Graph DB, published on Canvas