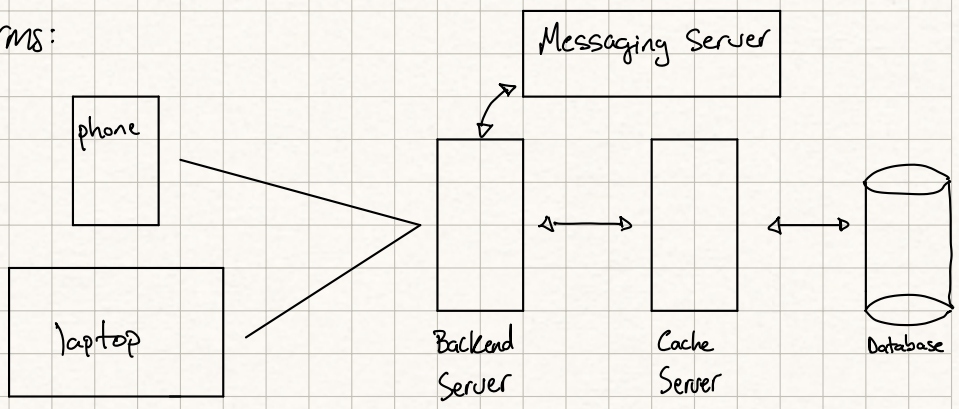
A database is a component in application architecture required to persist data

Data can be of many forms:

- structured
- unstructured
- semi-structured
- user-state
- ... data



UI runs on the client

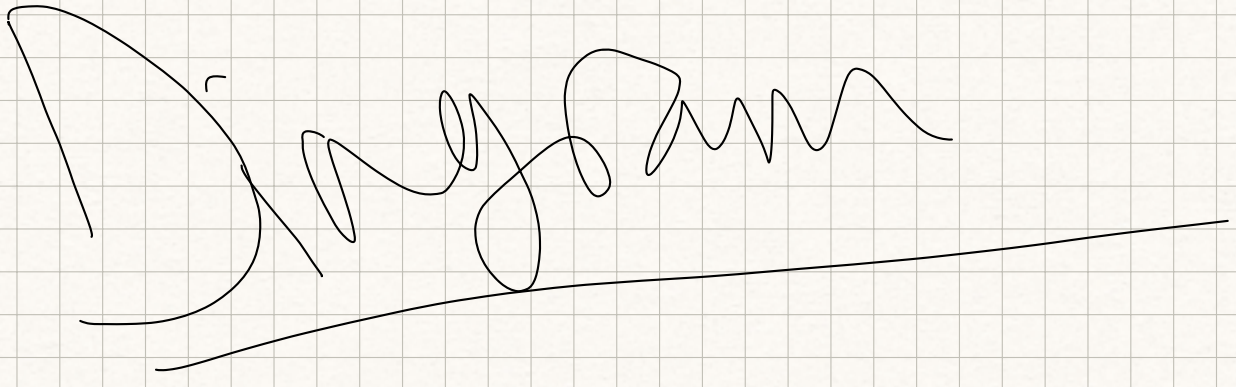
* Web Application Architecture

Structured data

- ***Structured Data*** is the type of data that conforms to a certain structure, typically stored in a database in a normalized fashion
- **Structured data is the most easy** to work with since it does **not** need any sort of data preparation before we can interact with it
- An example of this type of data is the details of a customer stored in a database row
  - The *customer id* would be of *integer type,*
  - the *name* would be of *string type* with a certain character limit,
  - *age* would be of *integer type* and so on
- Every *column* of the database *row* has some pre-defined rules for the data that is meant to be persisted in it.
- With structured data, we know what we're dealing with. Since the customer name is strictly of string type, we can run string operations on it without worrying about errors or exceptions
- **Structured data** is typically managed by a query language such as SQL (structured query language)

Unstructured Data

- Unstructured data has no definite structure. It is the heterogeneous type of data consisting of *text, image files, videos, multimedia files, pdfs, blob objects, word documents, machine-generated data, etc.*
- We generally have to deal with this kind of data when running *data analytics*. In data analytics architecture, the data streams in from multiple sources such as *IoT devices, social networks, web portals, industry sensors, etc., into the analytics system*
- We **cannot** directly interact with unstructured data. The initial data collected is **pretty raw.**
    - We have to make it flow through a data preparation stage that segregates it based on business logic and then analytics algorithms are run to extract meaningful information



Semi-Structured Data

- Semi-Structured Data is a mix of structured and unstructured data
- This data is often stored in data transport format such as **XML, JSON** and handled as per the business requirements

User State

- *User state* data is the data containing the information of activity the user performs on the website
- For instance, when browsing through an e-commerce website: the user typically browses through several product categories, sorts the products based on different parameters, clicks on the recommended products, adds a few of them to the wishlist and the availability notification list, and so on.

All this activity is the user state…

  - storing user state helps businesses improve the user browsing experience and the conversion rate on their website
  - Also, persisting the state enables the users to continue from where they left off when they log in next
  - It does not fell like they are starting fresh on a website

Relational Databases

What is a relational database?

- A *relational database* persists data containing relationships:
  - One to one
  - One to many
  - Many to many
  - Many to one, etc.
- Relational databases have a relational data model, data is organized is *tables* having *rows* and *columns* and *SQL is the primary data query language* used to interact with relational databases.
- **MYSQL** is an example of a relational database

What are relationships?

- Imagine you buy 5 different books from an online bookstore.
  - When you create an account at the bookstore, the system will assign you a customer id say: C1
  - Now C1 will be linked to 5 different books: b1,b2,b3,b4,b5
    - This is a one to many relationship
    - In the simplest of forms, one database table will contain the details of all the customers and another table will contain all the products in the inventory
  - Upon pulling the user object with the id C1 from the database, we can easily find what books C1 purchased via the relationship model

Data Consistency

- Besides the relationships, **relational databases also ensure saving data in a normalized fashion**.
  - In very simple terms, **normalized data means an entity occurs in only one place/table in its simplest and atomic form and is not spread through the database**
  - This helps maintain consistency in the data
    - In the future, if we want to update the data, we update it in JUST one place as opposed to updating the entity spread through multiple tables
    - This is troublesome, and things can quickly get inconsistent

ACID Transactions

- Besides ***normalization*** and ***consistency***, relational databases also ensure *ACID* transactions
- ACID stands for:
  - **Atomicity**: each statement in a transaction (to read, write, update, or delete data) is treated as a single unit. Either the entire statement is executed or none of it is executed. This helps prevent data loss and corruption
  - **Consistency**: ensures that transactions only make changes to tables in predefined, predictable ways.
  - **Isolation**: When multiple users are reading + writing from the same table all at once, isolation of their transactions ensures that the concurrent transaction don't interfere with or affect one another.
  - **Durability**: ensures that changes to your data made by successfully executed transactions will be saved, even if there's a system failure.
- An *ACID* transaction means if a transaction, say a financial transaction, occurs in a system, it will be executed with perfection without affecting any other processes or transaction.
- After the transaction is complete, the system will have a new state that is ***durable*** and ***consistent***

- In case anything amiss happens during the transaction, say a minor system failure, the entire operation is rolled back
- An *ACID* transaction happens with an initial state of the system, *STATE A, and completes with a final state of the system, STATE B.*
    - *Both the states are consistent and durable*
- *A relational database ensures that the system is either in STATE A* or *STATE B at all times.*
    - There is no middle state
    - If anything fails, the system always rolls back to *STATE A*

Relational Databases are great **IF:**

1. You need **strong consistency**
2. **Transactions**
3. **Relationships**

Typical examples of apps needing *strong consistency* are *stock trading, personal banking, etc.,* and relational data is common like *Facebook, LinkedIn, etc.*

Transactions and data consistency

If you are writing software that has anything to do with money or number that makes transactions, ACID and data consistency super important to you

Relational DBs shine when it comes to *transactions* and *data consistency*

They comply with the ACID rule, have been around for ages, and are battle-tested

Storing Relationships

- If your data has a lot of relationships that we typically come across in social networking apps
    - Like what friends of yours live in a particular city
    - Which of your friends already ate at the restaurant you plan to visit today
        - Relational databases suit well for this kind of data
    - Relational databases are built to store relationships

Popular Relational Databases

1. MySQL - an open source relationship database written in C and C++
2. PostgresSQL, an open source RDBMS written in C
3. Microsoft SQL Server, a proprietary RDBMS written by Microsoft
4. MariaDB, Amazon Aurora, Google Cloud SQL, etc

What is a NoSQL Database?

As the name implies, NoSQL databases have **no SQL**.

They are more JSON-based databases built for Web 2.0

NoSQL databases are built for high-frequency **read-writes**, typically required in social applications like micro-blogging, real-time sports apps, online massive multiplayer games, and so on.

How is a NoSQL database different from a relational database??

- why the need for NoSQL databases when relational databases were doing fine, battle-tested, well adopted by the industry and had no major persistence issues ?

Scalabilty

1. A big limitation with SQL-based relational databases is **scalability**
    A. Scaling relational databases is not trivial
        a. They have to be *shared, replicated* to make them run smoothly on a cluster
        b. This requires careful planning, human intervention and a skill set
2. Whereas with NoSQL databases, you can add new server nodes on the fly **and** scale without any human intervention

Todays websites need fast read-writes. There are billions of users connected with each other on social networks. A massive amount of data is generated every microsecond, and we need an infrastructure designed to manage this **exponential growth**

Ability to run on clusters

- NoSQL databases are designed to run intelligently on clusters, very minimal human intervention
- Today, the server nodes even have self-healing capabilities. The infrastructure is intelligent enough to self-recover from faults
- **However,** all this innovation does not mean old-school relational databases aren't good enough, and we don't need them anymore
    - Relational databases still work like c harm and are still in demand. They have a specific use case
- Also, NoSQL databases had to sacrifice strong consistency, ACID transactions, and much more to scale horizontally over a cluster and across the data centers
- The data with NoSQL databases is more *eventually consistent* as opposed to being ***strongly consistent***
    - This is completely fine
    - We don't need silver bullets, we are up to a mich harder task connection the world online

Pros of NoSQL DB

Besides their scalable design, NoSQL databases are also developer-friendly

Learning curve not so steep and schemaless

- the learning curve of NoSQL databases is less steep than that of relational databases
- When working with relational databases, a big chunk of our time goes into learning to:
    - design well-normalized tables
    - setting up relationships
    - Trying to minimize joins, and so on
- Relational databases are like drill sergeants. Everything has to be in place, neat and tidy and things **need** to be consistent
- NoSQL databases are a bit chilled out and relac
    - There are no strictly enforced schemas
    - You can work with the data however you want
    - You can always change stuff and move things around
    - Entities have not
        - Thus, there is a lot of flexibility
    - While this flexibility is good, it can also be bad…

Cons of NoSQL Databases

- Since the data is not normalized, this introduces the risk of it being inconsistent.
- An entity, since spread throughout the database, has to be updated at all places. It's hard for developers to remember all the locations of an entity in the databases -> leading to inconsistency
- Failing to update an entity at all places makes the data inconsistent
    - This is not a problem with relational databases since they keep the data normalized

No support for ACID transactions

Also, NoSQL distributed databases don't support ACID transactions

A few claim to do so, but not a global deployment level

ACID transactions in these dbs are limited to a certain entity hierarchy or a small deployment region where they can lock down nodes to update them

Things are comparatively simple because there is no stress of managing joins, relationships, n+1 query issues and so on.

Just fetch the object using its key, which is a constant O(1) operation, making the NoSQL databases fast and simpler

Popular NoSQL databases: MongoDB, Reid's, Neo4J, Cassandra

When to pick a NoSQL database?

- NoSQL databases are built to handle a large number of read-write operations due to the eventual consistency model.
    - With the ability to add nodes on the fly, they can handle more concurrent traffic, enabling us to scale fast
- They are also built to handle big data with minimal latency.
- Pick a NoSQL databases if you're looking to scale fast and willing to give up on strong consistency

Flexibility with data modeling

A NoSQL DB is a good fit if you are not sure about your data model during the initial phases of development and things are expected to change at a rapid pace.

NoSQL databases offer us more flexibility

Eventual consistency over strong consistency

- NoSQL databases are a good pick, **when** we do not need ACID transactions **and** are okay to give up **strong consistency**
- A good example of this is a **microblogging** site like **Twitter**
    - When a celebrity's tweet blows up and everyone likes and re-tweets it from across the world, does it matter if the like-count goes up or down a tad bit for a short while ?
    - The celebrity wouldn't care if instead of the actual 5.5 million likes, the system displayed the like-count as 5.25 million for a short while
- When a large application is deployed on hundreds of servers spread across the globe, the geographically distributed nodes take some time to

reach a global consensus

- Until they reach a consensus, the value of the entity inconsistent
  - The value of the entity **eventually becomes consistent** after a short while. This is what **eventual consistency is**
  - However, inconsistency does **not mean any** sort of data loss
    - It just means that the data **takes a short while to travel across the globe via the internet** cables under the ocean to reach a global consensus and become consistent
  - We experience this behavior all the time, especially on Youtube.
    - Often you might see a video with 10 views and 15 likes… how is this even possible?
  - It's not. The actual views are already more than the likes. It's just the count of views is consistent and takes a short while to get updated. I will discuss *eventual consistency*.

Running Data Analytics

- NoSQL databases also fit best for data analytics use cases, where we have to deal with an influx of massive amounts of data
- There are dedicated databases for use cases like this, such as
  - *Time-series databases*
  - *Wide-column*
  - *Document-oriented*

Is NoSQL More Performant than SQL?

- **NO**
- From a technology performance benchmarking standpoint, both relationship and non-relational database are equally performant
- More than the technology, **it's how we design our systems using a certain technology that decides the performance**
- Both SQL and NoSQL tech have their use cases

Real-World Case Studies

- **Facebook** uses **MySQL** for storing its social graph of millions of users. Although it did have to change the DB engine and make some tweaks, **MySQL** fits best with its use case
- Quota uses **MySQL** pretty efficiently by partitioning the data at the application level.

Using BOTH SQL and NoSQL databases in an application

- Why can't I use both ?
- You can, all the large-scale online services uses a mix of both to achieve the desired persistence behavior.
- The term for leveraging the power of multiple databases clubbed together is: **polyglot persistence**