**Databases**

Choosing the right database for your System Design

Some of the things that we'll cover in this module are:

1. What are some common and widely used databases
2. In which scenario you should choose what kind of a database

Databases! How do you know it’s the right one?

Whether you are a software engineer designing a new product, a student working on a project, or a job seeker preparing for your next system design interview, choosing the right storage solution always requires some consideration.

In this article, we will be discussing various storage solutions available in the market and which will be suitable in which scenarios.

First of all, databases will not impact your functional requirements. Whichever database you use, you can still achieve your functional requirements somehow, but at the cost of huge performance degradation. So when we say requirement, we usually mean non-functional requirements.

Factors

* Structure of the data
* Query pattern
* Amount or scale that you need to handle

These are the factors we need to consider when selecting which database to use. Now let us look at various types of storage solutions and some use cases where they will be suitable.

Caching Solution

If you are calling your database very frequently or making a remote call to independent services with high latency, you might want to cache some data locally at your end. Some of the most commonly used caching solutions are **Memcached, Hazelcast, and Redis**. You could also use some other solutions; this is not an exhaustive list. In the following articles, we will usually use **Redis** as it is one of the most widely used and stable solutions.

File Storage Solution

Assume you are working on something like Netflix and you need a data store for images, videos, etc. Now, in this case, a database is not very useful to us as we are storing files rather than information. Databases are meant to store information that can be queried, whereas files you do not need to query. You just deliver them as they are.

This is when we use something called **Blob (Binary Large Object) storage. Amazon S3** is an example of blob storage. Usually, blob storage is used in combination with a **Content delivery network** or a **CDN**. A CDN is a network of servers around the world that delivers content in different geographical locations with reduced latency. If the server you are getting content from is closer to your geographic location, the content will take less time (reduced latency) to be delivered from the server to you.

Storage Solutions Offering Text Search Capability

Let’s again take the Netflix example. Suppose you want to build a search functionality where the user can search by movie, genre, actor, actress, director, etc. Here you use a search engine like **Solr** or **Elasticsearch** which can support **fuzzy search**.

To understand fuzzy search, let us take an example of an Uber user searching for airprot. If you notice this is a typo, what the user means to search is airport. But if, because of this typo, we don’t provide any search results, it will be a very poor user experience. So we search for terms similar to airport in the database. This is known as fuzzy search.

Now a key point here is that these search engines are not databases. Databases provide a guarantee that once stored, our data will not be lost unless we delete it; search engines offer no such guarantee. This is why we should never use search engines like Elasticsearch as our primary data source. We can load the data to them from our primary database to reduce search latency and provide fuzzy and relevance-based text search.

Time Series Database

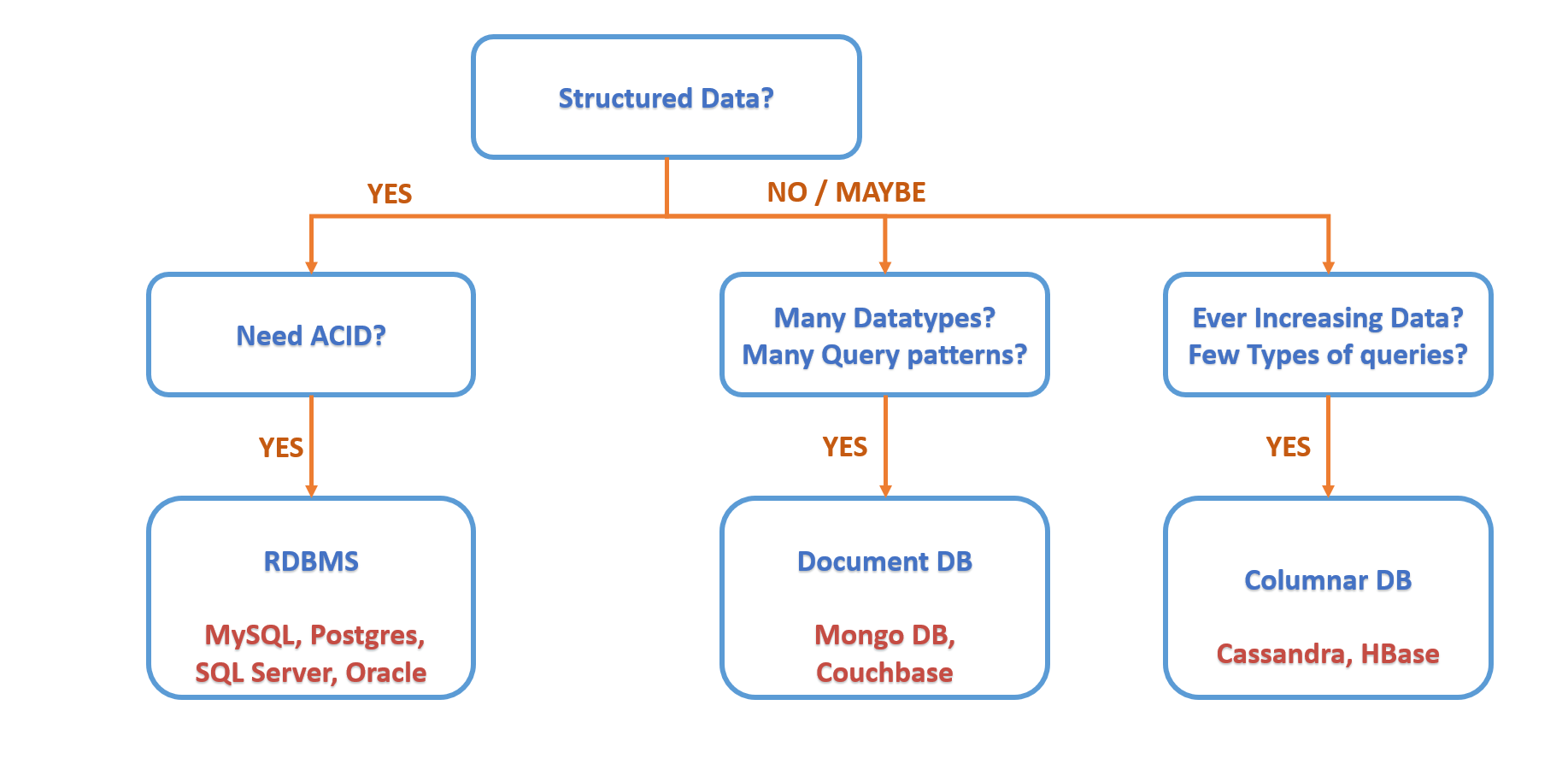
Suppose we are trying to build a **metric tracking system**. We will need something called a time-series database. Time-series databases are, in a way, an extension of relational databases, but unlike a standard relational DB, time-series databases will never be randomly updated. It will be updated sequentially in an append-only format.

Also, it will have more bulk reads for a certain time range as opposed to random reads. For example, how many people watched a video in the last 1 week, 10 days, 1 month, 1 year, and so on. Some examples of time series databases are **OpenTSDB and InfluxDB**.

Data Warehousing Storage Solution

Sometimes we need a large database to dump all of the data available to us, to perform analytics. Eg. a company like Uber will store all of their data so they can perform analytics to identify where Uber is not used very much, where are the hotspots, what are the peak hours, etc. These systems are not used for regular transactions but offline reporting. **Hadoop** is a very commonly used Data warehouse.

When to Use SQL or NoSQL



As mentioned at the beginning of the article, structure is one of the factors we use to identify what type of database we need to use. If we are storing structured information, or information that can be represented in a tabular format, we can use a relational database. Along with this, we will also consider if we need the database to be atomic, consistent, isolated, and durable (ACID).

* **Atomicity** means that you can guarantee that all transactions will either succeed completely or not at all. It’s all or nothing.
* **Consistency** means that you can guarantee the database remains in a consistent state before and after the transaction. This concerns the correctness of the data.
* **Isolation** means that all transactions will occur in isolation, one transaction will not be affected by another ongoing parallel transaction. This means the database should be able to process concurrent transactions without leading to inconsistency.
* **Durability** means that once a transaction has been completed, the changes are permanently written to disk, and won’t be lost in case of a system failure.

If you need ACID properties, then you need to use a relational DBMS. Some examples are **MySQL, Oracle, and Postgres**. But what if you don’t need ACID? Well, you can still use RDBMS, or you can use a **Non-relational database**.

Let's consider an example. Suppose you are trying to build a catalog for something like Amazon, where you want to store information about different products that have various attributes. These attributes will normally not be the same for different products. For instance, medicines will have an expiry date, but refrigerators will have energy ratings. In such a case, our data cannot be represented as a table. This means we need to use a **NoSQL** database.

Also, we don’t just need to store this data but also query on this data. Here comes the factor of **query pattern**. Which type of database we use here will be decided based on what type of data we store and what types of queries will be run on it. If we have vast data - not just volume but also a vast variety of attributes - and we need to run a vast variety of queries, we need to use something called a **Document DB**. **Couchbase** and **MongoDB** are some commonly used document databases.

**Fun fact**: Elasticsearch and Solr are special cases of document DBs.

But what if you don’t have a vast variety of attributes i.e. very limited variety of queries, but the size of the database increases very rapidly? For example, data collected by Uber for their drivers’ location pings. Now the number of Uber drivers will keep increasing day by day, and therefore so will the data collected every day. This results in an ever-increasing amount of data. In such cases, we use **Columnar DBs** like **Cassandra** or **HBase**.

There are some other alternatives as well, but these are the most widely used, tested, and stable ones. In most of the following solutions, you will notice we use Cassandra as it is lighter to deploy, whereas HBase is built on top of Hadoop; we would need to first set up Hadoop and then setup HBase on top of it. This makes the setup of HBase a little lengthy, but performance-wise both are pretty much the same. Basically, the idea is that if you have a huge scale of queries but less variety in queries, and most of your queries are such that you can include a common partition key in the where clause for each of the queries, then Cassandra works beautifully!

Okay, that is confusing! So let's look at this with an example. Let us assume we have stored Uber's ride-related data in a Cassandra with driver id as a partition key. Now when we want to fetch a ride for a particular driver on a particular date, Cassandra can find it easily enough. But if we want to find a customer's ride on a particular date, Cassandra will have to fan out this query to all the partitions since customer id is not a partition key. So what is the point of using Cassandra if it is not going to scale well!

Well, there is a simple enough fix. We can replicate the same data to another table or **column family** with a different partition key. Now when we receive the query for customer id and date, we can simply direct it to the table where the partition key is customer id. This is what we mean by a limited variety of queries but a huge scale. Cassandra (and HBase) can scale massively as long as the queries are of similar types.

If the queries are more diverse, then we will have to replicate again and again for each partition key, which we can, but only to a certain limit. If we cannot control the types of queries, then something like MongoDB might be the way to go. But if we just need a huge scale for a few types of queries, the Cassandra is the perfect solution.

Now, these are all happy case scenarios. If we have structured data and need ACID properties, use a relational database like MySQL. If we have a huge amount of data with a lot of attributes, we can use a document DB like Mongo DB. If we have simpler data with less diverse queries, we use columnar databases like Cassandra. But in real-world scenarios, our requirements are usually not so simple.

Now let’s shake things up a bit!

Let's consider the Amazon example again. If for a product we have only one item in stock but multiple users are trying to buy it, it should only be sold to one user, which means we need ACID here. So we should choose a Relational DB like MySQL.

But the orders-related data for Amazon will be ever-increasing and will have a variety of attributes. That means we should use a Columnar NoSQL database like Cassandra. So which one to go for? We decide to go with a combination of both. We can store the data of orders that are not yet delivered in a MySQL database, and once the order is completed, we can move it to Cassandra to be permanently stored.

But again, our requirements might be a little more complex. Suppose you want to build a reporting system for how many people bought a particular item. Now, on Amazon, products are sold by various users of different brands and different variations. So the report can not target a single product, rather, it should target a subset of products, which can be in either Cassandra or MySQL.

Such a requirement is an example of a situation where our best choice would be a document DB like Mongo DB. So we decide to keep a subset of this orders data in Mongo DB that tells us which users bought how much quantity of a certain product, at what time, on what date, etc. So suppose you want to check how many people bought sugar in the last month. You can get order ids from Mongo DB and use this order id to pick up the rest of the data from Cassandra or MySQL.

That should be it for Storage Solutions in System Design!