Show Submission Credentials

P3. NoSQL Primer An introduction to the NoSQL databases

16 days 2 hours left

✓ Introduction to NoSQL

Introduction to NoSQL

Why NoSQL?

The choice of NoSQL and SQL databases boils down to the requirements of the application and the nature of the data. NoSQL databases are designed as an addition to the field of databases and not as a replacement of SQL databases.

Motivation for industry

The scale of users, devices, and data continues to increase significantly. The challenge and competitive advantage are to provide the best possible experience for users or customers. Technology has to adapt to offer experiences that correspond to the evolving behavior of users.

“Walmart saw a sharp decline in conversion rate when page load time increased from one to four seconds. (Source: Walmart Labs)”

“Amazon noticed a 1% drop in sales for every 100ms of latency cost”

The browsing and shopping experience on Amazon is another good example because it can cost Amazon millions if they do not provide performant solutions. Furthermore, Amazon observed that many users prefer low latency over accuracy. They are happy to refresh their shopping cart but are not willing to tolerate a slow loading page. Hence, occasionally having an inaccurate but fast-loading shopping cart is favored over a shopping cart that is slow to load with accurate results.

Let’s look at some of the requirements of this competitive and fast-evolving market that has driven the development of the NoSQL paradigm where traditional RDBMSs fall short.

For clients/customers

- Provide high availability and scalability
- Process a large amount of data
- Be geographically aware

Customers expect a service provided by web services to be available at all times irrespective of whether it is a Black Friday or a Cyber Monday. With the advent of mobile phones, social networks and the Internet of things (IoT), large amounts of data are being generated. It is also important to provide customers with the same Quality of Service (QoS) irrespective of the fact that the user is in any part of the globe. Relational databases are generally deployed on a single server and the available resources such as memory, processing power, and storage, are fixed. Hence, bigger servers need to be added to meet the scaling requirements (vertical scaling) which tend to be costlier. However, it is just a matter of adding more servers to scale out for NoSQL databases (horizontal scaling).

For Developers

- A flexible data model or schema
- Agile development

Unlike relational databases that require structured schemas, NoSQL does not impose restrictions on the structure of the data. This provides for an Agile Development process and helps to achieve fast development and deployment lifecycles. We will discuss the flexible data model at a later section of this primer.

For Business owners

- Move on from costly enterprise solutions to open source solutions
- Exploit commodity hardware and the power of cloud computing resources in a cost-effective way

Due to the heavy increase in users, business owners have to think about how to manage the operational cost of their business. Traditional database solutions tend to be costlier when businesses need to scale as their data or user base grows.

References:

https://www.couchbase.com/binaries/content/assets/website/docs/whitepapers/nosql-for-retail-and-ecommerce-couchbase.pdf (https://www.couchbase.com/binaries/content/assets/website/docs/whitepapers/nosql-for-retail-and-ecommerce-couchbase.pdf)

https://www.mongodb.com/nosql-explained (https://www.mongodb.com/nosql-explained)

http://robertgreiner.com/2014/08/cap-theorem-revisited/ (http://robertgreiner.com/2014/08/cap-theorem-revisited/)

http://annovate.blogspot.com/2014/06/motivation-to-nosql-database.html (http://annovate.blogspot.com/2014/06/motivation-to-nosql-database.html)

The Challenge to Achieve Scalability

In the cloud era, one of the major features is scalability to satisfy the increasing workload. Vertical scaling (a.k.a. scale-up) is limited by the capacity of one single machine and does not align with the need of the distributed systems. Horizontal scaling (a.k.a. scale-out) is more practical to scale dynamically over time by adding more resources to the distributed systems. However, scaling out introduces additional complexity due to the distribution of an application or a system onto multiple networked machines, which could encounter intermittent or permanent failure. This complexity and necessity trade-off is best stated by the CAP theorem.

CAP Theorem

CAP theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees.

Consistency. Every read receives the most recent write **or an error**, but it is guaranteed that **no stale data** will be returned. All nodes will return the same result at the same time regardless of the execution of any operation.

Availability. The system is always available **with no downtime**. Every request receives a non-error response, which can be either the most recent write **or stale data**.

Partition Tolerance. Partition means the nodes are partitioned into multiple groups that cannot communicate with every other group **because of network failure**. For example, the network between group A and group B is cut off, or all the nodes in group B are down. Partition tolerance requires that during a network failure or partition between nodes, the system continues to operate.

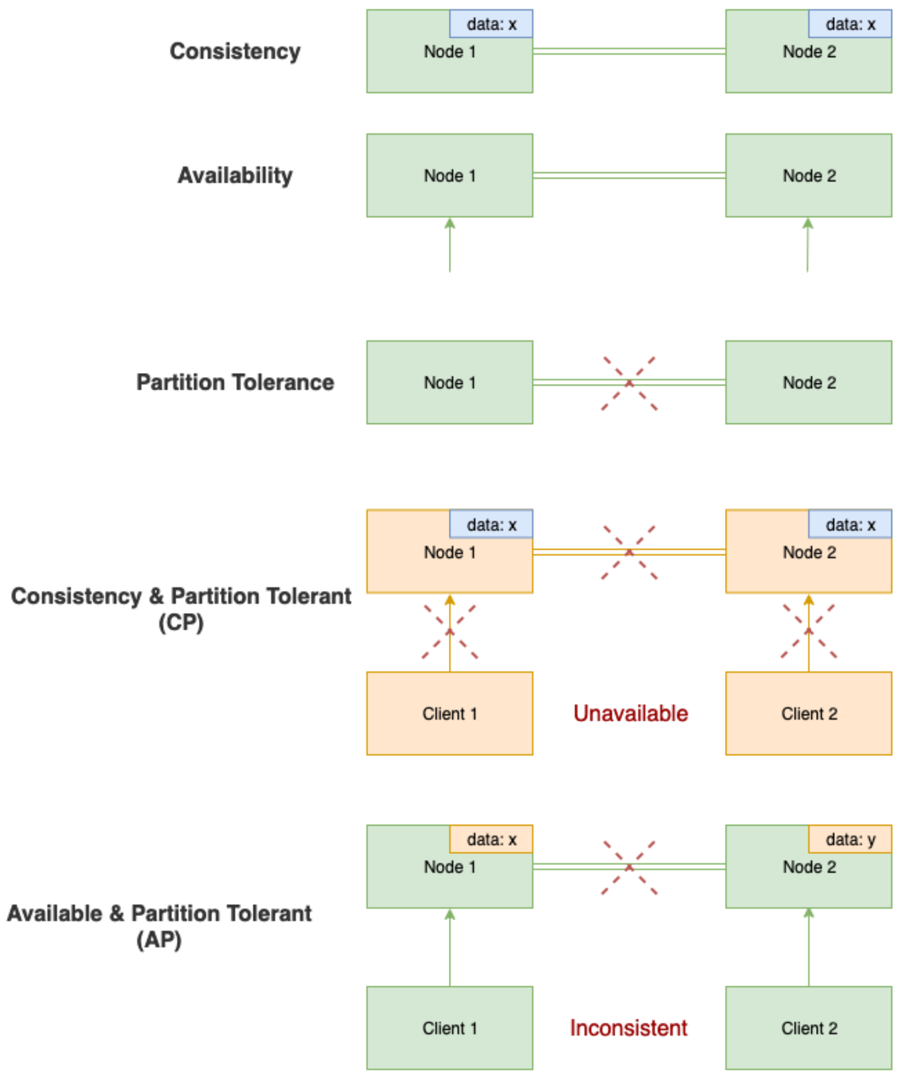

Figure 1: CAP Theorem Diagram

The CAP theorem applies when a network partition or failure happens, instead of at all times. If there were no network failures, both availability and consistency can be satisfied and there will be no trade-off between them. However, network failures are unavoidable for any distributed system, hence, network partitioning has to be tolerated.

If partition happens for a distributed data store, one has to make the trade-off between consistency and availability. During the network partitioning, if consistency is chosen over availability, the system will return an error or a timeout if particular information cannot be guaranteed to be the most recent and the clients will experience downtime; if availability is chosen over consistency, the system will return the most recent available version which can be stale.

Examples

Guarantees	Examples	Trade-off

CA	traditional relational databases e.g. MySQL, PostgreSQL	non-distributed
CP	HBase, MongoDB, Memcached, Redis, Azure SQL	downtime
AP	Cassandra, Amazon DynamoDB	stale data

Traditional SQL databases essentially do not meet the distributed feature, and NoSQL databases become the solutions. The trade-off is only between consistency and availability if the datastore is distributed.

Information

Quick Notes

CAP theorem is not an unconditional "two out of three" dilemma. It is a conditional trade-off when a network partition happens for a distributed system.

Keywords in CAP Theorem:

Consistency: no stale data

Availability: no downtime

Partition Tolerance: network failure tolerance in a distributed system

Low Latency v/s Strong Consistency

A network failure may be a rare event, but there is always network delay.

The latency between nodes can be regarded as a temporary partitioning. It will cause a temporary trade-off between consistency and availability. The CAP theorem indicates that a distributed data store (`P`) that guarantees low latency (`A`) has to lose consistency and allow stale data (`C`). If stale data is tolerable and low latency is what you care about, such as item recommendations on Amazon, you can choose the `AP` model with loose consistency such as eventual consistency. Eventual consistency means that stale data can be tolerated.

One of the best examples of the `AP` model to achieve low latency is Amazon DynamoDB. Amazon DynamoDB is "consistently single-digit millisecond latency at any scale" with eventual consistency.

You will explore the motivation and implementation of different consistency models in future projects.

Flexible Data Model

As we discussed above, traditional SQL databases are more suitable for structured data and sacrifice performance in favor of consistency. However, NoSQL databases offer the flexibility to handle unstructured data and also provide faster read and write to a large number of concurrent users by scaling horizontally and tolerating stale data. Furthermore, even when your system is not distributed, NoSQL databases can meet a set of practical requirements when SQL databases fall short.

Suppose you are going to build a fancy multimedia social website. Users can create posts with images and videos. Other users can add comments and vote on the posts. You are probably already thinking of a blueprint for the SQL tables. For example, one can start by creating the following tables: `User` , `Post` , `Comment` , `Image` , `Video` , `Vote` .

To achieve the core feature of the social website, you want to show each post with all the associated images, videos, music, comments, and votes. You notice that you have to perform a complex query joining all the tables and combining values to retrieve the result.

If you only care about the retrieval of the video addresses and it is not valuable to perform a table-specified aggregation analysis like "find the most popular video", is it worthwhile to maintain discrete tables like `Image` and `Video` ? However, you will have to do so if you want to conform to RDBMS's rigid approach and normalize the rules of the relational database to organize data, which has no tolerance for duplicate data and you have to eliminate any duplicate data by decomposing tables. SQL can be strict and inflexible for different scenarios.

Another example is when your website evolves and you need to update your data model or schema. For example, if you need to support audio in the posts, or if users demand that each image should have an optional title, etc.

Because of such scenarios, people started exploring other options. NoSQL databases attempt to address these issues. As a quick example, you can just store each post as a collection in MongoDB, a NoSQL document store, for quick retrieval:

```
{
    "_id": ObjectId("54c955492b7c8eb21818bd09"),
    "title":"post title",
    "date":"2016-04-30",
    "text":"My heart is on the cloud",
    "user":"boo",
    "images":[
        "http://social.s3.amazonaws.com/images/1.png"
    ],
    "videos":[
        {
            "url":"http://social.s3.amazonaws.com/videos/1.mp4",
            "title":"The first video sample"
        },
        {
            "url":"http://social.s3.amazonaws.com/videos/2.mp4",
            "title":"The second video sample"
        }
    ],
    "votes":12,
    "comments":[
        {
            "text":"the first comment",
            "date":"2016-05-01",
            "user":"far"
        },
        {
            "text":"the second comment",
            "date":"2016-05-01",
            "user":"boo"
        }
    ]
}
```

When you need to change the data format, you can simply adopt the changes with new posts in the future without needing to change your data model. As you can see, relational databases play an important role in many scenarios and are widely adopted in most organizations. However, there are also scenarios when alternative solutions are expected. These are driven by the need to:

- Handle unrelated, indeterminate or evolving data.
- Achieve faster time to market, enabled by agile development and flexible data models.
- Handle rapidly growing volumes of structured, semi-structured and unstructured data.
- Scale beyond the capacity constraints of existing systems.
- Free yourselves from expensive proprietary database software and hardware.

To meet these requirements, companies are building operational applications with NoSQL databases.

Basic Types of NoSQL Databases

Type	Definition	Example	Applications

Schema-less Key-Value Stores	The least complex NoSQL model which stores data in a unstructured schema-less way that consists of indexed keys and values.	Memcached, Redis	Generally used as a distributed cache
Wide Column Stores (Column Family Stores)	A column-based model that stores data tables based on columns rather than rows to enable fast data access, search and aggregation.	Cassandra, HBase	Used in applications which deal with large amounts of data like recording and storing logs about consumer history in an e-commerce platform
Document Stores	A data model to store semi-structured data as documents made up of tagged elements. Semi-structured data means the data shares some standard format or encoding, such as XML, JSON, etc. The data is grouped by collections and each document in the same collection can have the same or different structure.	MongoDB, Amazon DynamoDB	Highly general purpose due to the flexibility of the data model
Graph DBMS	A network database model that uses edges and nodes to represent the data	Neo4J, AWS Neptune	Navigate social network connections

MongoDB, HBase, Memcached, and Redis are the leading technology options in the NoSQL space. We will explore when and how to use them in future projects.