Big Data & NoSQL

Theoretical Explanation

Colin Pieper
Rob Lavrijsen
Minka Firth

Fontys Hogescholen
Semester 4 – Data Processing – Module 3
Jan 2022

Contents

Big Data

When the collection of data is immense, it might be impossible to store or process it efficiently in traditional data management tools. Often these collections are growing daily as well. When talking about these massive collections, we are talking about Big Data.

Big data can be defined by a lot of characteristics, but the three originals are considered Volume, Variety and Velocity. The most straightforward one, *Volume* is the quantity of the data. It is one of the determining factors whether it can be considered big data or not. With *Variety*, we are talking about the type and nature of the data and whether it is structured or unstructured. The *Velocity* of the data is the speed the data can be generated and processed to meet the necessary standards.

Structured and Unstructured Data

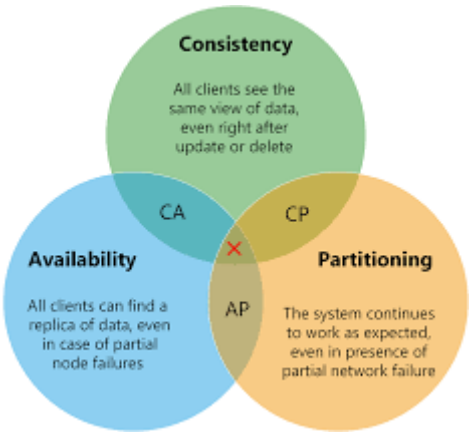
There are three types of Big Data: structured, semi-structured and unstructured data. Structured data is in a fixed format. That means the format is known beforehand. Usually these are relational databases where the data has been formatted into fields with values.

Unstructured data is not processed until it is used. It can be a collection of various file format such as chats, IoT sensor data, emails, social media commentary etc.

There are both pros and cons to both types of data. Structured data for example, can be easily used by machine learning and business users, but is severely limited by the storage options. While unstructured data can take on data stored in its native format, it is usually harder to understand for a standard business user due to this non-formatted nature.

Cap Theorem

According to computer scientist Eric Brewer only two of the three following guarantees can be provided in any distributed data store:
- **Consistency** Whichever node is connected to, all data can be seen by all clients at the same time.
- **Availability** Even if some nodes are down, a client requesting for data should get a response, without exception.
- **Partition Tolerance** Despite any number of dropped messages, the system continues to operate.



Since network failures are a threat to any distributed system, network partitioning always must be tolerated. Therefore, in case of network failures, a choice must be made between Consistency and Availability. These systems are called Available and Partition tolerant (AP) or Consistent and Partition tolerant (CP).

Simply put, in the case of network failure is it more important that the data is still available and accessible (AP) or that the data is always up to date and aligned (CP)?

Available and Partition Tolerant

In this case whenever a network failure happens, Consistency is given up for availability by simply responding to a query with the most recent data available. This might not be exactly accurate. AP databases are often used when it is more important that there is always data available instead of up to date. Often the information itself is not as important as the service.
Examples of Available and Partition Tolerant Databases:

- Cassandra
- DynamoDB
- Cosmos DB

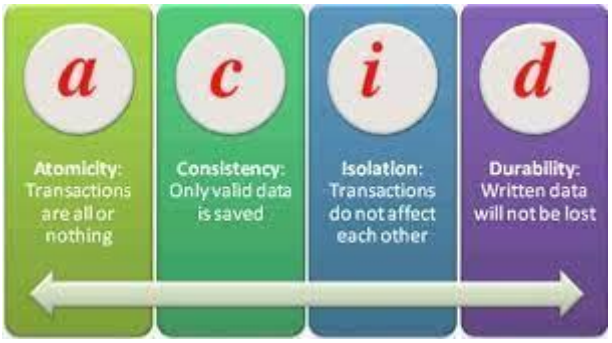
Consistency and Partition Tolerant

In this case it is particularly important to make sure the data is always available and consistent. This is especially important with financial applications, where small incorrection can easily become a colossal mistake. It is also important when multiple clients need the same information, that they all receive the same information.
Examples of Consistent and Partition Tolerant Databases:

- MongoDB
- Redis
- HBase

Sometimes this theory is misinterpreted that there should *always* be a choice between Consistency and Availability in any system, at any time, but this choice should only be made in case of a network failure. All three can be provided at other times.

ACID



ACID stands for Atomicity, Consistency, Isolation and Durability. It is the model one follows when using a CP database.

Atomicity means that either all occurs, or nothing occurs. It means that if any component of an operation fails, everything fails, and partial updates are therefore avoided. If the process is stopped, the database is returned to the state it was in before the transaction started.

Consistency is often mistaken for *correctness.* However, in this case with Consistency, Gray and Reuter meant that the data in the database should be valid to all predefined rules. Any combination of constraints, cascades and triggers must be followed, to make sure that the state of the database remains unchanged by the transaction.

There are often multiple transactions going back and forth at the same time. **Isolation** makes sure that despite interacting with different transactions while they are still in progress, the integrity of these transactions is uncompromised. It also determines the visibility of transaction integrity to other systems and users. Through moderating the access to the data, transactions seem to run sequentially instead of concurrently.

Even in the case of failures, the results of a transaction should be permanent. **Durability** ensures that transactions will remain committed even in the case of network failures, crashes, and system failures etc. This can be achieved by making sure that before the commitment, the transaction's logs are moved to a non-volatile storage.

Acid is mostly used by financial institutions because it is so important that the data is consistent. No user wants to open their banking account and see that money was taken from an account, but not actually sent to another account. Either it stays in the first account, or it moves to another one.

BASE

BASE stands, slightly less elegantly, for Basically Available, Soft state, Eventually Consistent. Instead of enforcing consistency, the data is spread and replicated across the different nodes of the database cluster, making it **Basically Available**. It appears as if the database is working most of the time.

Because there is no guaranteed consistency, it is possible for data values to change. That also means that the data might not be mutually consistent with each other. This is called **soft state.**

And lastly while it might say that the process was successfully committed, it might take a while for it to actually happen, meaning it will **Eventually be Consistent.** And in some cases, this might be good enough. Sometimes data does not have to be up to date and accurate.

Differences with Relational Databases

Differences

I think the most significant difference between a relational database, or a NoSQL database is the structure (or lack thereof). In the case of a RDBM, there are clear relationships and connections between different tables. This is the prime characteristic of these databases. It would be impossible to either make or retrieve data from these databases without them. NoSQL however there are many types of databases and are very often unstructured. There can be several types of data in the database, all thrown together.
There are many more differences though.

RDBMS	**NoSQL**
A schema or model is required before implementing the database.	No predefined schema is necessary.
Usually vertically scaled.	Usually horizontally scaled.
Higher maintenance.	Lower/cheaper Maintenance.
Mature system (with loads of tools, libraries, and tutorials)	Somewhat younger system (However, documentation and tutorials have become more common recently)
Usually better Consistency	Often poor Consistency
Works better for medium to large amounts of data.	Optimized for enormous amounts of Data

Pros and Cons NoSQL

Pros	**Cons**
Continuous availability	Not as mature as SQL
Flexible scalability	Queries are less flexible
No data models necessary and easier to modify	Often more BASE compliant, therefore atomicity and integrity are often lost.
(in general) a lot faster than SQL	
Can handle a lot of data types	

Use Cases

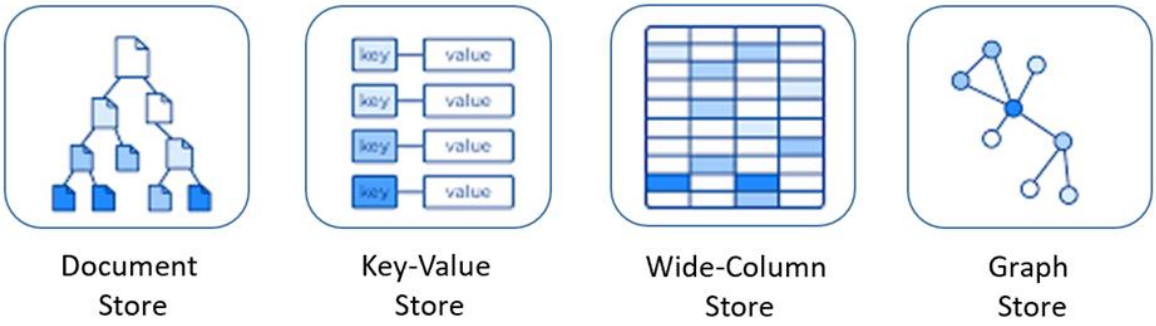
Due to high scalability, ease of access and high performance, NoSQL has gained a lot more popularity recently, which leads to people using NoSQL databases when this is not really an applicable solution. When you are dealing with transaction-oriented systems, it is important to ensure the atomicity and integrity of the data, it would therefore be more suitable to use a RDBM.

When dealing with highly structured data that does not require a lot of scaling, there are not a lot of benefits from using a NoSQL database. Especially when one is required to perform a lot of complex queries, as querying can be more flexible in a RDBM.

If you are working with a copious amount of unstructured data, it might not fit a relational data model, and a NoSQL database might therefore be more suited.
This system would also be easily horizontally scaled, which can be very costly when using a RDBM.

The most important thing to keep in mind when deciding whether to use RDBM or NoSQL is that a RDBM will always be more acid compliant, which is especially important in cases like financial systems, where the Atomicity and Integrity of the data is essential.

Different Types of NoSQL databases



Document Store | Key-Value Store | Wide-Column Store | Graph Store

Key value - every value is assigned to a specific key. Every key is unique and often called the Index key. These index keys are often Strings while the values can be JSON or XML structured, but also Strings etc. With only two columns they almost resemble a simplified relational database.

- Dynamo DB
- Redis
- BerkleyDB

Document - instead of having two columns, the values are stored in Document. These are often independent and differently structured XML or JSON files that have no relations to each other.

- MongoDB
- CouchDB
- Lotus Notes

Wide column – Usually databases store and retrieve their data by rows. In the case of column-wide, the data is stored by the columns in a table. Also, unlike relational database, the format and names of the columns can vary from row to row. It is often interpreted as a two-dimensional key-value store.

- Google’s Bigtable
- Cassandra
- HBase

Graph - in this version, there are no index lookups. Every Node is connected to an adjacent element. Graph Databases are especially made for traversing through connected data. Because each record represents a relationship with another node, the queries can be quicker than in a relational database.

- Neo4j
- JanusGraph

Optimalization of Queries

Indexing

In a default NoSQL database, documents are not sorted. When performing a query, the system must look through all the available documents to find those that match the query. To avoid looking through unnecessary documents, indexes can be implemented. Each index is given a specific Field you want to use. When performing a query, it will sort the documents by the index, avoiding scanning of unnecessary documents. If there was a big data set with a lot of different people in there and you want to know how many people were born in a certain year, it would be profitable to add an index for the field Birth Year. The indexes can be both in descending and ascending order.

There are different kind of indexes that can be implemented. There is the most straightforward one called a *Single Field Index*, in this case, as implied, you only use one field as an index. In the Case of a *Compound Index,* multiple fields are indexed within a single index.

Aggregation

Map Reducing

MapReduce is a framework that uses custom JavaScript functions. The first function *Map* reads a block of data that is processed, and key-value pairs are the output. Keys with multiple values are collected and condensed into aggregated data by the *reduce* function. This data is

Single-Purpose Aggregation

These are quite simple aggregation processes like returning unique documents or counting all the documents within a collection. This is more of an in-between option that neither gives the flexibilities or the capabilities of the other aggregation options.

Aggregation Pipeline

To break down queries into easier stages, aggregation pipelines are often implemented. The data goes through these stages in a certain flow of operations that transforms, processes and returns result. Each output of the recent stage is the input for the next stage. This way it is also possible to return totals, averages, maximum and minimum values for groups of documents. A very basic example of a pipeline looks as followed:

input -> **$match** -> **$group** -> **$sort** -> output

In the first stage the data is filtered which reduces the number of documents that are used as input for the next stage. Only documents that match the specified conditions are given to the next stage. Going into the *group* stage, the documents are input by the specified _id

expression. The output consists of documents for each unique grouping. In the last stage, everything is sorted.

Sharding

Another way of optimizing the speed of the database is through the sharding partitioning pattern. Sharding is a way of horizontal partitioning. This way, each shard is held separately on their own individual database server. Instead of increasing the performance (e.g., Upgrading the CPU, increasing RAM, etc) of the original machine (vertical scaling), the performance is improved by adding servers.

This becomes very suitable solution when a single machine can no longer handle the large workloads. If read and write operations can be contained to a single shard, the operation capacity of both these is increased, because the data is distributed across multiple shards. This also increases the availability of the database. If an entire shard is unavailable, most of the data will remain functional. It does not only spread the load of the data, but also increases the total storage capacity of the system.

Sources

Aggregation Pipeline Optimization — MongoDB Manual. (z.d.). MongoDB. Geraadpleegd op 14 januari 2022, van https://docs.mongodb.com/manual/core/aggregation-pipeline-optimization/

Aleksic, M. (2021, 12 mei). *ACID vs. BASE: Comparison of Database Transaction Models*. Knowledge Base by PhoenixNAP. Geraadpleegd op 14 januari 2022, van https://phoenixnap.com/kb/acid-vs-base

Database relationships. (z.d.). © Copyright IBM Corporation 2014. Geraadpleegd op 14 januari 2022, van https://www.ibm.com/docs/en/mam/7.6.0?topic=structure-database-relationships

Gall, R. (2019, 12 september). *The CAP Theorem in practice: The consistency vs. availability trade-off in distributed databases*. Packt Hub. Geraadpleegd op 14 januari 2022, van https://hub.packtpub.com/the-cap-theorem-in-practice-the-consistency-vs-availability-trade-off-in-distributed-databases/

Indexes — MongoDB Manual. (z.d.). MongoDB. Geraadpleegd op 14 januari 2022, van https://docs.mongodb.com/manual/indexes/

Johnson, J. (z.d.). *CAP Theorem for Databases: Consistency, Availability & Partition Tolerance*. BMC Blogs. Geraadpleegd op 14 januari 2022, van https://www.bmc.com/blogs/cap-theorem/

MongoDB. (z.d.-a). *MongoDB Aggregation Pipeline*. Geraadpleegd op 14 januari 2022, van https://www.mongodb.com/basics/aggregation-pipeline

MongoDB. (z.d.-b). *NoSQL vs Relational Databases*. Geraadpleegd op 14 januari 2022, van https://www.mongodb.com/scale/nosql-vs-relational-databases

Pedamkar, P. (2021, 2 maart). *RDBMS vs NoSQL*. EDUCBA. Geraadpleegd op 14 januari 2022, van https://www.educba.com/rdbms-vs-nosql/

Rajani, V. (2021, 27 oktober). *A Dive Deep into Types of NoSQL Databases*. Blazeclan. Geraadpleegd op 14 januari 2022, van https://www.blazeclan.com/blog/dive-deep-types-nosql-databases/

RDBMS vs NoSQL. (z.d.). LoginRadius. Geraadpleegd op 14 januari 2022, van https://www.loginradius.com/blog/async/relational-database-management-system-rdbms-vs-nosql/

Resource Center: Talend Guides and Tutorials. (2021, 16 december). Talend - A Leader in Data Integration & Data Integrity. Geraadpleegd op 14 januari 2022, van https://www.talend.com/resources/

robvet. (2021, 30 november). *Relational vs. NoSQL data*. Microsoft Docs. Geraadpleegd op 14 januari 2022, van https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/relational-vs-nosql-data

Sasaki, B. M. (2018, 14 november). *Graph Databases for Beginners: ACID vs. BASE Explained*. Neo4j Graph Database Platform. Geraadpleegd op 14 januari 2022, van https://neo4j.com/blog/acid-vs-base-consistency-models-explained/

Sheldon, R. (2021, 3 mei). *How to choose between SQL and NoSQL databases*. Simple Talk. Geraadpleegd op 14 januari 2022, van https://www.red-gate.com/simple-talk/databases/nosql/how-to-choose-between-sql-and-nosql-databases/

Team, P. F. (2019, 25 juni). *NoSQL Databases: The Definitive Guide*. Pandora FMS Monitoring Blog. Geraadpleegd op 14 januari 2022, van https://pandorafms.com/blog/nosql-databases-the-definitive-guide/

Vargas, K. (2021, 4 augustus). *The Main NoSQL Database Types*. Studio 3T. Geraadpleegd op 14 januari 2022, van https://studio3t.com/knowledge-base/articles/nosql-database-types/

What is NoSQL? Non-Relational Databases Explained. (z.d.). DataStax. Geraadpleegd op 14 januari 2022, van https://www.datastax.com/nosql