NoSQL & REDIS

NoSQL Content:

Characteristics of NoSQL Databases

NoSQL – The Relational Database

ACID Issues

Normalization & Denormalization

CAP Theorem and Distributed Database Management Systems

SQL&NoSQL Database Types

NoSQL Cluster TYPES

Redis Content:

Redis installation and configuration

Basic Redis commands

Redis Sorted Sets, Hyperlog, Streaming, Transactions

Redis Sentinel

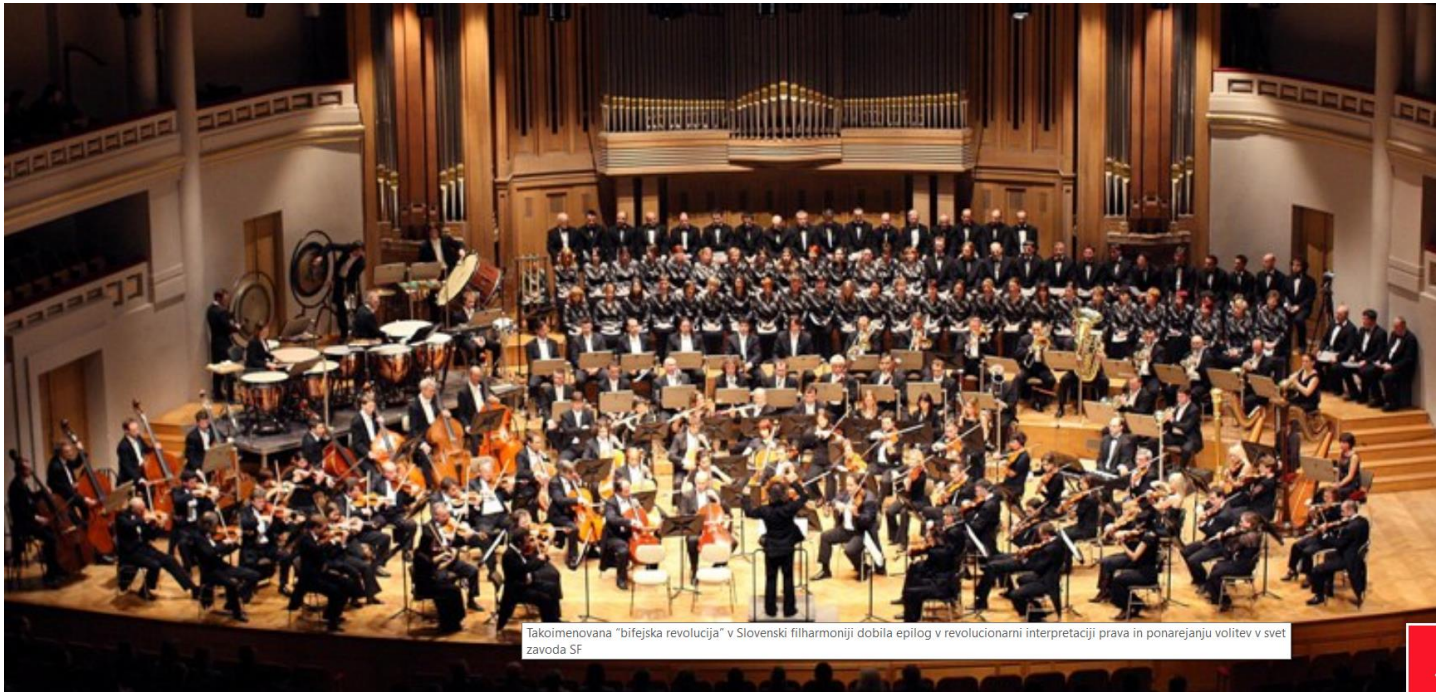
Redis Cluster

Redis Monitoring

Paredicma Redis Cluster

Petronas Towers = ?

Orange Town = ?



Takoimenovana "bifejska revolucija" v Slovenski filharmoniji dobila epilog v revolucionarni interpretaciji prava in ponarejanju volitev v svet zavoda SF

Characteristics of NoSQL Databases

The term NoSQL now generally refers to a particular group of DBMSs that share certain characteristics, such as the following.

- Non-Relational
- Open Source
- Schema-Less
- Horizontally Scalable
- Lack of Adherence to ACID Principles
- No Standard Query Language
- Denormalized Data

Not all NoSQL databases will possess all of these characteristics. Many NoSQL systems will possess perhaps several of the above characteristics. However, most of these characteristics are inherently lacking in relational databases.

The First NoSQL Databases

There have been many "non SQL" databases throughout the years. Even before SQL was first proposed in 1974, and the first RDBMS was commercially released in 1979, there were still databases.

Here are three of the first (NoSQL) database management systems ever built:

- MultiValue is a NoSQL database that was first developed in 1965 as part of the Pick operating system..
- MUMPS (Massachusetts General Hospital Utility Multi-Programming System) has been around since 1966. MUMPS which is schema-less, and uses a key-value database engine, is a classic example of a NoSQL database.
- IBM IMS is a joint hierarchical database and information management system that was developed in 1996.

Of course, no one called these "NoSQL" databases – SQL hadn't even been invented yet. It wasn't until after 2009 that these older DBMSs were associated with NoSQL.

NoSQL – The Relational Database

NoSQL is also the name of a relational database management system (RDBMS) originally developed by Carlo Strozzi in 1998.

NoSQL – the RDBMS – is named as such because it doesn't use SQL as its query language. NoSQL – the RDBMS – has no connection to the NoSQL movement that began in 2009.

ACID Properties in DBMS

A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.
In order to maintain consistency in a database, before and after transaction, certain properties are followed. These are called **ACID**properties.

Atomicity
By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves following two operations.
—**Abort**: If a transaction aborts, changes made to database are not visible.
—**Commit**: If a transaction commits, changes made are visible.
Atomicity is also known as the 'All or nothing rule'.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

Consistency
This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to correctness of a database. Referring to the example above,
The total amount before and after the transaction must be maintained.
Total **before T** occurs = **500 + 200 = 700**.
Total **after T occurs** = **400 + 300 = 700**.
Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result T is incomplete.

Isolation
This property ensures that multiple transactions can occur concurrently without leading to inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.
Let **X**= 500, **Y** = 500.
Consider two transactions **T** and **T".**

T	T''
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write	Read (X) Read (Y) Z: = X + Y Write (Z)

Suppose **T** has been executed till **Read (Y)** and then **T''** starts. As a result , interleaving of operations takes place due to which **T''**reads correct value of **X** but incorrect value of **Y** and sum computed by
T'': (X+Y = 50, 000+500=50, 500)
is thus not consistent with the sum at end of transaction:
T: (X+Y = 50, 000 + 450 = 50, 450).
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after a they have been made to the main memory.

Durability:
This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if system failure occurs. These updates now become permanent and are stored in a non-volatile memory. The effects of the transaction, thus, are never lost.
The **ACID** properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.

Isolation Levels

There are four levels of isolation. Higher isolation limits the ability of users to concurrently access the same data. The higher the isolation level, the greater system resources are required and the more likely database transactions will block one another.

As the isolation level is lowered, the more there is a chance that users will encounter read phenomena such as uncommitted dependencies, also known as dirty reads, which result in data being read from a row that has been modified by another user but not yet committed to the database.

1. **Serializable** is the highest level, which means that the transactions will be completed before another transaction is able to start.
2. **Repeatable** reads allow transactions to be accessed once the transaction has started, even though it hasn’t been finished.
3. **Read committed** allows the data to be accessed after the data has been committed to the database, but not before then.
4. **Read uncommitted** is the lowest level of isolation and allows data to be accessed before the changes have been made.

Normalization & Denormalization

Insert Anomaly

EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		
???	???	Atlanta	312-555-1212			

There are facts we cannot record until we know information for the entire row. In our example we cannot record a new sales office until we also know the sales person. Why? Because in order to create the record, we need provide a primary key. In our case this is the EmployeeID.

Update Anomaly

EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		

In this case we have the same information in several rows. For instance if the office number changes, then there are multiple updates that need to be made. If we don't update all rows, then inconsistencies appear.

Deletion Anomaly

EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
~~1004~~	~~John Hunt~~	~~New York~~	~~212-555-1212~~	~~Dell~~	~~HP~~	~~Apple~~
1005	Martin Hap	Chicago	312-555-1212	Boeing		

Deletion of a row causes removal of more than one set of facts. For instance, if John Hunt retires, then deleting that row cause us to lose information about the New York office.

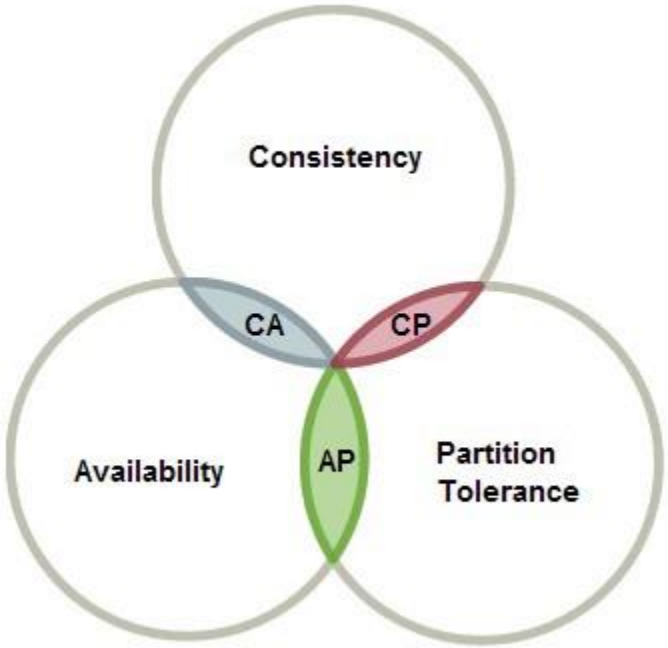
Key Differences Between Normalization and Denormalization

1. Normalization is the technique of dividing the data into multiple tables to reduce data redundancy and inconsistency and to achieve data integrity. On the other hand, Denormalization is the technique of combining the data into a single table to make data retrieval faster.
2. Normalization is used in **OLTP** system, which emphasizes on making the insert, delete and update anomalies faster. As against, Denormalization is used in **OLAP** system, which emphasizes on making the search and analysis faster.
3. Data integrity is maintained in normalization process while in denormalization data integrity harder to retain.
4. Redundant data is eliminated when normalization is performed whereas denormalization increases the redundant data.
5. Normalization increases the number of tables and joins. In contrast, denormalization reduces the number of tables and join.
6. Disk space is wasted in denormalization because same data is stored in different places. On the contrary, disk space is optimized in a normalized table.

BASIS FOR COMPARISON	NORMALIZATION	DENORMALIZATION
Basic	Normalization is the process of creating a set schema to store non-redundant and consistent data.	Denormalization is the process of combining the data so that it can be queried speedily.
Purpose	To reduce the data redundancy and inconsistency.	To achieve the faster execution of the queries through introducing redundancy.
Used in	OLTP system, where the emphasize is on making the insert, delete and update anomalies faster and storing the quality data.	OLAP system, where the emphasis is on making the search and analysis faster.
Data integrity	Maintained	May not retain
Redundancy	Eliminated	Added
Number of tables	Increases	Decreases
Disk space	Optimized usage	Wastage

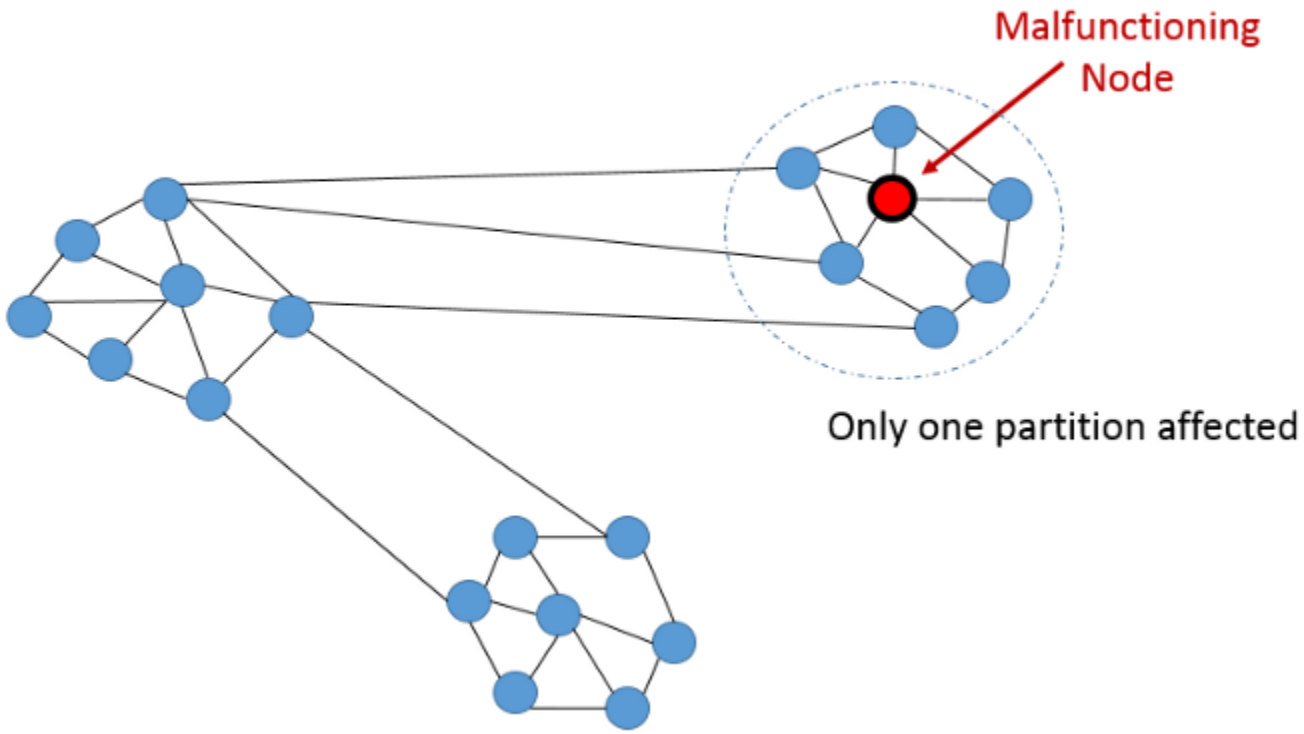
CAP Theorem and Distributed Database Management Systems

CAP Theorem is a concept that a distributed database system can only have 2 of the 3: Consistency, Availability and Partition Tolerance.



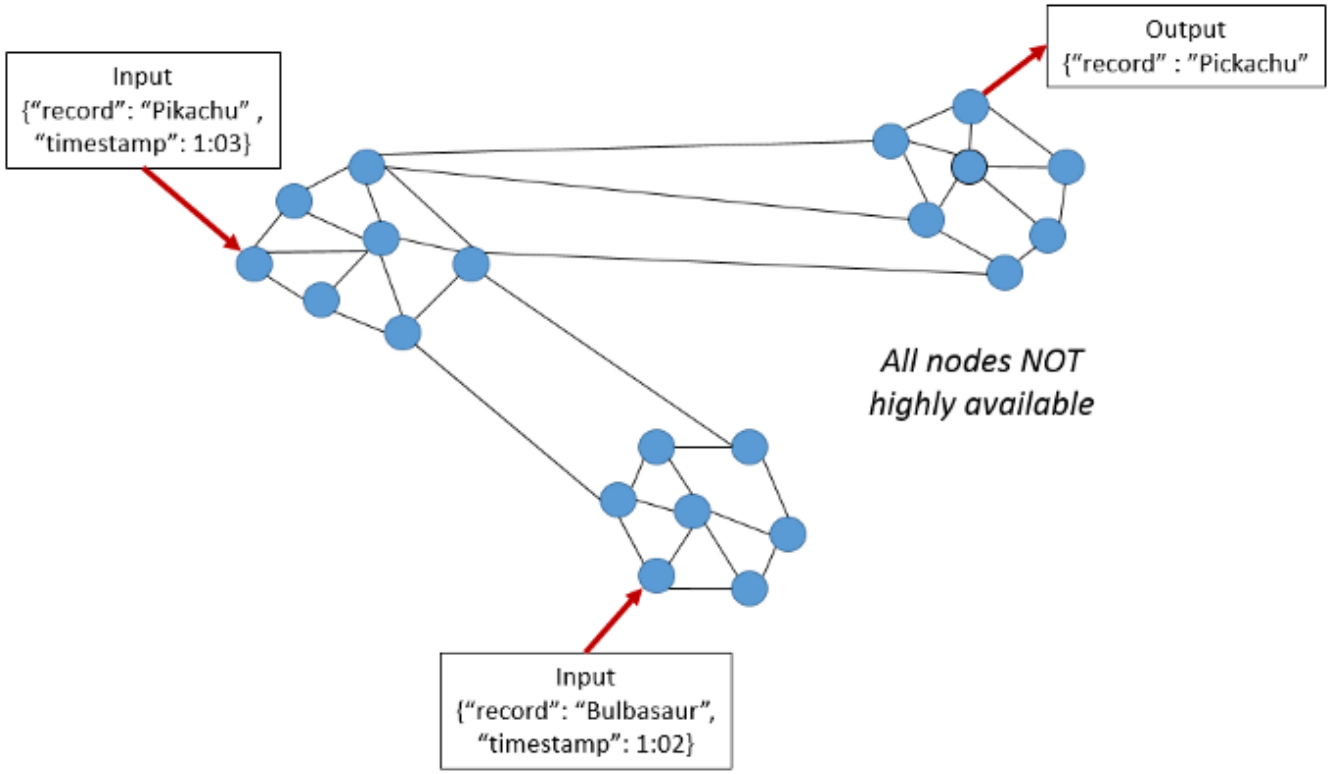
CAP Theorem is very important in the Big Data world, especially when we need to make trade off's between the three, based on our unique use case. On this blog, I will try to explain each of these concepts and the reasons for the trade off. I will avoid using specific examples as DBMS are rapidly evolving.

Partition Tolerance



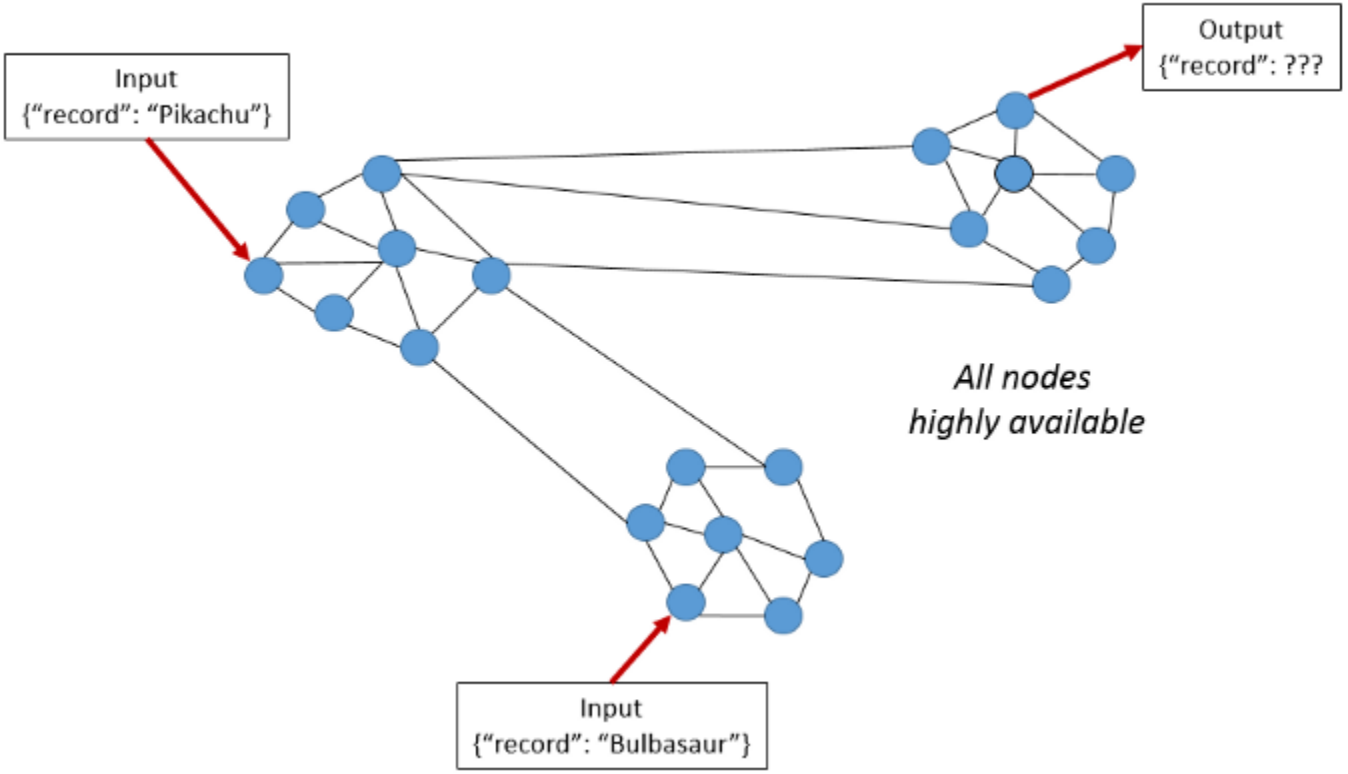
This condition states that the system continues to run, despite the number of messages being delayed by the network between nodes. A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network. Data records are sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages. When dealing with modern distributed systems, **Partition Tolerance is not an option. It's a necessity**. Hence, we have to trade between Consistency and Availability.

High Consistency



This condition states that all nodes see the same data at the same time. Simply put, performing a *read* operation will return the value of the most recent *write* operation causing all nodes to return the same data. A system has consistency if a transaction starts with the system in a consistent state, and ends with the system in a consistent state. In this model, a system can (and does) shift into an inconsistent state during a transaction, but the entire transaction gets rolled back if there is an error during any stage in the process. In the image, we have 2 different records ("Bulbasaur" and "Pikachu") at different timestamps. The output on the third partition is "Pikachu", the latest input. However, the nodes will need time to update and will not be Available on the network as often.

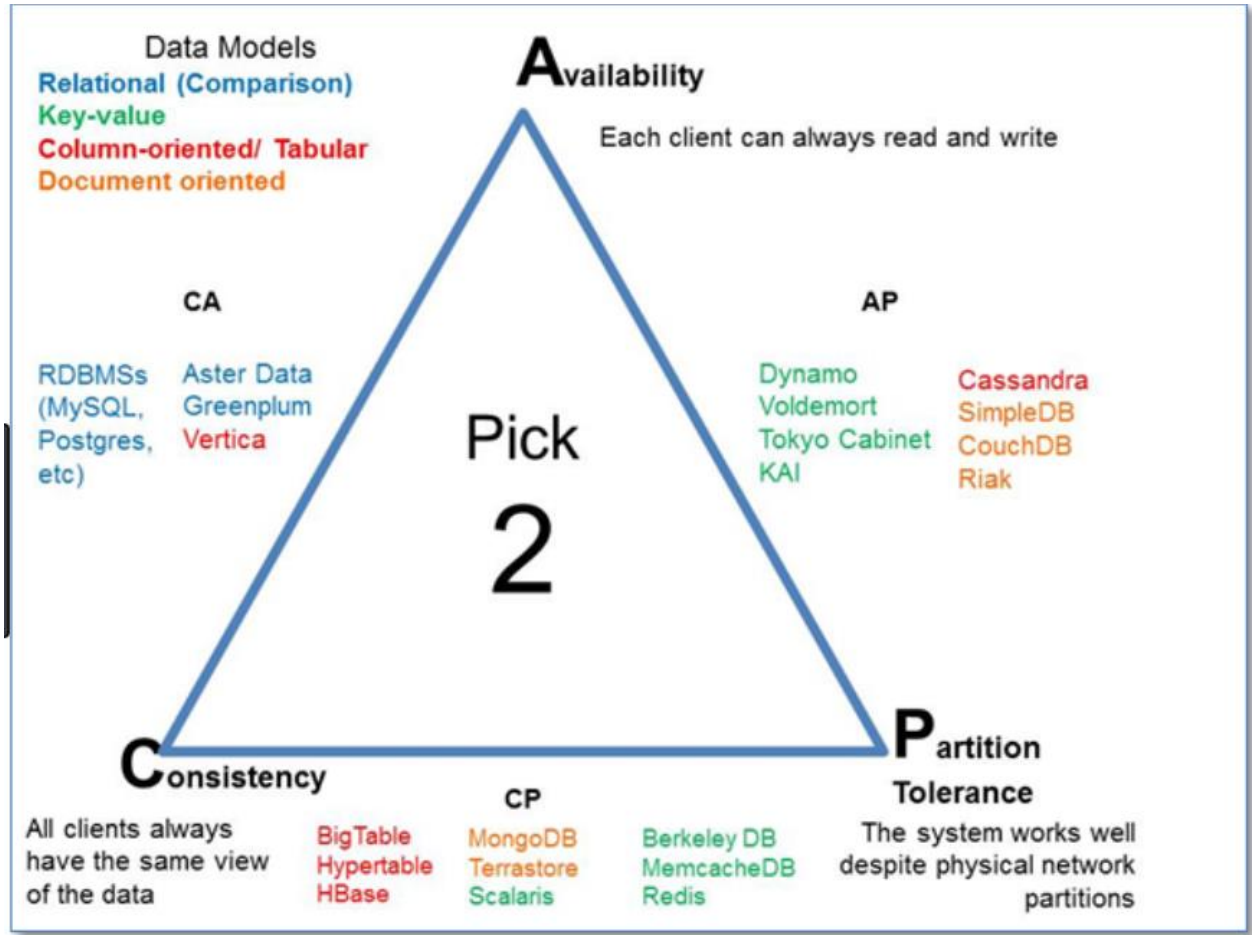
High Availability

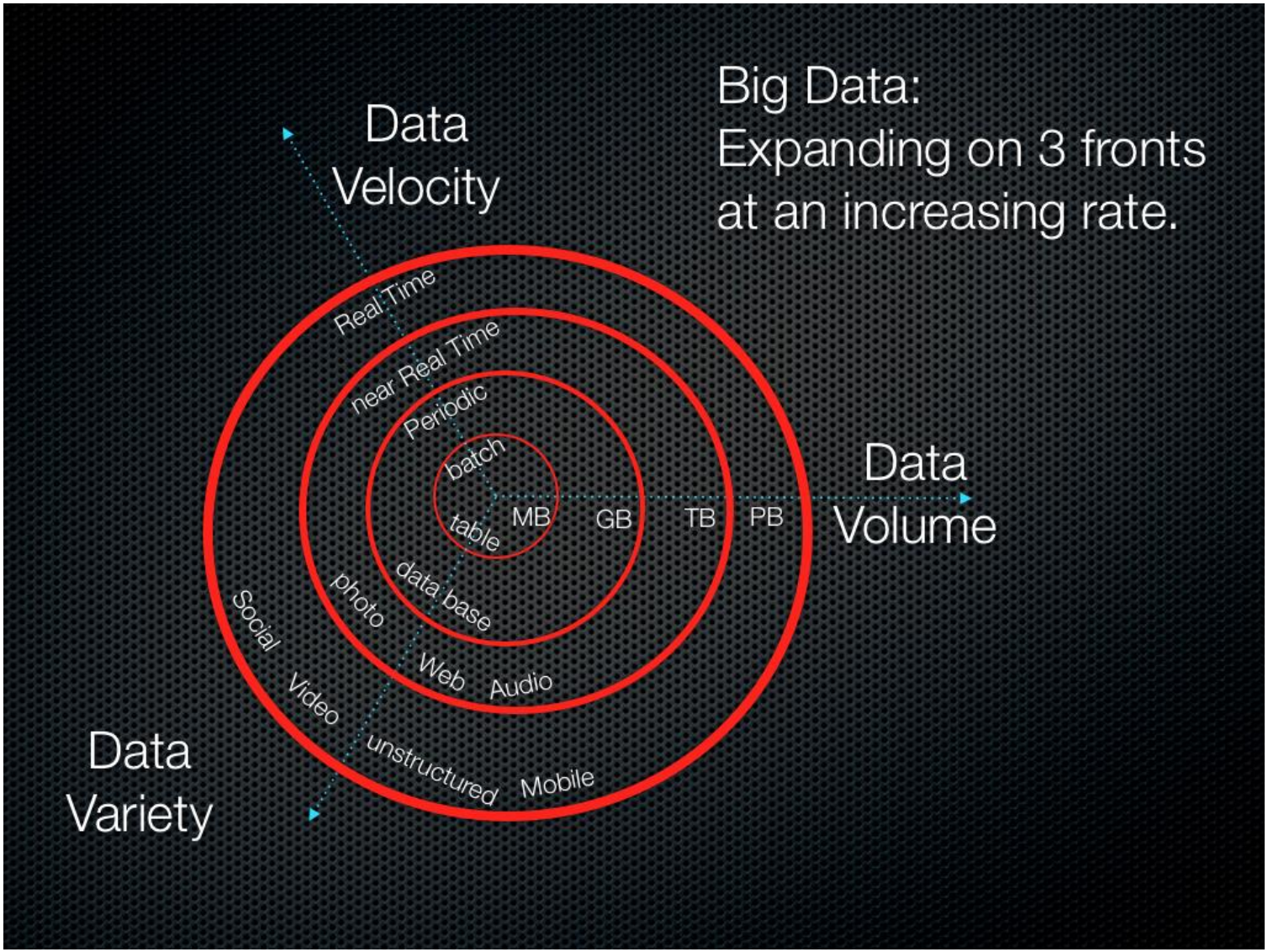


This condition states that every request gets a response on success/failure. Achieving availability in a distributed system requires that the system remains operational 100% of the time. Every client gets a response, regardless of the state of any individual node in the system. This metric is trivial to measure: either you can submit read/write commands, or you cannot. Hence, the databases are time independent as the nodes need to be available online at all times. This means that, unlike the previous example, we do not know if “Pikachu” or “Bulbasaur” was added first. The output could be either one. Hence why, high availability isn’t feasible when analyzing streaming data at high frequency.

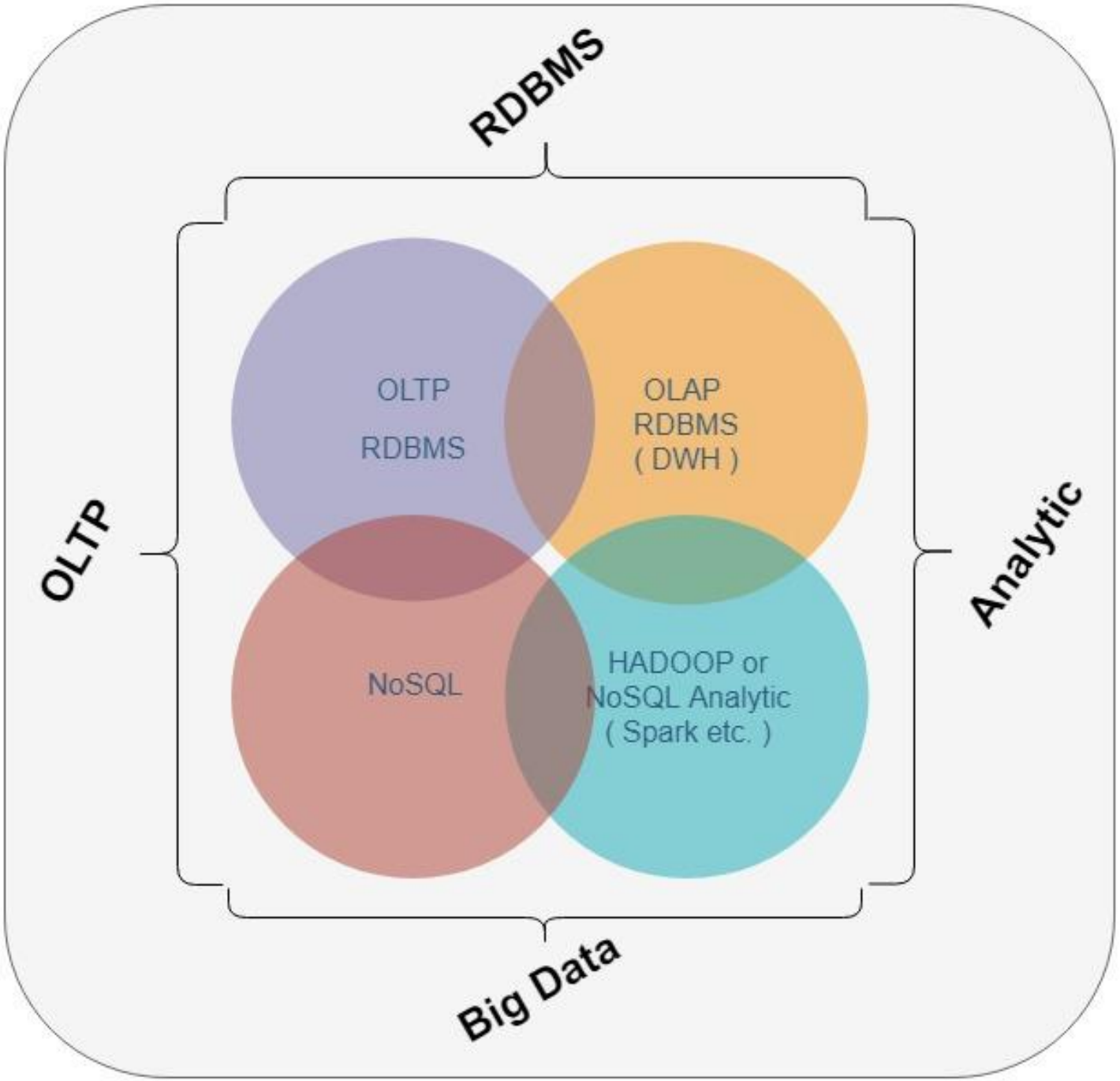
Conclusion

Distributed systems allow us to achieve a level of computing power and availability that were simply not available in the past. Our systems have higher performance, lower latency, and near 100% up-time in data centers that span the entire globe. Best of all, the systems of today are run on commodity hardware that is easily obtainable and configurable at affordable costs. However, there is a price. Distributed systems are more complex than their single-network counterparts. Understanding the complexity incurred in distributed systems, making the appropriate trade-offs for the task at hand (CAP), and selecting the right tool for the job is necessary with horizontal scaling.

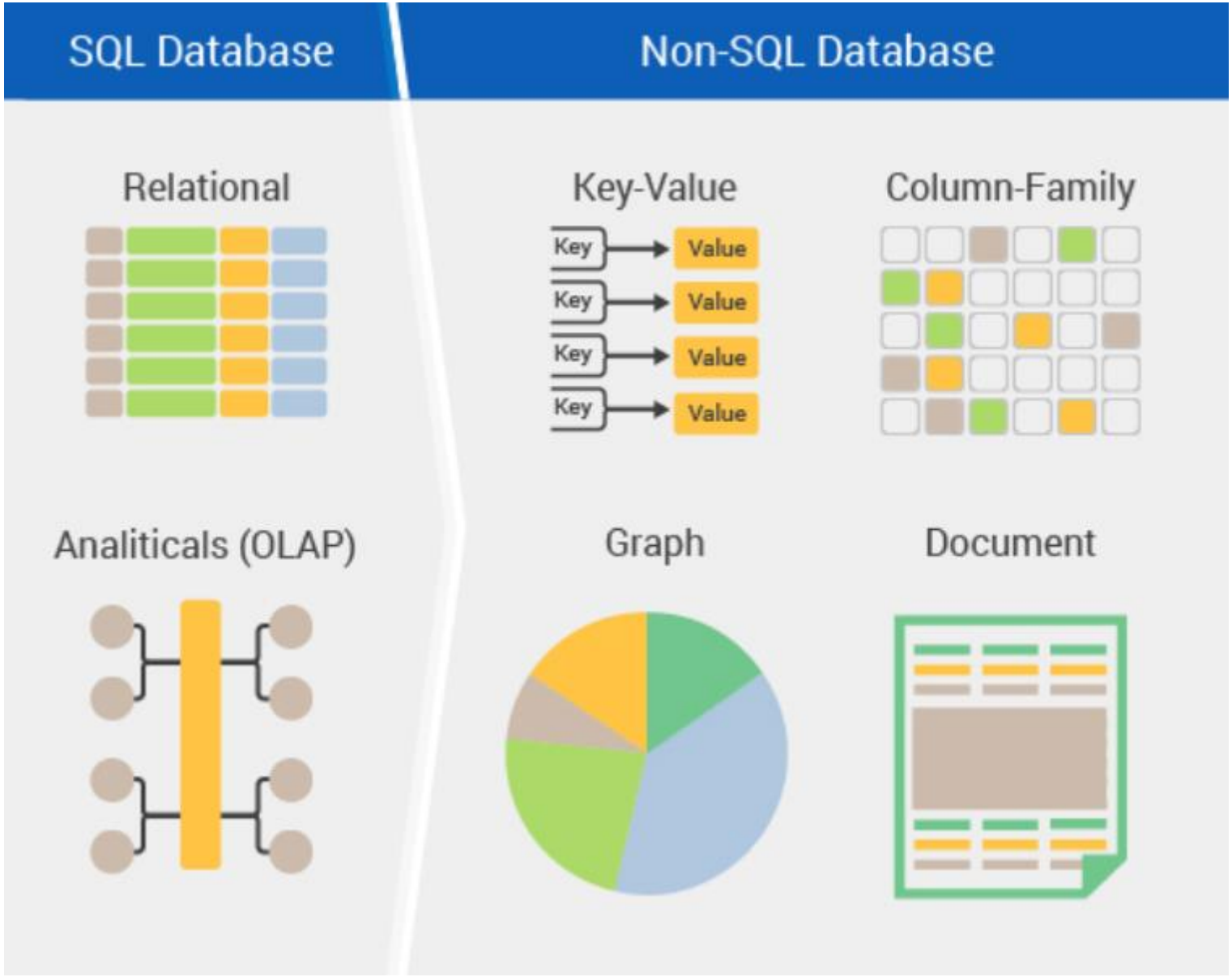




https://www.datasciencecentral.com/forum/topics/the-3vs-that-define-big-data


RDBMS
OLTP
Analytic
Big Data
OLTP
RDBMS
OLAP
RDBMS
(DWH)
NoSQL
HADOOP or
NoSQL Analytic
(Spark etc.)

SQL&NoSQL Database Types



NoSQL databases are often categorised under four main types. Some databases are a mix between different types, but in general, they fit under the following main categories.

Key-Value

A key-value database, is a database that uses a simple key/value method to store data.

The key-value part refers to the fact that the database stores data as a collection of key/value pairs. This is a simple method of storing data, and it is known to scale well.

Here's an example of a key-value store:

Key	Value
Bob	(123) 456-7890
Jane	(234) 567-8901
Tara	(345) 678-9012
Tiara	(456) 789-0123

This is a simple phone directory. The person's name is the key, and the phone number is the value.

This type of database is very quick to query, due to its simplicity.

The key-value model is well suited to storing things like user profiles and session info on a website, blog comments, telecom directories, IP forwarding tables, shopping cart contents on e-commerce sites, and more.

Examples of key-store database management systems include:

- Redis, Oracle NoSQL Database, Voldemorte, Aerospike, Oracle Berkeley DB

Document Store

A document store database uses a document-oriented model to store data. It is similar to a key-value database in that it uses a key-value approach. The difference is that, the value in a document store database consists of semi-structured data.

Also known as a document oriented or aggregate database, a document store database stores each record and its associated data within a single *document*. Each document contains semi-structured data that can be queried against using various query and analytics tools of the DBMS.

The documents in document stores are usually XML or JSON, but some DBMSs use other languages, such as BSON, YAML, etc.

Here's an example of a document written in JSON:

```
{
  '_id' : 1,
  'artistName' : { 'Iron Maiden' },
  'albums' : [
```

```
    {
        'albumname' : 'The Book of Souls',
        'datereleased' : 2015,
        'genre' : 'Hard Rock'
    }, {
        'albumname' : 'Killers',
        'datereleased' : 1981,
        'genre' : 'Hard Rock'
    }, {
        'albumname' : 'Powerslave',
        'datereleased' : 1984,
        'genre' : 'Hard Rock'
    }, {
        'albumname' : 'Somewhere in Time',
        'datereleased' : 1986,
        'genre' : 'Hard Rock'
    }
]
```

```
}
```

In a document store, this document can be retrieved by referring to the `_id`.

Document store databases can be used for a wide variety of use cases. They are ideal for content management systems, blogging platforms, and other web applications.

They are also well suited for user generated content such as blog comments, chat sessions, tweets, ratings, etc.

Document stores are also great for providing real time analytics and other reporting features.

Here are examples of document store DBMSs.

- MongoDB, DocumentDB, CouchDB, MarkLogic, OrientDB

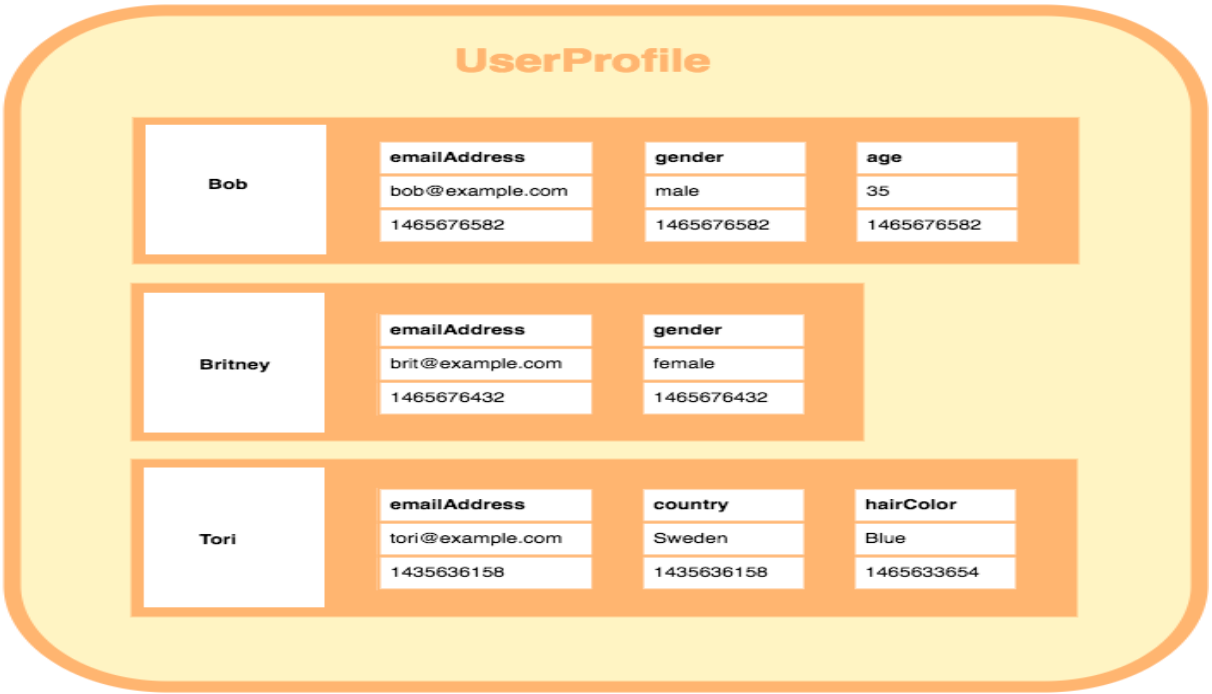
Column Store

A column store database is a type of database that stores data using a column oriented model.

Column stores work in a similar fashion to relational databases in that they have rows, columns, and tables (also known as *column families*). However, these work differently in column store databases.

In a column store database, the columns in each row are contained within that row. Each row can have different columns to the other rows. They can be in a different order, then can even have different data types, etc.

Here's a diagram to demonstrate this:



A column family containing 3 rows. Each row contains its own set of columns.

This method of storing data can be extremely quick to load and query.

Columnar databases can also store massive amounts of data, and they can be scaled easily using massively parallel processing (MPP), which involves having data spread across a large cluster of machines.

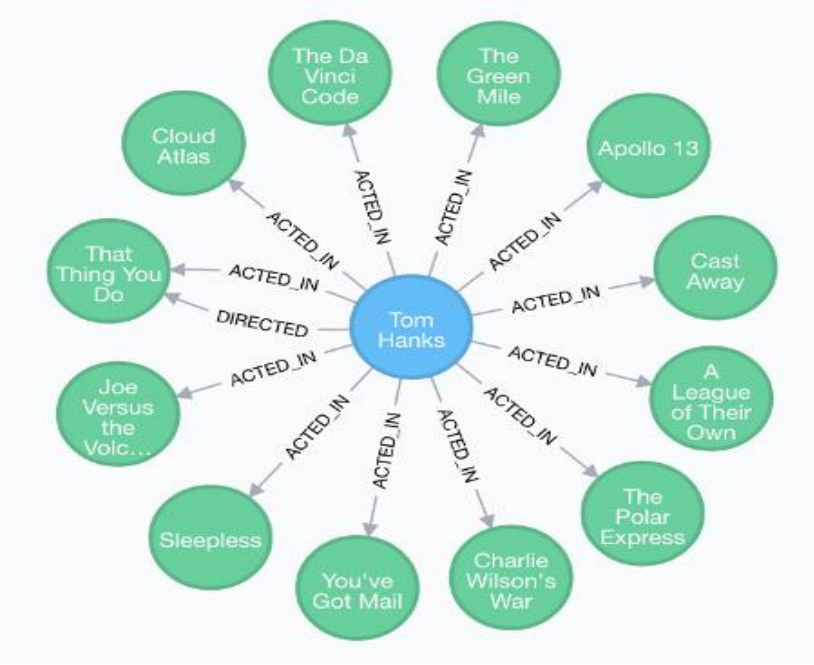
Examples of column store databases include:

- Bigtable, Cassandra, HBase, Vertica, Druid, Accumulo, Hypertable

Graph

Graph databases use a graphical model to represent and store the data.

Here's a basic example of how graph databases store and present data:



Example of a simple graph database.

The circles are *nodes* – they contain the data. The arrows represent the relationships that each node has with other nodes.

Graph databases are an excellent choice for working with connected data – data that contains lots of interconnected relationships. In fact, graph databases are much more suited to displaying relational data than relational databases.

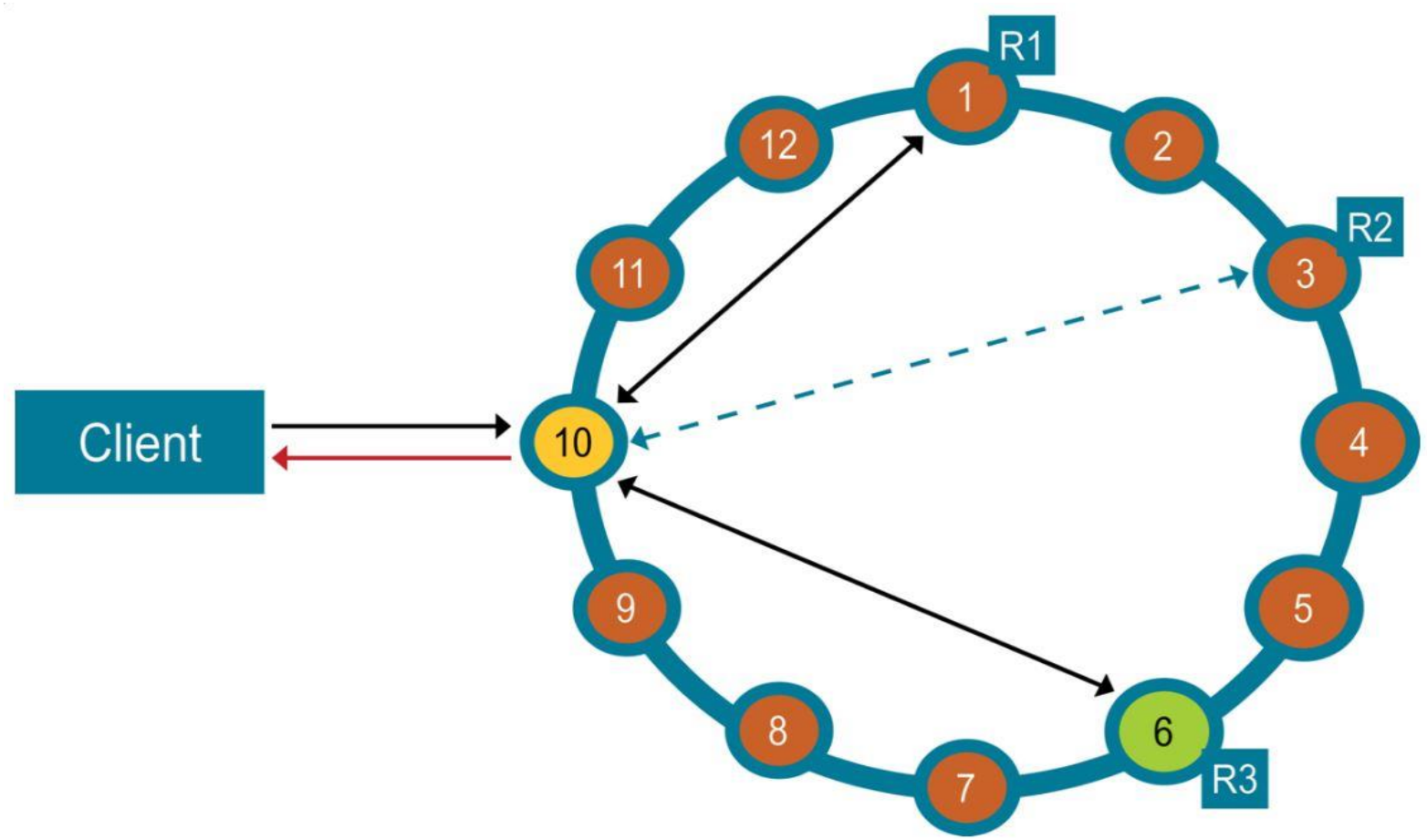
Graph databases are very well suited to applications like social networks, realtime product recommendations, network diagrams, fraud detection, access management, and more.

As with most NoSQL databases, there's no fixed schema as such. Any "schema" is simply a reflection of the data that has been entered. As more varied data is entered, the schema grows accordingly.

Examples of graph databases include Neo4j, Blazegraph, and OrientDB.

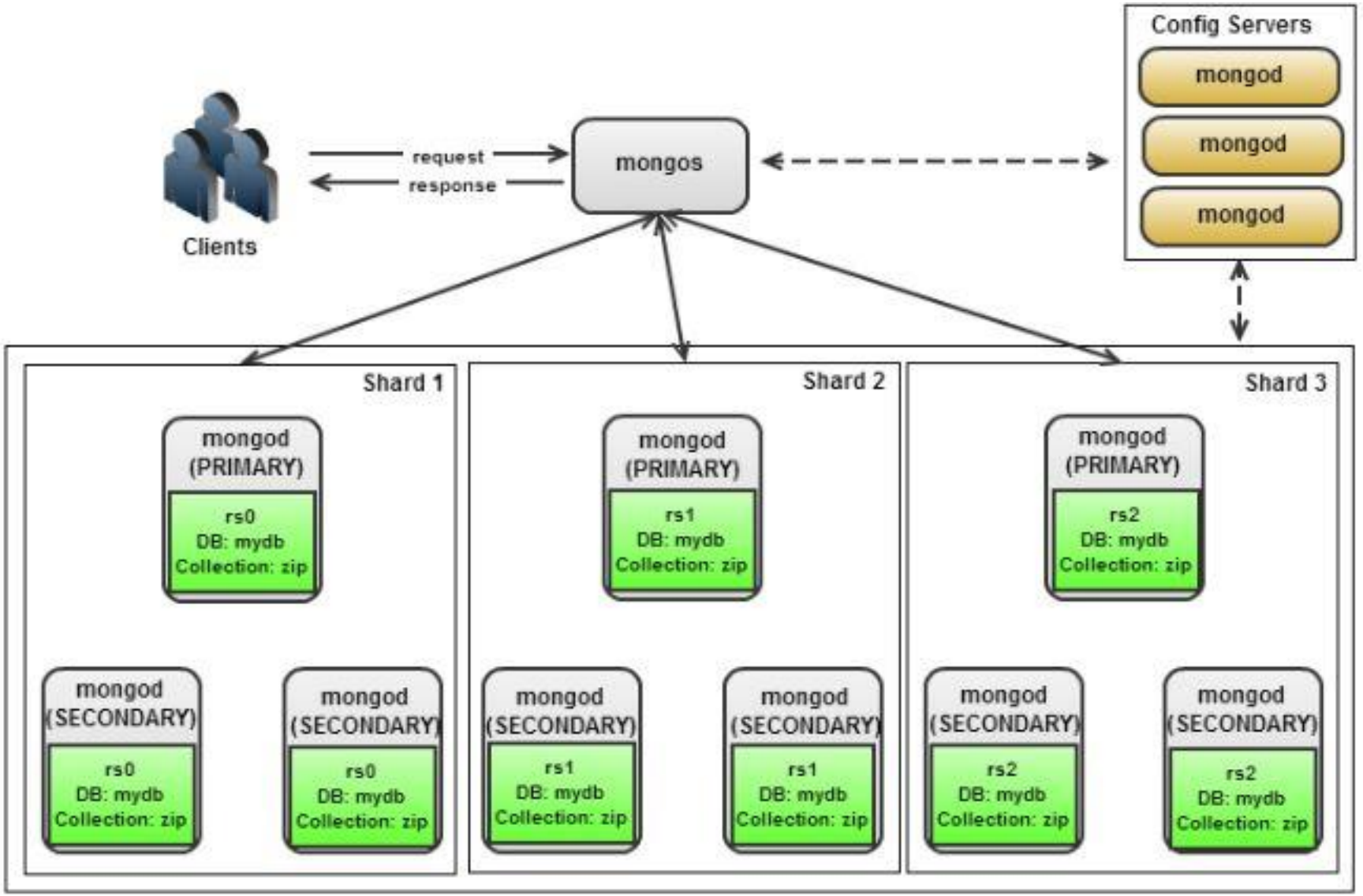
NoSQL Cluster TYPES

1. Token Range Partition Base (Dynamo) Cluster

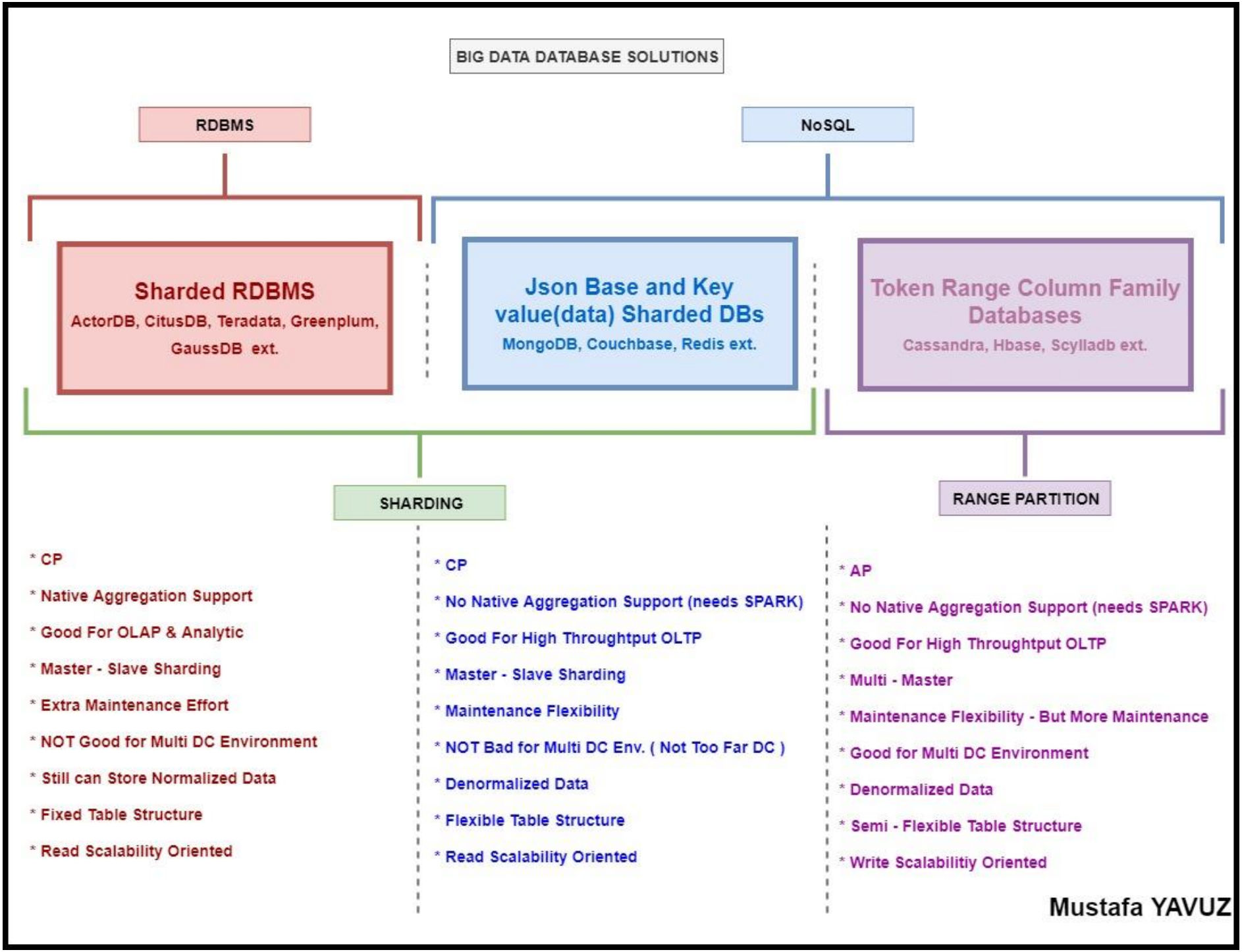


https://intellipaat.com/blog/cass-1/

2. Sharding (Master - Slave) Cluster



https://dba.stackexchange.com/questions/82551/data-distribution-in-mongos-with-shards-or-replica-sets


BIG DATA DATABASE SOLUTIONS
RDBMS
NoSQL
Sharded RDBMS
ActorDB, CitusDB, Teradata, Greenplum, GaussDB ext.
Json Base and Key value(data) Sharded DBs
MongoDB, Couchbase, Redis ext.
Token Range Column Family Databases
Cassandra, Hbase, Scylladb ext.
SHARDING
RANGE PARTITION
* CP
* Native Aggregation Support
* Good For OLAP & Analytic
* Master - Slave Sharding
* Extra Maintenance Effort
* NOT Good for Multi DC Environment
* Still can Store Normalized Data
* Fixed Table Structure
* Read Scalability Oriented
* CP
* No Native Aggregation Support (needs SPARK)
* Good For High Throughtput OLTP
* Master - Slave Sharding
* Maintenance Flexibility
* NOT Bad for Multi DC Env. (Not Too Far DC)
* Denormalized Data
* Flexible Table Structure
* Read Scalability Oriented
* AP
* No Native Aggregation Support (needs SPARK)
* Good For High Throughtput OLTP
* Multi - Master
* Maintenance Flexibility - But More Maintenance
* Good for Multi DC Environment
* Denormalized Data
* Semi - Flexible Table Structure
* Write Scalabilitiy Oriented
Mustafa YAVUZ

Golden Rules

- Do NOT make pieces bigger.
- Do NOT make pieces smaller than it needs.
- Think about management, monitoring, maintenance
- Use Physical Servers
- Do NOT use Shared Storages (SAN or NAS) – worse choise
- Do NOT use Virtual Servers
- Might use Docker Architecture
- Needs High Throughput Network Infrastructure
- Use better IO latency Disks
- Client Driver is an Important Issue
-

Ref 1: https://techdifferences.com/difference-between-normalization-and-denormalization.html

Ref 2: https://www.essentialsql.com/get-ready-to-learn-sql-database-normalization-explained-in-simple-english/

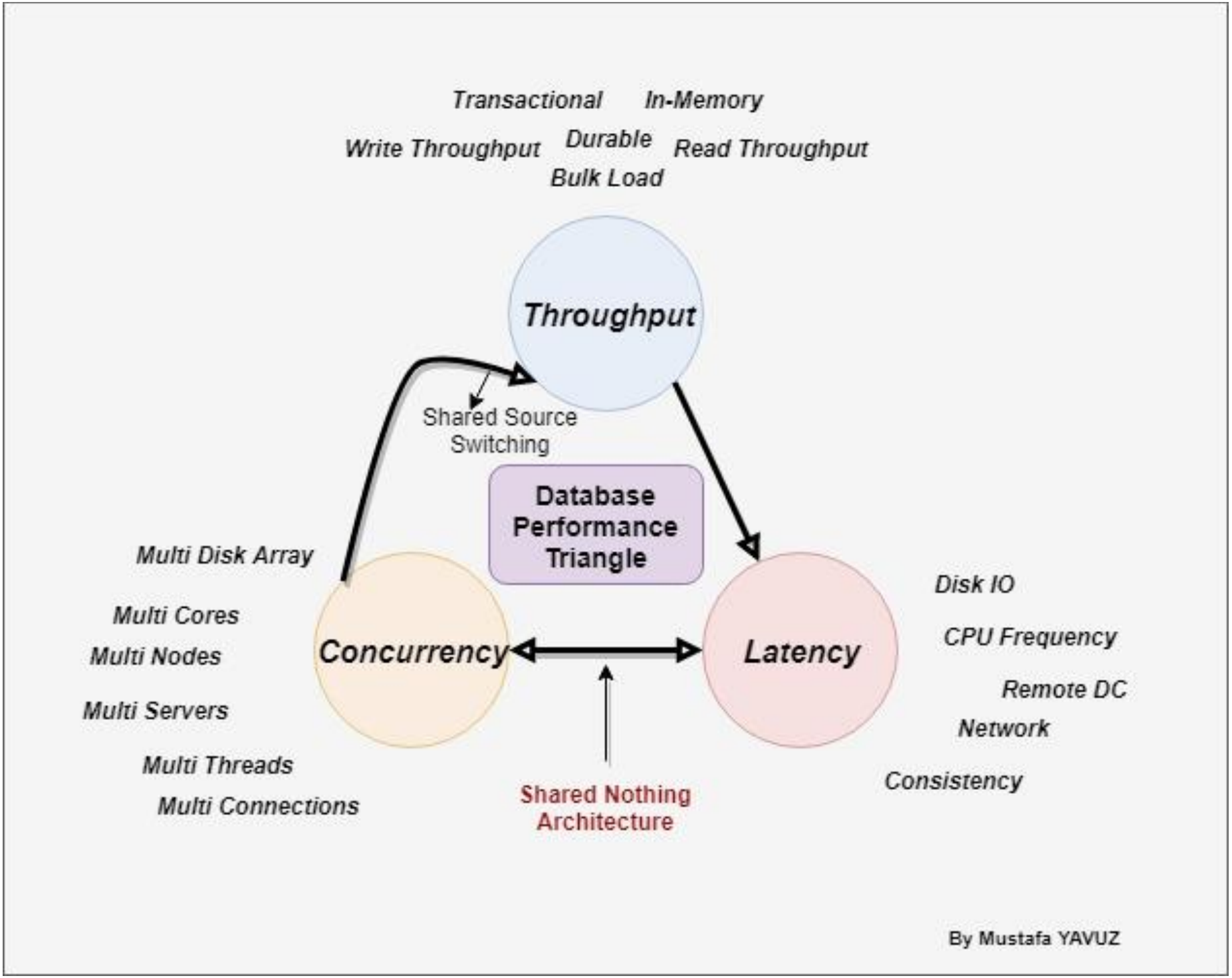
Ref 3: https://database.guide/what-is-nosql/

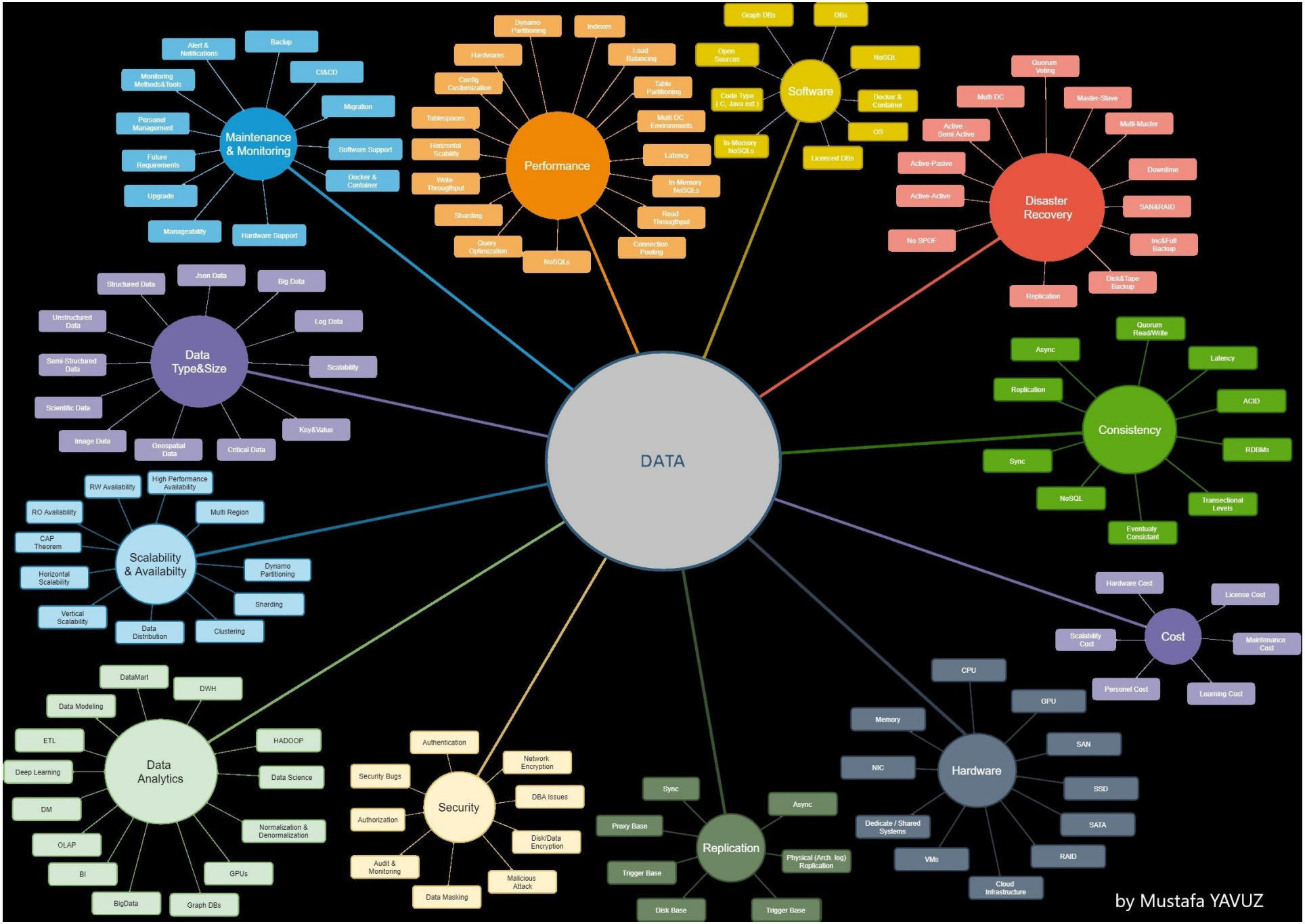
Ref 4: https://database.guide/nosql-database-types/

Ref 5: https://www.geeksforgeeks.org/acid-properties-in-dbms/

Ref 6: https://www.lifewire.com/isolation-definition-1019173

Ref 7: https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e


Transactional
In-Memory
Write Throughput
Durable
Read Throughput
Bulk Load
Throughput
Shared Source
Switching
Database
Performance
Triangle
Multi Disk Array
Multi Cores
Multi Nodes
Multi Servers
Multi Threads
Multi Connections
Concurrency
Latency
Disk IO
CPU Frequency
Remote DC
Network
Consistency
Shared Nothing
Architecture
By Mustafa YAVUZ


DATA
Maintenance & Monitoring
Alert & Notifications
Backup
CI&CD
Migration
Software Support
Docker & Container
Hardware Support
Manageability
Upgrade
Future Requirements
Personel Management
Monitoring Methods&Tools
Performance
Dynamo Partitioning
Indexes
Load Balancing
Table Partitioning
Multi DC Environments
Latency
In-Memory NoSQLs
Read Throughput
Connection Pooling
NoSQLs
Query Optimization
Sharding
Write Throughput
Horizontal Scability
Tablespaces
Config Customization
Hardwares
Software
Graph DBs
DBs
NoSQL
Docker & Container
OS
Licensed DBs
In-Memory NoSQLs
Code Type (C, Java ext.)
Open Sources
Disaster Recovery
Quorum Voting
Multi DC
Master-Slave
Multi-Master
Downtime
SAN&RAID
Inc&Full Backup
Disk&Tape Backup
Replication
No SPOF
Active-Active
Active-Pasive
Active-Semi Active
Data Type&Size
Structured Data
Json Data
Big Data
Log Data
Scalability
Key&Value
Critical Data
Geospatial Data
Image Data
Scientific Data
Semi-Structured Data
Unstructured Data
Consistency
Quorum Read/Write
Latency
ACID
RDBMs
Transectional Levels
Eventualy Consistant
NoSQL
Sync
Replication
Async
Scalability & Availabilty
RW Availability
High Performance Availability
Multi Region
Dynamo Partitioning
Sharding
Clustering
Data Distribution
Vertical Scalability
Horizontal Scalability
CAP Theorem
RO Availability
Cost
Hardware Cost
License Cost
Maintenance Cost
Learning Cost
Personel Cost
Scalability Cost
Data Analytics
DataMart
DWH
HADOOP
Data Science
Normalization & Denormalization
GPUs
Graph DBs
BigData
BI
OLAP
DM
Deep Learning
ETL
Data Modeling
Security
Authentication
Network Encryption
DBA Issues
Disk/Data Encryption
Malicious Attack
Data Masking
Audit & Monitoring
Authorization
Security Bugs
Replication
Sync
Async
Physical (Arch. log) Replication
Trigger Base
Disk Base
Proxy Base
Trigger Base
Hardware
CPU
GPU
SAN
SSD
SATA
RAID
Cloud Infrastructure
VMs
Dedicate / Shared Systems
NIC
Memory
by Mustafa YAVUZ