# Introduction:

When not to use SQL:
- Needing high availability in the data: system always up; no SPF
- Have large amounts of data
- Need linear scalability: vertical scaling → adding more nodes, linearly increase performance.
- Low latency.
  - Need fast read & write

# NoSQL Databases Explained!

↳ Not Only; accepts unstructured data

# Key-Value Database: uses "key" associated with one collection of values.

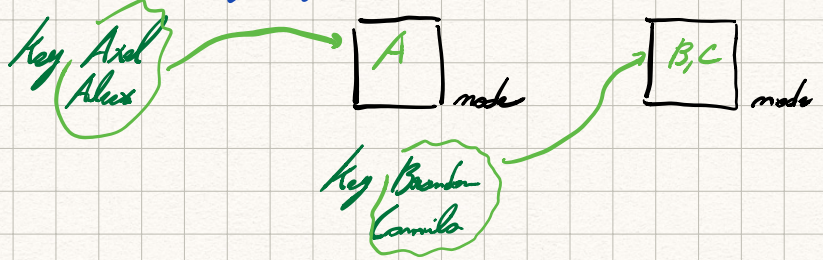
↳ may be a hash ↳ Stored in a blob

⚠ Key-value → have no "query language"

Store, retrieve & update data → set, get, delete

⚠ Key-value → "key" selection is critical → queries are performed just in how to it.

→ Similar "keys" go to nearby physical location → can lead to failure if no redundancy.




⚠ Consistency issues → if two machines have the key, there are problems of value consistency.

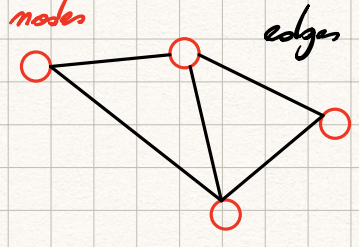
# Column-family Database: stores data in column families as rows. These rows have many columns associated with a particular row.

| Row Key X | Column 1 name1: value1 | Column 2 name2: value2 | Column N nameN: valueN |
|---|---|---|---|

| Row Key Y | Column 1 name1: value1 | Column 3 name3: value3 | Column N nameN: valueN |
|---|---|---|---|

- Compression: stores data efficiently through data compression and by using data partitioning
- Agg. Queries due to structure; aggregation operations are performant
- Scalability: more scalable than other DB.
- Fast to load & query.

# Graph DB: store data in the form of nodes & edges




Nodes → Entities

Edges → Relationship
- directional significance
- properties

→ Adding New node: easy

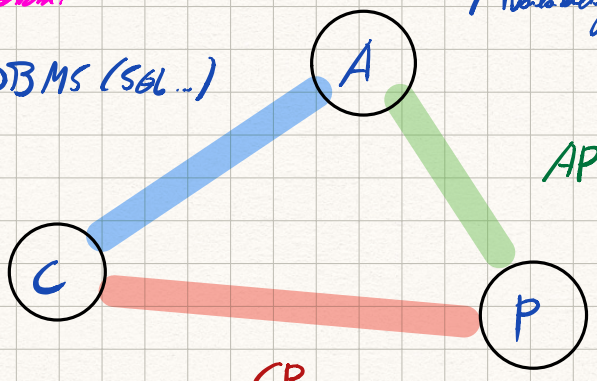
→ Modifying current node relationships: hard

# Choosing the right "No-SQL" Database:

→ Data you need to store → Data model

→ Data scaling expectations

# CAP Theorem:




CA: RDBMS (SQL...)

Availability: each client can always read & write

AP: Cassandra

Partition Tolerant: system works well despite physical network partition

Consistency: all clients have the same view of the data

CP
- HBase
- MongoDB

# Distributed Database:

→ In order to have higher availability → You need to have copies of your data.

↓ All copies may not be up to day

Eventual consistency: if no updates to a data item, eventually (i.e. not immediately), all access to that item return last updated value

↳ a consistency model in distributed computing

↳ If no new updates are made to new data, eventually all accesses will return last updated value.

→ Also called optimistic replication

⇒ When the system achieves eventual consistency, the system has converged / Replica convergence.

⚠ Conflict Resolution: systems of data reconciliation

↗ anti-entropy: exchanging versions & updates.

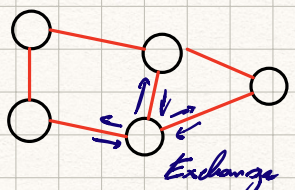
↳ reconciliation: choosing an appropriate final state when concurrent updates have occured.

Strategies → "last writer wins"
↳ "first writer wins"

When → read repair: correction is done when a read finds an incorrect write repair: correction takes place during write operation
asynchronous repair: correction is not part of read and write.

# Apache Cassandra:

→ Structure



Peer to Peer
homogeneous Nodes

Exchange
inter-node State information

Homogeneous Nodes

DATA

1. Sequential written commit log (per-server = "write activity")

2. Data is then indexed and written in a memory structure
   ↳ memtable

FLOW

When full

3. Data written to a disk in SSTable
   ↳ Automatically → Replicated
   ↳ Partitioned

→ Data: a partitioned row store database
- Require a primary key

→ When a petition arrives: Written in CQL (Similar to SQL),

the receiving node acts as "coordinator"

→ Data Replication: When an old copy of data is detected, while reading, it returns the most updated value and performs a "read repair" (slows reading speed to enforce consistency)

→ Keyspace: x Replication factor: nº of machines with a copy of data.
- Replica placement Strategy: strategy for replicas placement
  - Simple strategy
  - Old Network Topology Strategy
  - ↳ Network Topology Strategy
- Column families: a "table", each one containing many rows.
  ↳ Not defined from creation; columns can be added at any moment, to a give column family.

Column:

name: byte[] ; value: byte[] ; clock: byte[]

Super Column:

name: byte[] , cols: map < byte[], column >

As "super column" columns are physically saved together, better combine columns that are usually queried together.

# CAP Theorem:

Theorem in computer science that states it is impossible for a distributed data storage to simultaneously offer more than two out of the 3 guarantees:

- Consistency: every read from the database gets the latest (and correct) piece of data or an error.
- Availability: every request is received and responded; no data (most recent) guarantee
- Partition Tolerance: the system continues to work regardless of loosing network connectivity between nodes.

→ CAP Consistency vs ACID RDB?

ACID relates to database rules: Atomic all components of a transaction are treated together, either all succeed or none
Consistent transactions must follow DB rules (schema constrains)
Isolated concurrent execution of transactions is translated into a linear one
Durable committed transactions are NOT UNDONE

CAP: relates to the state of the system "vis-a-vis" replicas of a same value, across the node clusters for a given specific times

# Denormalizing in Apache Cassandra:

- Denormalization is critical in Apache Cassandra there are NO JOINS.
  → Need to think about your queries in order to build the tables.

Normalized Tables → Denormalized Tables
RDB → Apache Cassandra

Which are your queries?

# Data modeling:

- Denormalizing is not just ok → MUST
- ALWAYS Queries first
- Denormalizing done for fast reads
- One Table per query → good strategy
- It is optimized for fast Writes (performs read repair).
- NO JOINS

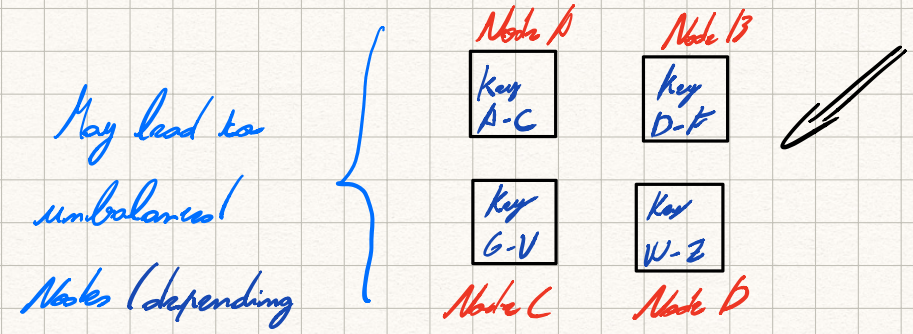
# PRIMARY KEY:

- Must be unique
- Composed of either just the PARTITION KEY // PARTITION KEY ⊕ CLUSTERING COLS.
  - Simple "Primary Key" → Just the PARTITION KEY
  - Composed "Primary Key"
- PARTITION KEY will determine distribution of data across the system.

Partition Key Row
- Simple → Primary Key
- Compos → Primary Key ⊕ Clustering Columns

⟹ HASH

↓ Stored on the node in the system that holds that range of values.




May lead to unbalanced Nodes (depending on Key Partitioning in Node).

↳ Pick a Key (Primary Key) that is more evenly distributed.

# Clustering Columns:

- Clustering column will sort the data in sorted ascending order.
- More than one clustering column can be added
- Clustering columns will sort in order of how they were added to the primary key.

⚠ Can use as many clustering columns as required; use them in the "SELECT STATEMENT" on the same order

Example:

```
CREATE TABLE music_library (year int, artist_name text, album_name text,
    PRIMARY KEY ((year), artist_name, album_name))
```

Composite key

(year) → Partition key

artist_name, album_name → Clustering columns (ordered ascending order).

# The WHERE clause:

- Data Modeling in Apache Cassandra is query focused!

The Partition key must be used; any "clustering columns" must be used in the order employed in the PARTITION KEY definition