Implementation considerations in making No SQL systems

pm_jat @ daiict

Data Distribution and Replication

- Data Distribution
  - What is?
  - Why do we do it? “Scaling”
  - How do we do it?
- Replication
  - What is?
  - Why do we do it? “Fault Tolerance”
  - How do we do it?
- What are the factors that affect and are considered while distribution and replication is done.
- Consistency Support

What is Data Distribution or Sharding?

- Sharding splits data on multiple nodes, each of **node does its own reads and writes**.

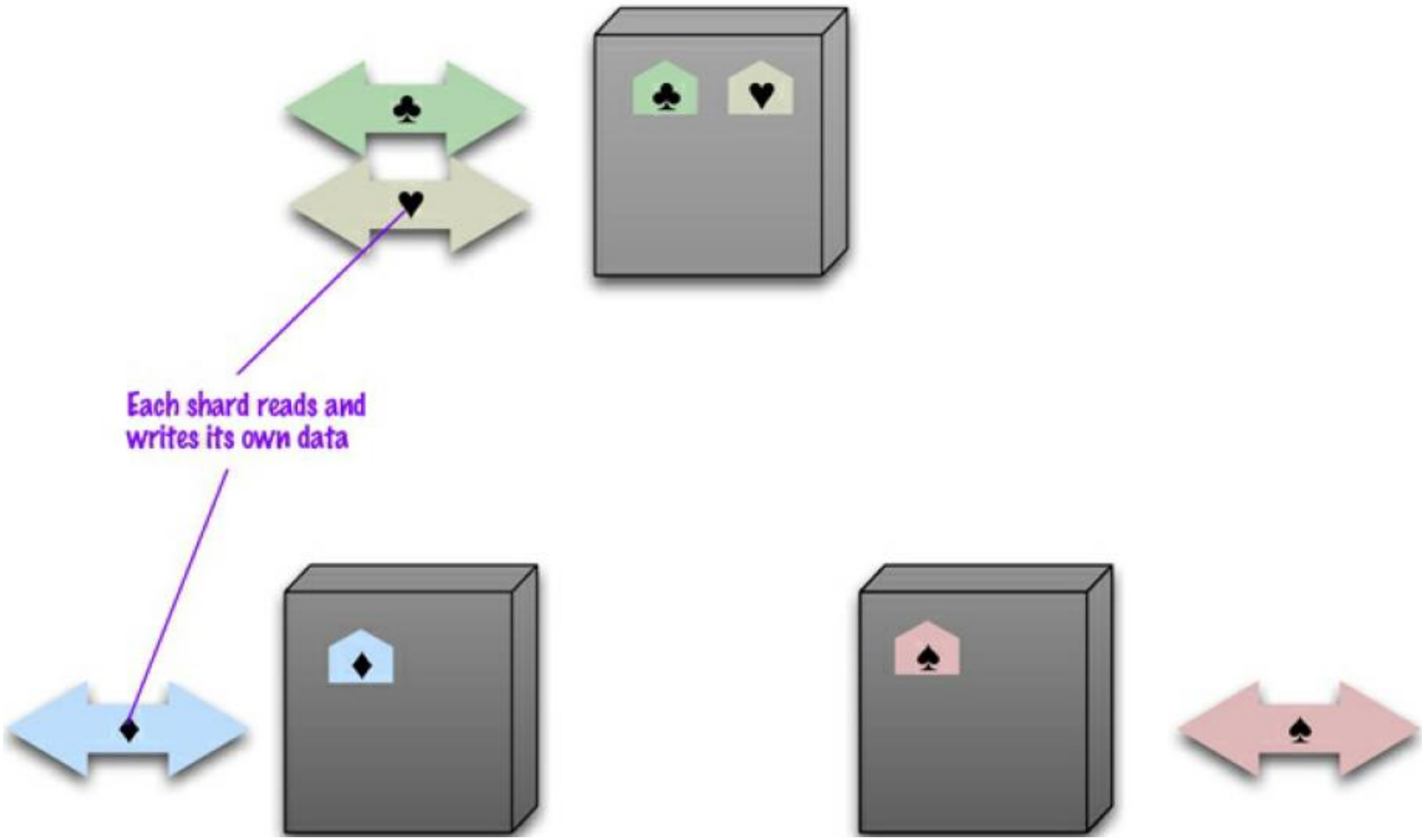


Figure source: No SQL Distilled

making No SQL systems

Data Distribution / Sharding

- What is it?
  - Database is split on multiple nodes, each of **node does its own reads and writes**.
- Why do we do it?
  - To scale out horizontally, “use cluster” for computation
- How do we do it?
  - Use some strategy, primarily “<u>hashing based on Key</u>”, and considers certain factors (shown next)

How do we do Sharding?

- Partitioning is be done such that -

  (1) a query can be answered from a single server
    – efficient query answering

  (2) each server gets workload evenly distributed
    – load balancing

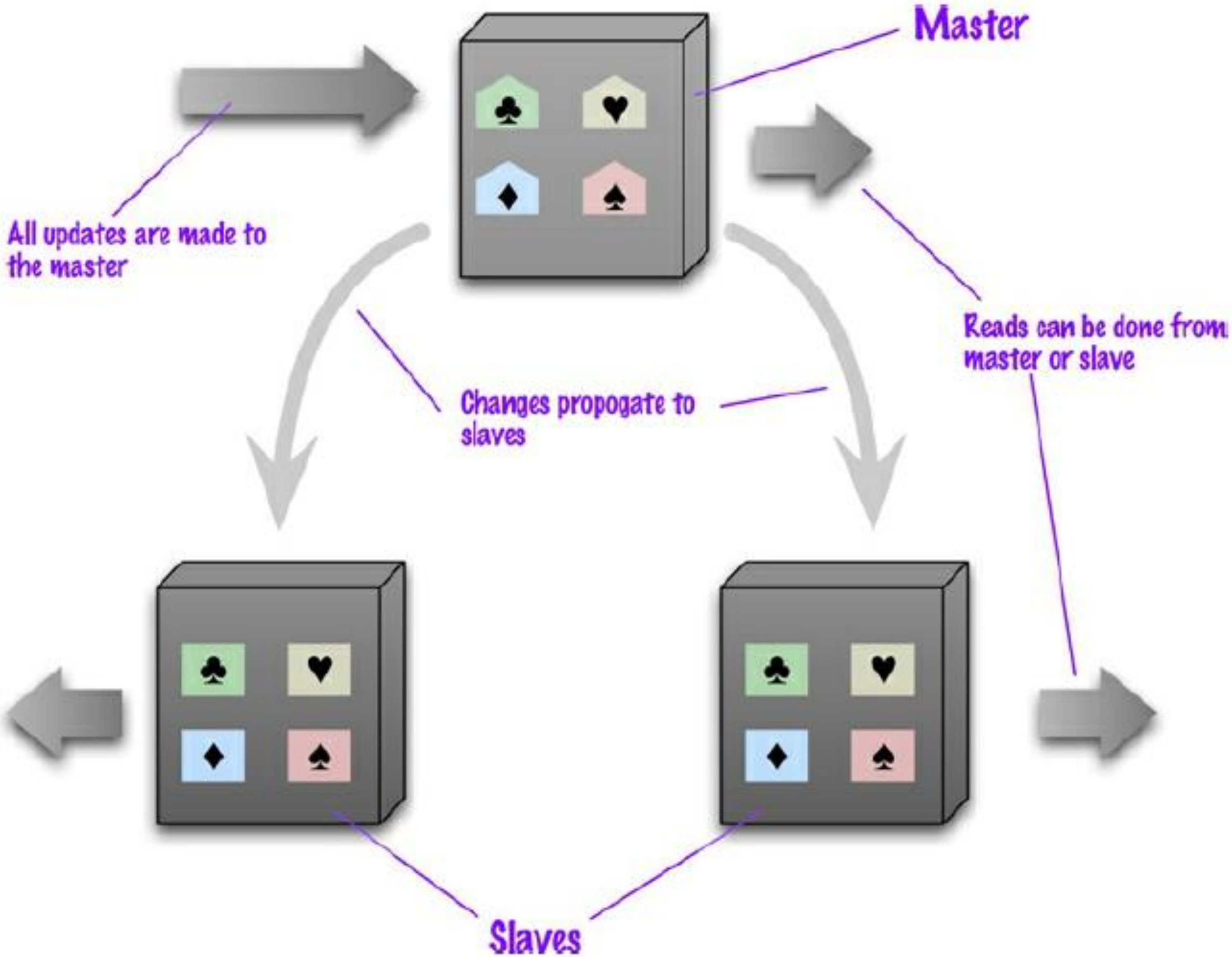
Replication

- What is?
  - Replicate data on multiple servers (nodes) with a replication factor, say 3.
- Why do we do it?
  - Fault tolerance – to deal with system and network failures.
  - If a server containing data fails, read/write can be done on other server.
- How do we deal with it?
  - Strategies like “Master-Slave”, and “Peer to Peer”
  - Requires going hand to hand with “Consistency Support”

Master-slave replication

- Let us understand in single server context
- We have few replicas - one of them act as Master node, and others as slaves
- Write are done only on Master node while read can be done from any other server
- A general strategy is reads are always routed to slave nodes
- <u>Mongo DB uses Master Slave Strategy</u>!

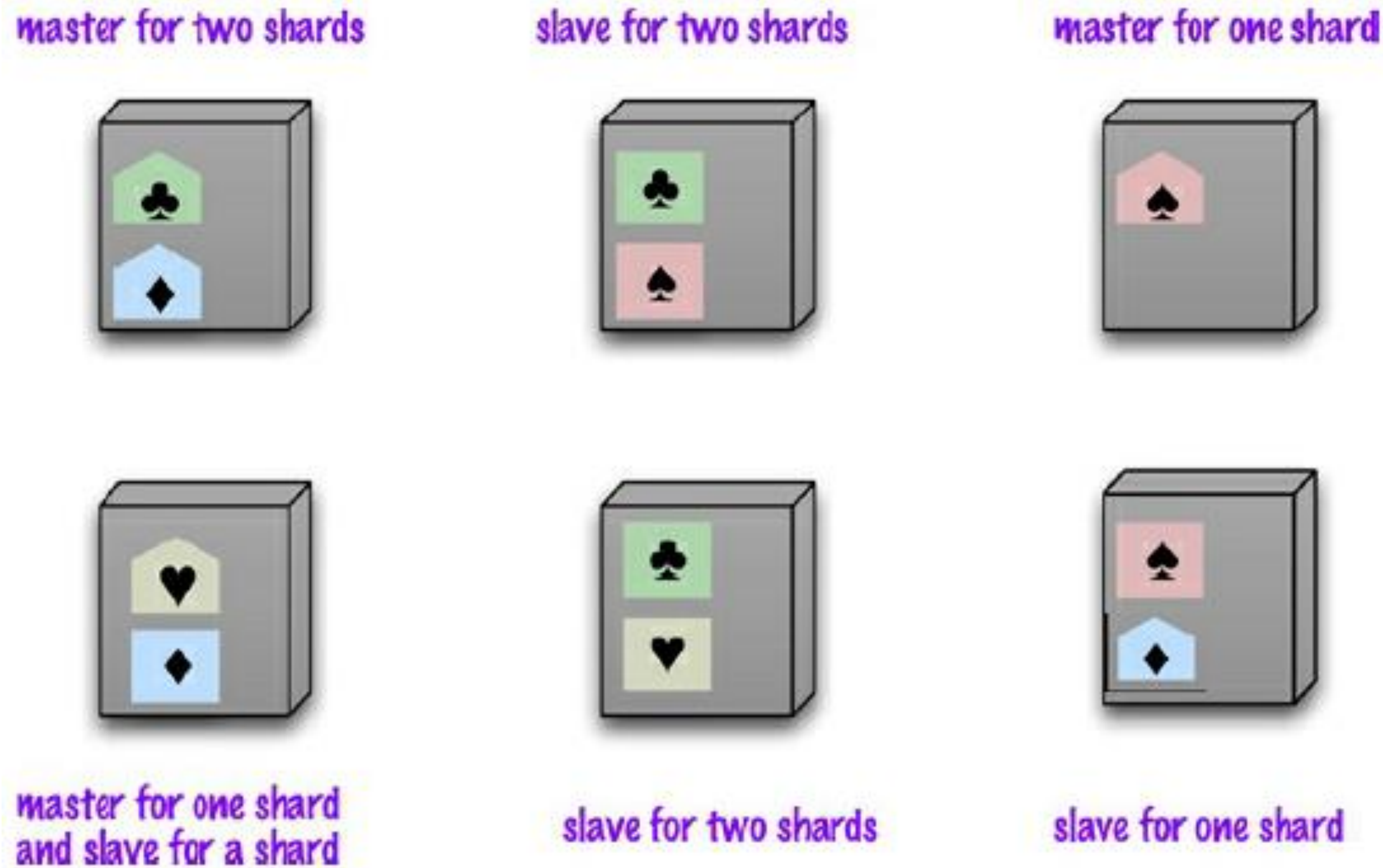
Master-slave replication



Master Slave with Sharding

- We have data partitions
- This means we have multiple masters, but each “data item” has only a single master.
- Depending on configuration, we may choose a node to be a master for some data and slaves for others, or we may dedicate nodes for master or slave duties only

Master-slave replication



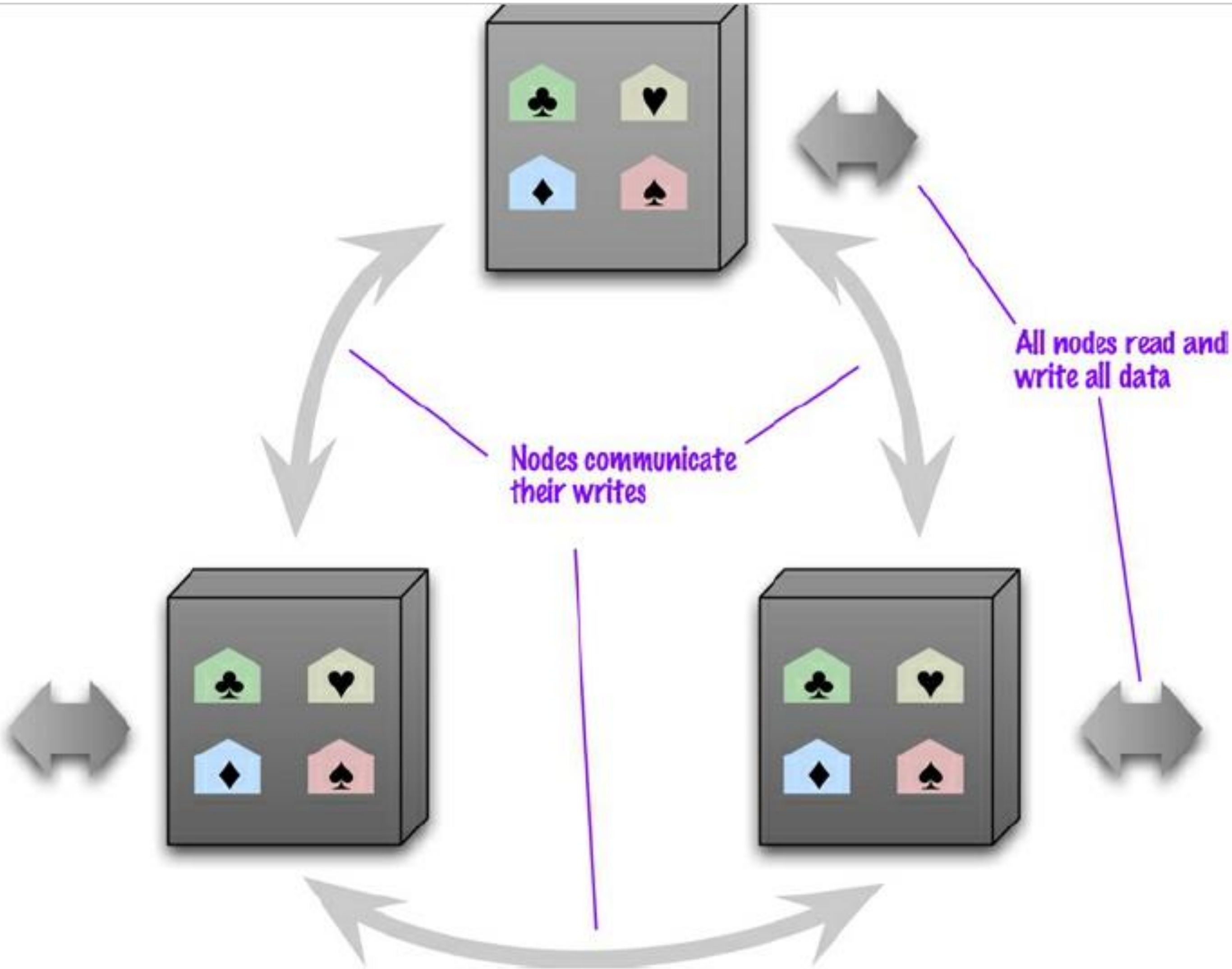
Master-slave replication

- "Writing ONLY ON master" brings in few concerns:
  - It can not scale out for writing, as writing is limited to master and master is only one.
  - If master node becomes unreachable, this adds to delays.
  - If master node fails, there is more serious problem. One of the slave can be made is master but, that adds significantly to the latency (delay), and
  - Still there is possibility of loosing data of master!
- However, strategy is good for reads. **Can scale well for reads**
  - reads can still be continued even when master node fails.

Peer to Peer Replication

- Master-slave replication helps with read scalability but doesn't help with scalability of writes.
- It provides resilience against failure of a slave, but not of a master.
- This essentially makes master as single point of failure.
- Peer-to-peer replication attacks this problem by not having a master.
- All the replicas have equal weight, they can all accept writes, and the loss of any of them doesn't prevent access to the data store.

Peer to Peer Replication



making No SQL systems

Peer to Peer Replication

- The peer-to-peer arrangement looks very fine
  - We can perform read and write on any node
  - This take care of any node failing
  - We can improve the performance by simply adding more nodes, all nodes are equal
- Dynamo DB uses Peer to Peer Strategy.

Problems with Replication

- Replication brings in a primary problem of "**inconsistency**".
- Different clients reading different slaves, and hence may be reading different values (there is always possibility of replicas being inconsistent; eventually though they will not)
- In the worst case, a client cannot read a write that it just made. **GOT IT**?
  – Your program write a data item X, and immediately attempts reading it, and
  – on READ, since it may read from different replica that has not got update so far, will not get what itself wrote?

Problems with Replication - Problems

- Consistency problems becomes higher in Peer-to-Peer replication
- While “stale read” is one. It might also bring in WRITE-WRITE CONFLICT and that leads to things like “LOST UPDATE” or so.

A question?

- Do we still would use No SQL systems, where we have
  - No Sharding
  - No Replication
- Yes!
- For
  - Ease of programming - Impedance mismatch
  - For special purpose data like graph databases!
  - These systems may turn out to be more efficient than Relational systems for simple “key value” based database access patterns.

Consistency … what is it?

- Here is what we learnt in context classical databases
- Does not have “dirty read” and “lost update”
- Consistent read (repeatable read) in a transaction
- Read should be reading latest update of data
- Relational databases support ACID properties.
  - Atomic execution of transaction
  - Ensure Serializability (for isolation), and
  - So forth

Consistency and No SQL databases

- The “traditional” database systems that are based on monolithic architecture implements consistencies very well
  - ACID compliant systems provide “strict consistency”.
- On the other hand, modern web scale systems that have “partitioned” and “replicated” data struggle to provide such a consistency.
- CAP theorem proposed by Eric Brewer [2] show that “high-availability” and “consistency” conflict when data are “partitioned” and “replicated” on the network.
- Therefore, sacrifice on consistency is acceptable in No-SQL systems in favor of scaling and availability.

CAP Theorem[2]

- **Consistency**: degree of consistency after the execution of an update operation.
- A distributed system is typically considered to be consistent if after an update operation of some writer, all readers see updated value. [shall see more in a moment]
- **Availability** is System's ability to deal with node failure; i.e. system is till available with all of its data even if some nodes, or any other hardware failure (or down due to some maintenance work)

CAP Theorem[2]

- **Partition Tolerance ("Network Tolerant")**
  - It is system's ability to recognize additional part of data becoming available temporarily or permanently
  - It is system's ability to deal with dynamic addition or deletion of nodes

- CAP Theorem says that: A distributed database systems can have at most two of these properties!

CAP Theorem[2]

Choice	Traits	Examples
Consistence + Availability (Forfeit Partitions)	2-phase-commit cache-validation protocols	Single Server databases Cluster databases LDAP xFS file system
Consistency + Partition tolerance (Forfeit Availability)	Pessimistic locking Make minority partitions unavailable	Classical Distributed Databases Distributed locking Majority protocols
Availability + Partition tolerance (Forfeit Consistency)	expirations/leases conflict resolution optimistic	Cluster friendly databases Web caching[sic!] DNS

BASE as new consistency descriptor for No SQL systems

- Eric Brewer [2] also presents BASE, acronym for **B**asically **A**vailable, **S**oft-state, **E**ventual consistency.
- Presents a comparison with ACID (next slide)

ACID vs. BASE

Inktomi

ACID	BASE
◆ Strong consistency	◆ Weak consistency – stale data OK
◆ Isolation	◆ Availability first
◆ Focus on "commit"	◆ Best effort
◆ Nested transactions	◆ Approximate answers OK
◆ Availability?	◆ Aggressive (optimistic)
◆ Conservative (pessimistic)	◆ Simpler!
◆ Difficult evolution (e.g. schema)	◆ Faster
	◆ Easier evolution

← But I *think* it's a *spectrum* →

PODC Keynote, July 19, 2000

Consistency in “Distributed” and “Replicated” context

- Understanding of Consistent requires revisit in the context of “Distributed” and “Replicated” databases.
- The article “Eventual Consistency”[3] describes following of consistency models in this context.
  - Strict Consistency
  - Weak Consistency
  - Eventual Consistency
  - Casual Consistency, Read Your Write Consistency, …
- <u>These are in client’s view of “Consistency”.</u>

Consistency Models [3]

- Before looking into lets establish the context
  - Database is replicated with a factor of K
  - There are processes A, B, and C, such that
    - A write a data item X
    - B and C read that data item
- **Strict|Strong consistency** requires that after update of X is confirmed, any subsequent read (by A, B, or C) will get the updated value.

Consistency Models [3]

- **Weak consistency**: in this case, system does not guarantee subsequent reads getting the updated value.
- It takes a while before every replica gets updated by a write.
- Time interval of update and every replica getting updates is called "*inconsistency window*"
- **Eventual consistency:** it is a specific form of weak consistency; the storage system guarantees that if new updates are made to the object, eventually all reads will get the correct (updated) version; that is when "inconsistency windows" gets over!
- The duration of "inconsistency window" depends on factors like communication delays, the load on the system, and the replication factor (number of replicas are involved).

Consistency Models [3]

- The eventual consistency model has a number of variations that are important to consider:
- **Causal consistency:** If process A has communicated to process B that it has updated a data item, a subsequent access by process B will get the updated value.
- However, read by process C that has no information about A's write, may not get the updated value of (Eventually, of course!)
- **Read-your-writes consistency** is an important model where process A, after having updated a data item, always accesses the updated value and never sees an older value.
- This is a special case of the causal consistency model

Consistency Models [3]

- **Session consistency**: this is a practical version of the previous model, where a process accesses the storage system in the context of a session.
- As long as the session exists, the system guarantees “read-your-writes consistency”
- However, if the session terminates because of a certain failure, guarantee is not carried forward to newer session.
- **Monotonic read consistency**. If a process has seen a particular value for the object, any subsequent accesses will never return any previous values.

Consistency Models [3]

- **Monotonic write consistency**. In this case, the system guarantees to serialize the writes by the same process.
- Systems that do not guarantee this level of consistency are notoriously difficult to program.
- Real systems may be implementing multiple models of these consistencies!

References/Further Readings

[1] Chapter 4 and 5 of book *NoSQL distilled*.

[2] Brewer, Eric A.: *Towards Robust Distributed Systems*. Portland, Oregon, July 2000. –Keynote at the ACM Symposium on Principles of Distributed Computing (PODC) on 2000-07-19.

[3] Vogels, Werner. "Eventually consistent." *Communications of the ACM* 52.1 (2009): 40-44.
http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[4] Lloyd, Wyatt, et al. "A short primer on causal consistency." *USENIX; login magazine* 38.4 (2013): 41-43.