NoSQL systems: sharding, replication and consistency

Riccardo Torlone
Università Roma Tre
Data distribution

- NoSQL systems: data distributed over large clusters
- Aggregate is a natural unit to use for data distribution
- Data distribution models:
  - Single server (is an option for some applications)
  - Multiple servers
- Orthogonal aspects of data distribution:
  - Sharding: different data on different nodes
  - Replication: the same data copied over multiple nodes
    - master-slave
    - peer-to-peer
Sharding

- Different parts of the data onto different servers
  - Horizontal scalability
  - Ideal case: different users all talking to different server nodes
  - Data accessed together on the same node — aggregate unit!
- Pros: it can improve both reads and writes
- Cons: Clusters use less reliable machines — resilience decreases
Improving performance

Main rules of sharding:

1. Place the data close to where it’s accessed
   - Orders for Boston: data in your eastern US data center

2. Try to keep the load even
   - All nodes should get equal amounts of the load

3. Put together aggregates that may be read in sequence
   - Same order, same node

- Many NoSQL databases offer auto-sharding
  - the database takes on the responsibility of sharding
Master-Slave Replication

- **Master**
  - is the authoritative source for the data
  - is responsible for processing any updates to that data
  - can be appointed manually or automatically

- **Slaves**
  - A replication process synchronizes the slaves with the master
  - After a failure of the master, a slave can be appointed as new master very quickly
Pros and cons of Master-Slave Replication

• Pros
  • More read requests:
    • Add more slave nodes
    • Ensure that all read requests are routed to the slaves
  • Should the master fail, the slaves can still handle read requests
  • Good for datasets with a read-intensive dataset

• Cons
  • The master is a bottleneck
    • Limited by its ability to process updates and to pass those updates on
    • Its failure does eliminate the ability to handle writes until:
      • the master is restored or
      • a new master is appointed
  • Inconsistency due to slow propagation of changes to the slaves
  • Bad for datasets with heavy write traffic
Peer-to-Peer Replication

- All the replicas have equal weight, they can all accept writes.
- The loss of any of them doesn’t prevent access to the data store.
Pros and cons of peer-to-peer replication

• Pros:
  • you can ride over node failures without losing access to data
  • you can easily add nodes to improve your performance

• Cons:
  • Inconsistency!
    • Slow propagation of changes to copies on different nodes
      • Inconsistencies on read lead to problems but are relatively transient
      • Two people can update different copies of the same record stored on different nodes at the same time - a write-write conflict.
      • Inconsistent writes are forever.
Sharding and Replication on MS

- We have multiple masters, but each data only has a single master.
- Two schemes:
  - A node can be a master for some data and slaves for others
  - Nodes are dedicated for master or slave duties
Sharding and Replication on P2P

- Usually each shard is present on three nodes
- A common strategy for column-family databases
Key points

- There are two styles of distributing data:
  - Sharding distributes different data across multiple servers
    - each server acts as the single source for a subset of data.
  - Replication copies data across multiple servers
    - each bit of data can be found in multiple places.
- A system may use either or both techniques.
- Replication comes in two forms:
  - Master-slave replication makes one node the authoritative copy that handles writes while slaves synchronize with the master and may handle reads.
  - Peer-to-peer replication allows writes to any node; the nodes coordinate to synchronize their copies of the data.
- Master-slave replication reduces the chance of update conflicts but peer-to-peer replication avoids loading all writes onto a single point of failure.
Consistency

- Biggest change from a centralized relational database to a cluster-oriented NoSQL
  - Relational databases: strong consistency
  - NoSQL systems: mostly eventual consistency
Update Consistency

- **Write-write conflict**: two people updating the same data item at the same time
- If the server serialize them (sequential consistency):
  - One is applied and immediately overwritten by the other
  - Lost update
- Solutions:
  - **Pessimistic** approach
    - Prevent conflicts from occurring
    - Usually implemented with write locks managed by the system
  - **Optimistic** approach
    - Lets conflicts occur, but detects them and takes action to sort them out
    - Approaches:
      - conditional updates: test the value just before updating
      - save both updates: record that they are in conflict and then merge them
- Do not work if there’s more than one server (peer-to-peer replication)
Pessimistic vs optimistic approach

- Concurrency involves a fundamental tradeoff between:
  - safety (avoiding errors such as update conflicts) and
  - liveness (responding quickly to clients).
- Pessimistic approaches often:
  - severely degrade the responsiveness of a system
  - leads to deadlocks, which are hard to prevent and debug.
Read Consistency (or read-write conflict)

- A read in the middle of two logically-related writes

- **Logical consistency**: no read or read-write conflicts
Transactions on NoSQL databases

- Graph databases tend to support ACID transactions
- Aggregate-oriented NoSQL database:
  - Support atomic updates, but only within a single aggregate
  - To avoid inconsistency: orders in a single aggregate
  - Update over multiple aggregates: possible inconsistent reads
  - **Inconsistency window**: length of time an inconsistency is present

- Amazon’s documentation:
  - ..inconsistency window for SimpleDB service is usually less than a second..
Replication consistency

• Replication is another source of inconsistency

• **Eventual consistency**: at any time nodes may have replication inconsistencies but, if there are no further updates, eventually all nodes will be updated to the same value

• **Stale** data: out of date
Logical and eventual consistency

- Eventual consistency is independent from logical consistency but replication can lengthen the inconsistency window.
  - Two updates on the master performed in rapid succession
  - Delays in networking can lengthen the inconsistency on a slave
- Consequences of inconsistency windows
  - Different people see different data at the same time: usually tolerated
  - Read-your-writes consistency: should be guaranteed
- **Session consistency**: within a user’s session
  - Sticky session: tied to one node (session affinity)
  - Version stamps: every interaction latest version stamp
Version stamps

- Help you detect concurrency conflicts.
- When you read data, then update it, you can check the version stamp to ensure nobody updated the data between your read and write.
- Version stamps can be implemented using counters, GUIDs (a large random number that’s guaranteed to be unique), content hashes, timestamps, or a combination of these.
- With distributed systems, a vector of version stamps (a set of counters, one for each node) allows you to detect when different nodes have conflicting updates.
The CAP Theorem

- Why you may need to relax consistency
- Proposed by Eric Brewer in 2000
- Formal proof by Seth Gilbert and Nancy Lynch in 2002

"Given the properties of Consistency, Availability, and Partition tolerance, you can only get two"

- **Consistency**: all people see the same data at the same time
- **Availability**: if you can talk to a node in the cluster, it can read and write data
- **Partition tolerance**: the cluster can survive communication breakages that separate the cluster into partitions unable to communicate with each other
Network partition
CA systems

- A single-server system is the obvious example of a CA system
- CA cluster: if a partition occurs, all the nodes would go down
  - A failed, unresponsive node doesn’t infer a lack of CAP availability
- A system that suffer partitions: tradeoff consistency VS availability
  - Give up to some consistency to get some availability
An example

- Ann is trying to book a room of the Ace Hotel in New York on a node located in London of a booking system
- Pathin is trying to do the same on a node located in Mumbai
- The booking system uses a peer-to-peer distribution
- There is only a room available
- The network link breaks
Possible solutions

- **CP**: Neither user can book any hotel room, sacrificing availability
- **caP**: Designate Mumbai node as the master for Ace hotel
  - Pathin can make the reservation
  - Ann can see the inconsistent room information
  - Ann cannot book the room
- **AP**: both nodes accept the hotel reservation
  - Overbooking!
- These situations are closely tied to the domain
  - Financial exchanges? Blogs? Shopping charts?
- **Issues**:
  - how tolerant you are of stale reads
  - how long the inconsistency window can be
- **BASE approach** (Basically Available, Soft state, Eventual consistency)
Durability

- You may want to trade off durability for higher performance
  - Main memory database: if the server crashes, any updates since the last flush will be lost
  - Keeping user-session states as temporary information
  - Capturing telemetric data from physical devices
- Replication durability: occurs when a node processes an update but fails before that update is replicated to the other nodes.
How many nodes need to be involved to get strong consistency?
- Write quorum: $W > \frac{N}{2}$
- The number of nodes participating in the write ($W$) must be more than the half the replication factor ($N$)

How many nodes you need to contact to be sure you have the most up-to-date change?
- Read quorum: $R + W > N$

In a master-slave distribution one $R/W$ is enough
- A replication factor of 3 is usually enough to have good resilience
Key points

- Write-write conflicts occur when two clients try to write the same data at the same time. Read-write conflicts occur when one client reads inconsistent data in the middle of another client’s write.
- Pessimistic approaches lock data records to prevent conflicts. Optimistic approaches detect conflicts and fix them.
- Distributed systems (with replicas) see read-write conflicts due to some nodes having received updates while other nodes have not. Eventual consistency means that at some point the system will become consistent once all the writes have propagated to all the nodes.
- Clients usually want read-your-writes consistency, which means a client can write and then immediately read the new value. This can be difficult if the read and the write happen on different nodes.
- To get good consistency, you need to involve many nodes in data operations, but this increases latency. So you often have to trade off consistency versus latency.
- The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency.
- Durability can also be traded off against latency, particularly if you want to survive failures with replicated data.
- You do not need to contact all replicants to preserve strong consistency with replication; you just need a large enough quorum.