L3. NoSQL[1]

S1. Consistency

Dr A Boronat

[1] Materials based on [1, ch 5,9]

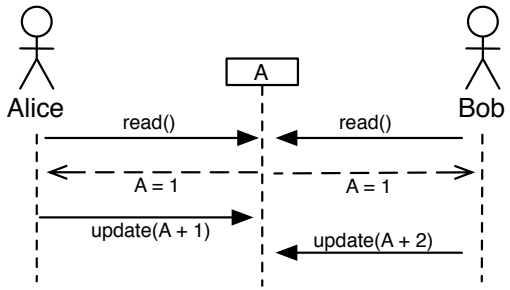
Table of Contents

Consistency

- A data store is a database that may be physically distributed across multiple servers
- A consistency model is a contract between applications and the data store: if applications agree to obey certain rules, the data store promises to work correctly
- Strong consistency
  - all accesses are seen by all applications in the same order (sequentially)
  - only one consistent global state
  - usually in single-node data servers
- No longer true in NoSQL settings
  - consistency can be traded for availability
  - at the expense of getting inconsistencies

Update Consistency: write-write conflicts

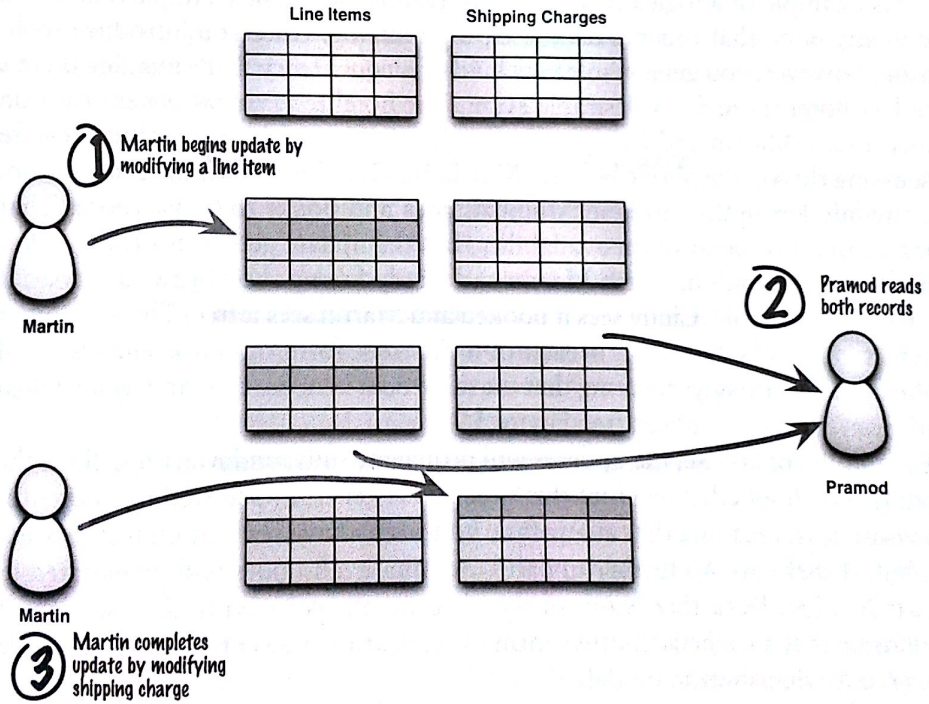
- Write-write conflict: two applications updating the same data item at the same time



- approaches assuming sequential consistency (all nodes apply operations in the same order)
  - when updates reach the server, it serializes them
  - trade-off between safety (avoiding errors, i.e. conflicts) and liveness (responding quickly to clients)
  - pessimistic approaches
    - lock data records to prevent conflicts
  - optimistic approaches
    - conditional update: test the value just before updating it to see if it's changed
    - allow conflicts and merge the updates

Read Consistency: read-write conflicts

- inconsistent read or read-write conflict



- aggregates support transactions
- approaches to deal with data in different aggregates
  - inconsistency window: length of time an inconsistency is present
  - different people will see different data at the same time
  - particularly problematic when you get inconsistencies with yourself, e.g. blog posting

Replication Consistency

Replication Consistency

- the same data item has the same value when read from different replicas
  - cache is a form of replication following the master-slave distribution model
  - stale data: out of date, not synced
- eventually consistency: when the updates propagate fully to all replicas
- replication can exacerbate logical inconsistency (update and read) by lengthening its inconsistency window

Read-your-writes consistency

- session consistency
  - sticky session
  - version stamps

CAP Theorem

- Consistency, Availability and Partition tolerance: pick any two
  - 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 partition the cluster into isolated components (split brain)
- Under CA
  - relational dbs
  - cluster: if a node fails, the whole cluster fails
- Main idea: trade consistency for availability
- Main idea (actually): trade consistency for latency
- NoSQL systems "change" ACID for BASE: Basically Available, Soft state, Eventual Consistency

Consistency vs availability: write consistency

- Totally consistent: write requests are serialized globally (synchronization must occur, i.e. communication)
  - availability: if a network link breaks, the system is not available
- Designate one node as master with master-slave replication
  - read inconsistencies may happen (due to stale data)
  - availability: if the network link to the master breaks, we cannot write
- Peer-to-peer: all nodes can receive writes
  - availability: if a network link breaks, the local node is independent and still works
  - read/update inconsistencies may happen
  - solution: enable domain-specific error handling policies

Consistency vs durability

Scenarios

- Web apps in-memory db can have smaller response time
  - storing user-session state
  - potential problems: loss of data if not flushed to a physical db or synchronized via HTTP
- Capturing telemetric data from physical devices
  - capture data at a faster rate at a cost of losing the last updates
- Replicated data: offline cache (browser) and online db (master-slave)
  - inconsistency when a node processes an update but fails before the update is replicated to the other nodes
  - replication durability can be improved by ensuring that the master waits for some replicas to acknowledge the update before the master acknowledges it to the client
  - slows down updates and makes the cluster unavailable if the slave fails

Quorums: write

- Idea: you don't need to contact all replicas to preserve strong consistency with replication
- Given
  - W = number of nodes participating in the write (confirming the write)
  - N (replication factor) = number of replicas
  - R = number of nodes you need to contact for a read
- Peer-to-peer distribution model
  - Write quorum is how many nodes need to acknowedge a write to ensure consistency
    - you need a majority, i.e. $W > N/2$
  - Read quorum is how many nodes you need to contact to ensure that you have the most up-to-date change
    - it depends on how many nodes need to confirm a write: $R + W > N$
    - we can get strongly consistent reads even if we don't have strong consistency on our writes
    - $N = 3$ ensures good consistency, usually
- Slave-master
  - write to the master: this avoids write-write conflicts
  - read from the master: to avoid read-write conflicts

MongoDB: consistency

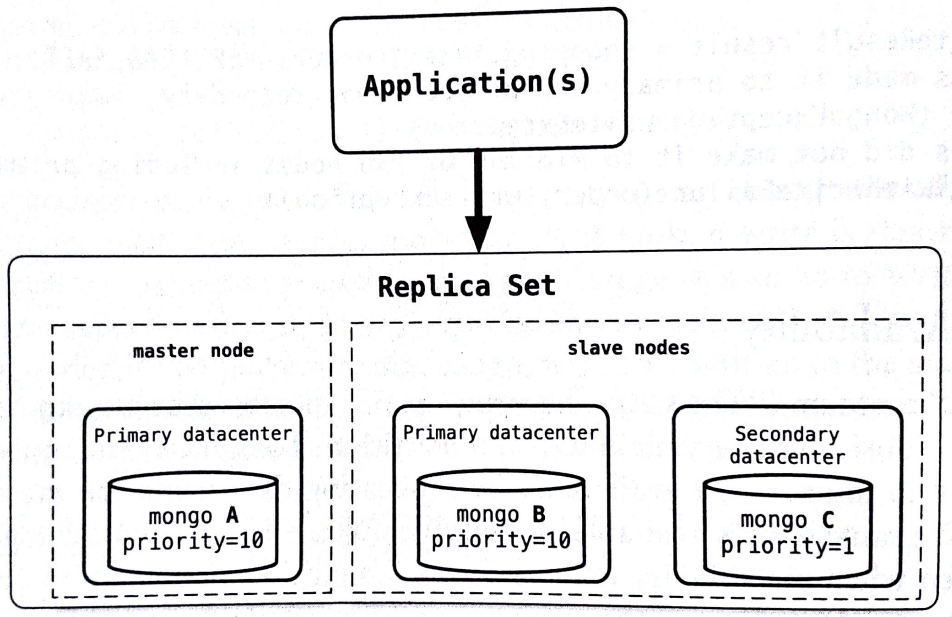
- MongoDB implements replication using replica sets
  - two or more nodes participating in an asynchronous master-slave replication
  - replica set nodes elect the master among themselves
  - when the master node goes down, a new master node is elected automatically (opaque to the application)

MongoDB: consistency

- Write consistency
  - wait for the writes to be replicated to all the slaves or a given number of slaves
  - we can specify W (number of replicas that need to acknowledge the write)
  - maximum value is majority
- Write consistency vs latency: a WriteConcern is used to indicate whether a client has to wait for the writes to be synced to disk or to propagate to two or more slaves
  - by default, a write is successful once the db receives it
  - with `coll.setWriteConcern(REPLICAS_SAFE)`, the writes are written to the master and some slaves
  - with `coll.insert(doc, REPLICAS_SAFE)`, we can specify the write concern per operation
- Read consistency vs latency: read from slaves with `MongoDB::slaveOk()` to increase performance
  - for all operations
  - per operation

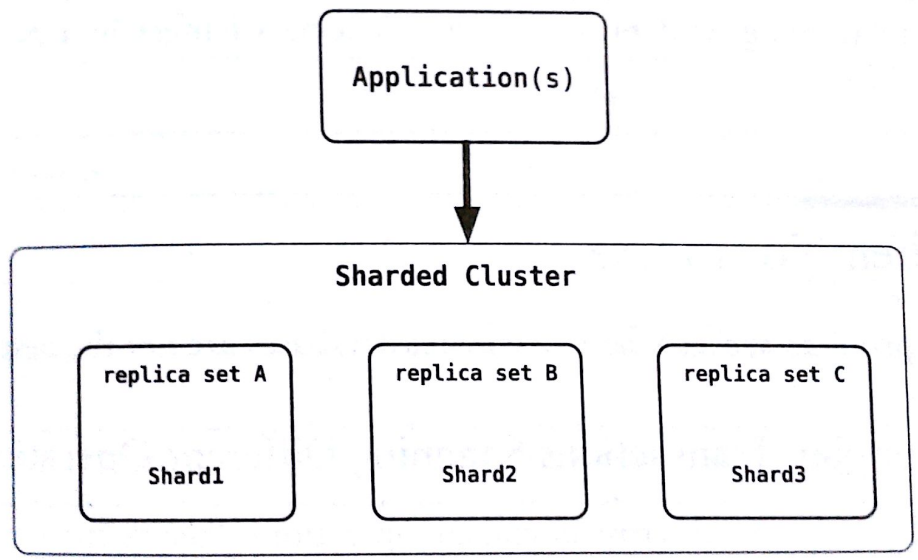
MongoDB: availability

- Replica sets are useful for
  - data redundancy
  - automated failover
  - read scaling
  - server maintenance without downtime
  - disaster recovery



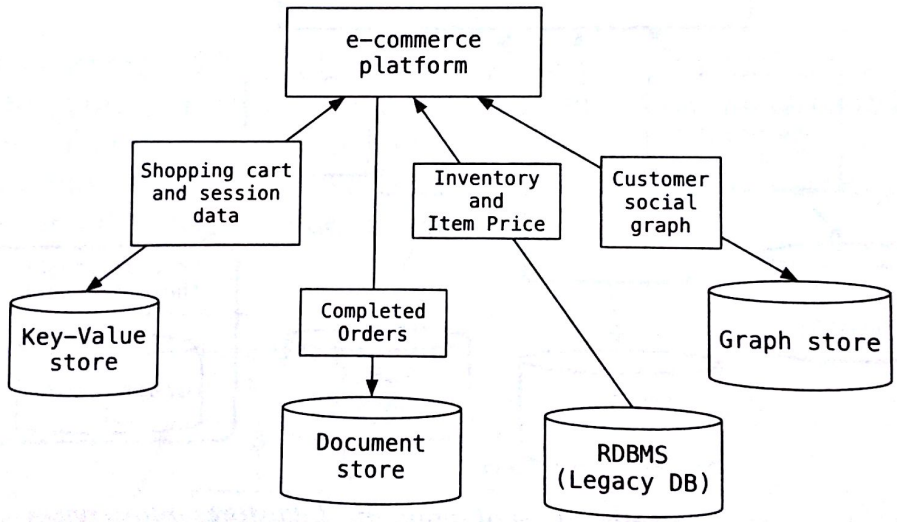
MongoDB: Scaling

- Horizontal scaling for reads
  - add more slaves to a replica set `ReplicaSet::add(URI)`
  - use `MongoDB::slaveOk()` to redirect reads to slaves
  - advantages
    - no need to restart application
    - no downtime
- Horizontal scaling for writes
  - use sharding to partition the store in disjoint (shards) by a particular field (key field)
  - data is dynamically moved between nodes to ensure that shards are always balanced
  - rebalancing does not affect availability (only latency)
  - we can add more nodes to cluster and increase the number of writeable nodes
  - each shard can have a replica set to scale out reads



Polyglot Persistence: do we need a schema?

- There is a wide choice of data storage solutions
  - encapsulating data access into services reduces the impact of data storage choices



- NoSQL stores facilitate sw development productivity
- NoSQL stores enable trading consistency for access performance
  - larger data volumes
  - reducing latency
  - improving throughput
- At the expense of shifting responsibility to the application (performance)
  - i.e. application complexity (such is life!)
  - that is, the schema only disappears from the db

References


Pramod J. Sadalage and Martin Fowler.
NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.
Addison-Wesley Professional, 1st edition, 2012.