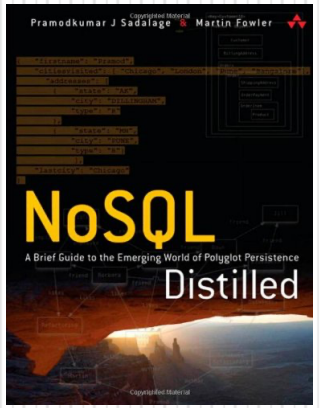
# NoSQL – Sharding, Replication & Consistency

By Riccardo Torlone

# NoSQL systems: sharding, replication and consistency

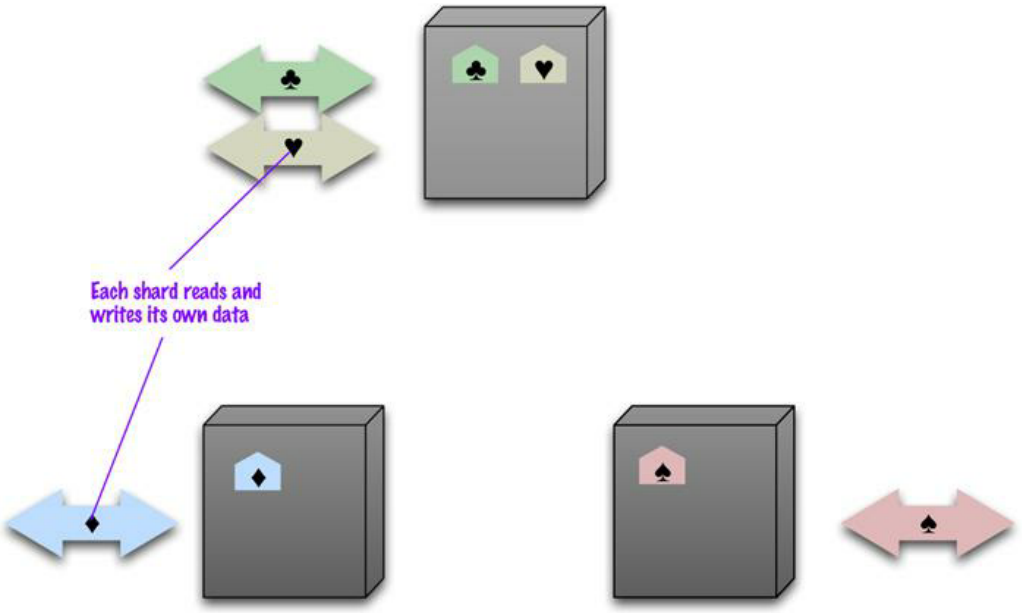
Riccardo Torlone
Università Roma Tre



# 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

# 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 data that may be read in sequence
   - Same order, same node

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

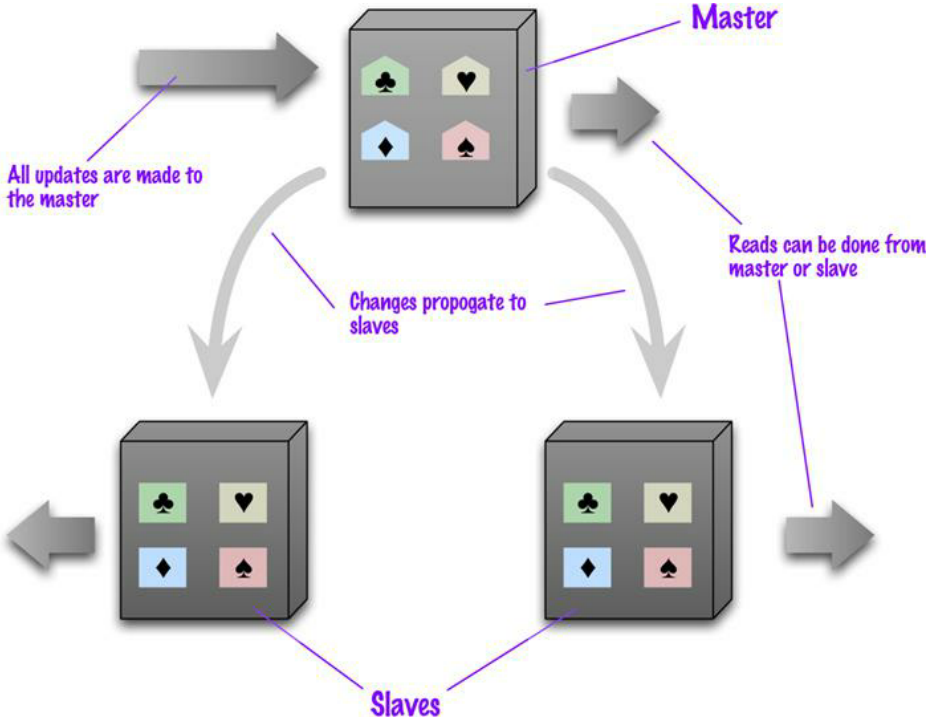
# Replication

It comes in two forms:

**master-slave**

**peer-to-peer**

# Master-Slave Replication

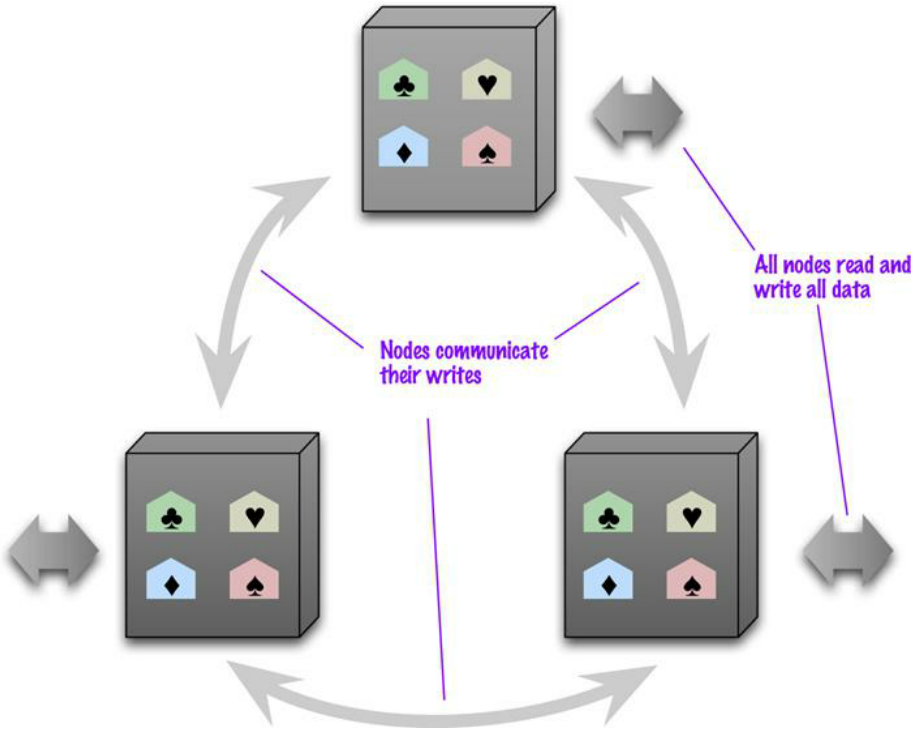



- Master
  - is the authoritative source for the data
  - is responsible for processing any updates to that data
  - can be appointed manually or automatically
- Slaves
  - A replication process synchronizes the slaves with the master
  - After a failure of the master, a slave can be appointed as new 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

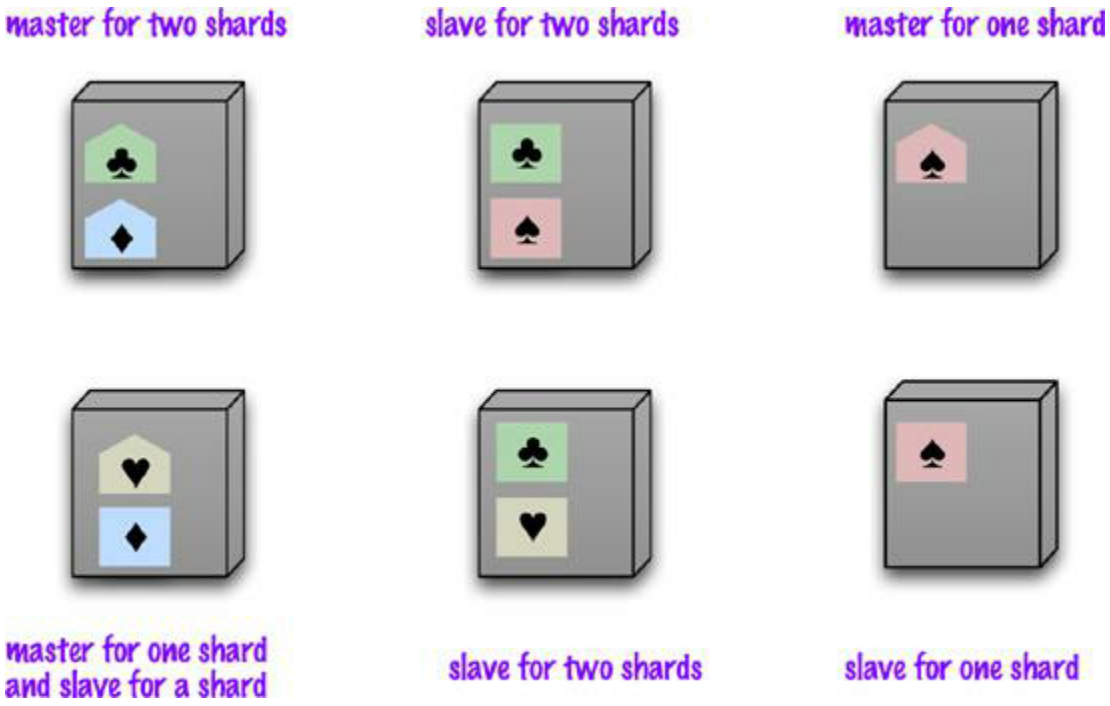



- All the replicas have equal weight, they can all accept writes
- The loss of any of them doesn't prevent access to the data store.

# Pros and cons of peer-to-peer replication

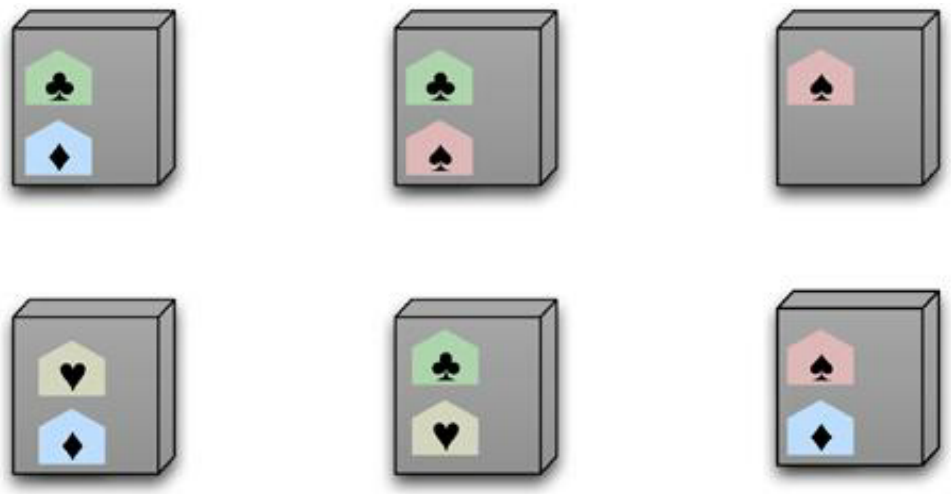
- Pros:
  - you can ride over node failures without losing access to data
  - you can easily add nodes to improve your performance
- Cons:
  - Inconsistency!
    - Slow propagation of changes to copies on different nodes
      - Inconsistencies on read lead to problems but are relatively transient
    - 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



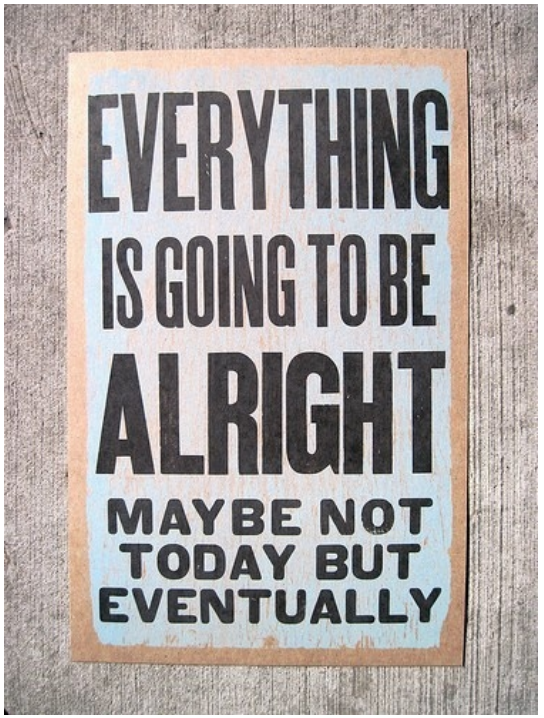
- Usually each shard is present on three nodes
- A common strategy for column-family databases

# 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

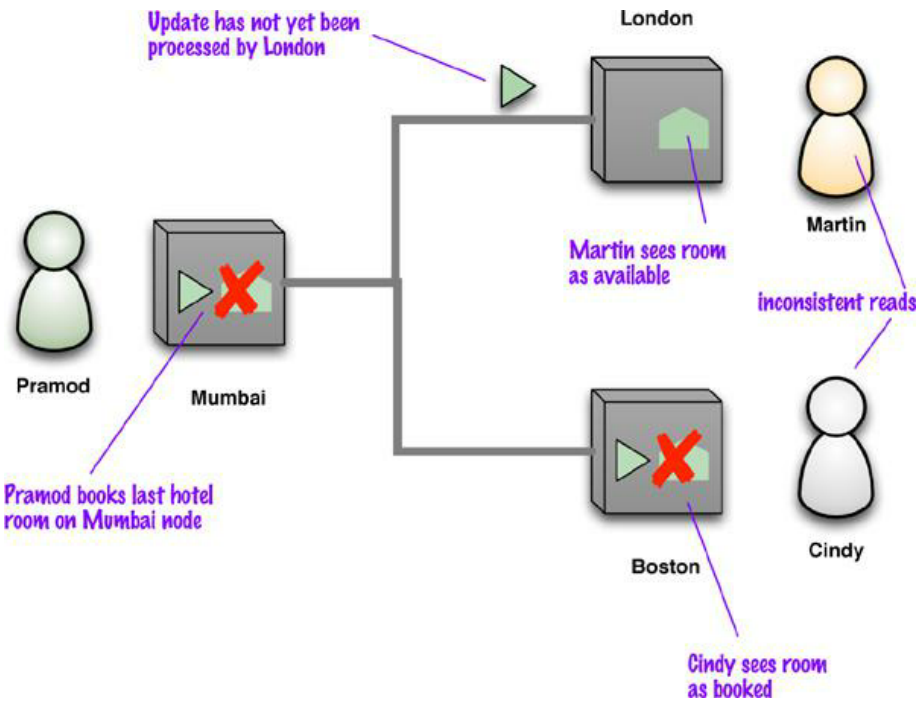



# Conflicts

- **Read-read (or simply read) conflict:**
  - Different people see different data at the same time
  - Stale data: out of date
- **Write-write conflict**
  - Two people updating the same data item at the same time
  - If the server serialize them: one is applied and immediately overwritten by the other (lost update)
- **Read-write conflict:**
  - A read in the middle of two logically-related writes

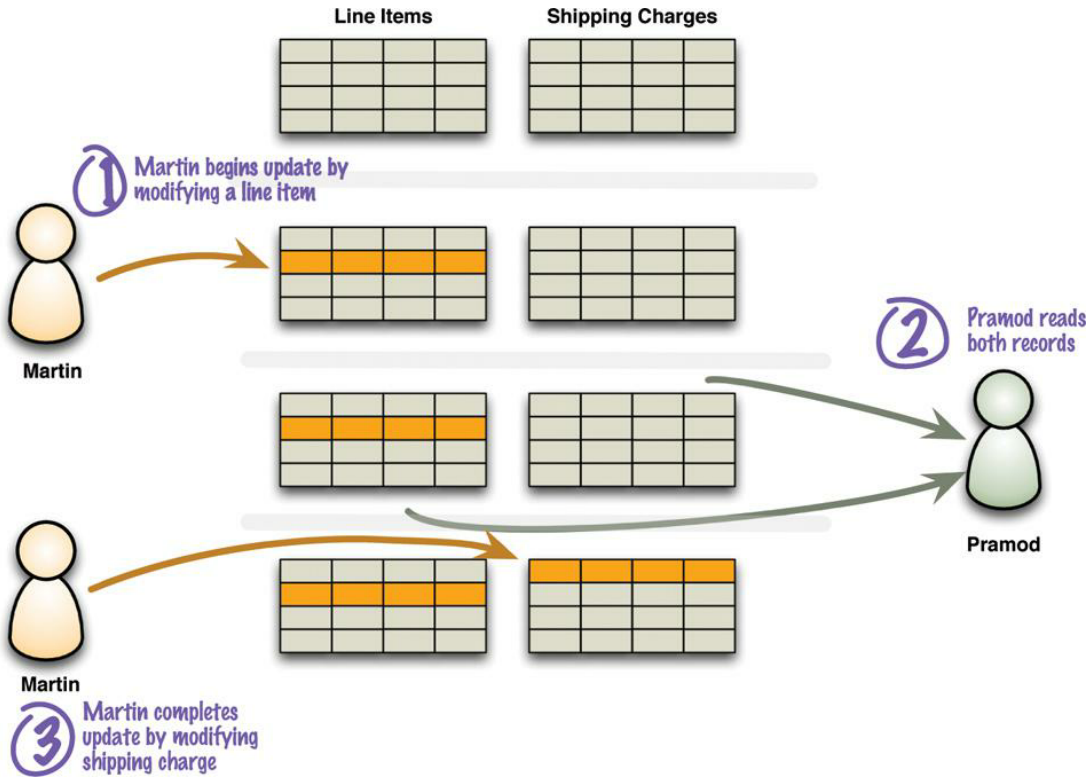
# Read conflict

- Replication is a source of inconsistency



# Read-write conflict

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



# 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 (for write-write conflicts):
    - conditional updates: test the value just before updating
    - save both updates: record that they are in conflict and then merge them

# Pessimistic vs optimistic approach

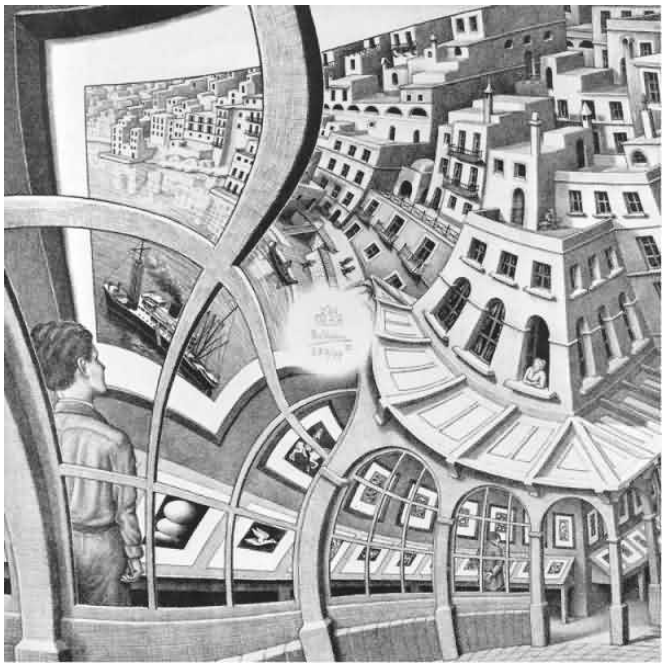
- Concurrency involves a fundamental tradeoff between:
  - consistency (avoiding errors such as update conflicts) and
  - availability (responding quickly to clients).
- Pessimistic approaches often:
  - severely degrade the responsiveness of a system
  - leads to deadlocks, which are hard to prevent and debug.

# Forms of consistency

- **Strong (or immediate) consistency**
  - ACID transaction
- **Logical consistency**
  - No read-write conflicts (atomic transactions)
- **Sequential consistency**
  - Updates are serialized
- **Session (or read-your-writes) consistency**
  - Within a user's session
- **Eventual consistency**
  - You may have replication inconsistencies but eventually all nodes will be updated to the same value

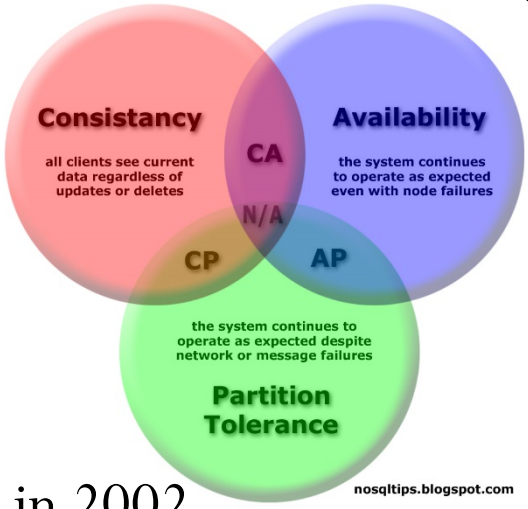
# Transactions on NoSQL databases

- Graph databases tend to support ACID transactions
- Aggregate-oriented NoSQL database:
  - Support atomic transactions, but only within a single aggregate
  - To avoid inconsistency in our example: 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..

# The CAP Theorem

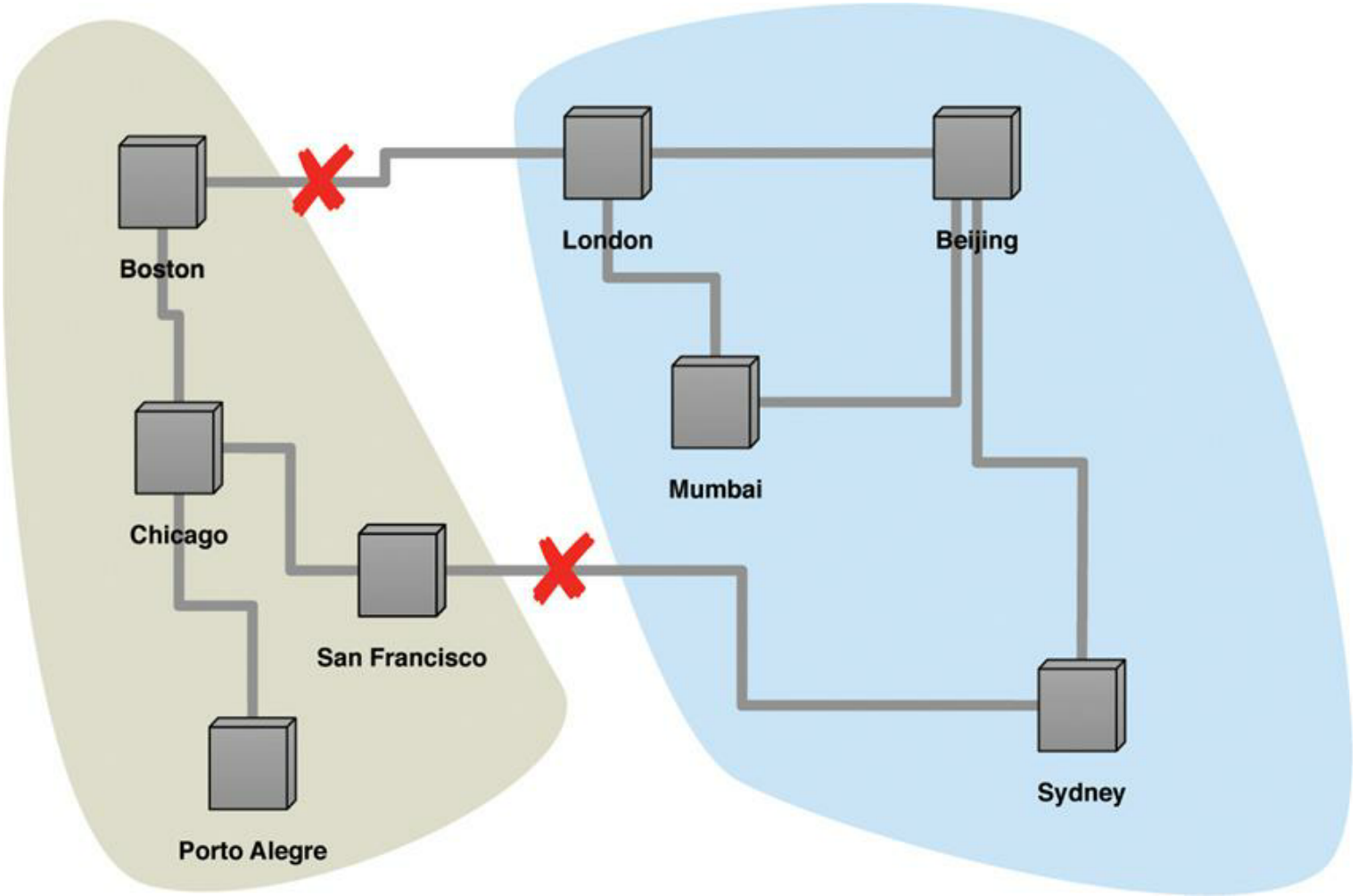



- Why you may need to relax consistency
- Proposed by Eric Brewer in 2000
- Formal proof by Seth Gilbert and Nancy Lynch in 2002

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

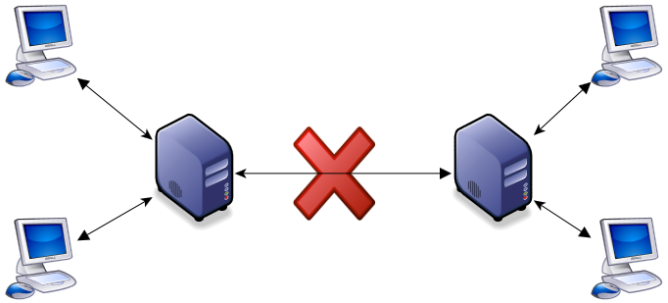
- **Consistency**: all people see the same data at the same time
- **Availability**: Every request receives a response – without guarantee that it contains the most recent write
- **Partition tolerance**: The system continues to operate despite communication breakages that separate the cluster into partitions unable to communicate with each other

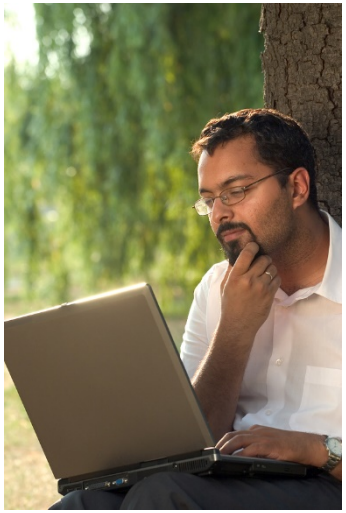
# Network partition



# 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

- CA: Neither user can book any hotel room
- CP: 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
- Common approach:
  - BASE (Basically Available, Soft state, Eventual consistency)

# CA vs AP systems

- A single-server system is the obvious example of a CA system
  - Traditional RDBMS
- CA cluster: if a partition occurs, all the nodes would go down
  - Data center (rare and partial partitions)
  - Availability in CAP: every request received by a non-failing node in the system must result in a response (a failed, unresponsive node doesn't infer a lack of availability)
- A system that suffer partitions:
  - Distributed cluster
  - Tradeoff between consistency VS availability
  - Give up to some consistency to get some availability
  - AP is usually the preferred choice since AC is less responsive

# 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 sensor data from physical devices
- Replication durability: occurs when a master processes an update but fails before that update is replicated to the other nodes.

# Practical approach: Quorums

- **Problem**: how many nodes you need to contact to be sure you have the most up-to-date change?
- N: replication factor, W: n. of confirmed writes, R: n. of reads that guarantees a correct answer
  - Read quorum: $R + W > N$
  - Write quorum: $W > N/2$
- How many nodes need to be involved to get consistency?
  - Strong consistency: $W = N$
  - Eventual consistency: $W < N$
- In a master-slave distribution one W and one R (to the master!) 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. Read conflicts occur when when two clients reads inconsistent data.
- 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 replicas to preserve strong consistency with replication; you just need a large enough quorum.