Distributed Data Management

Introduction to
data replication and
the CAP theorem

Replication

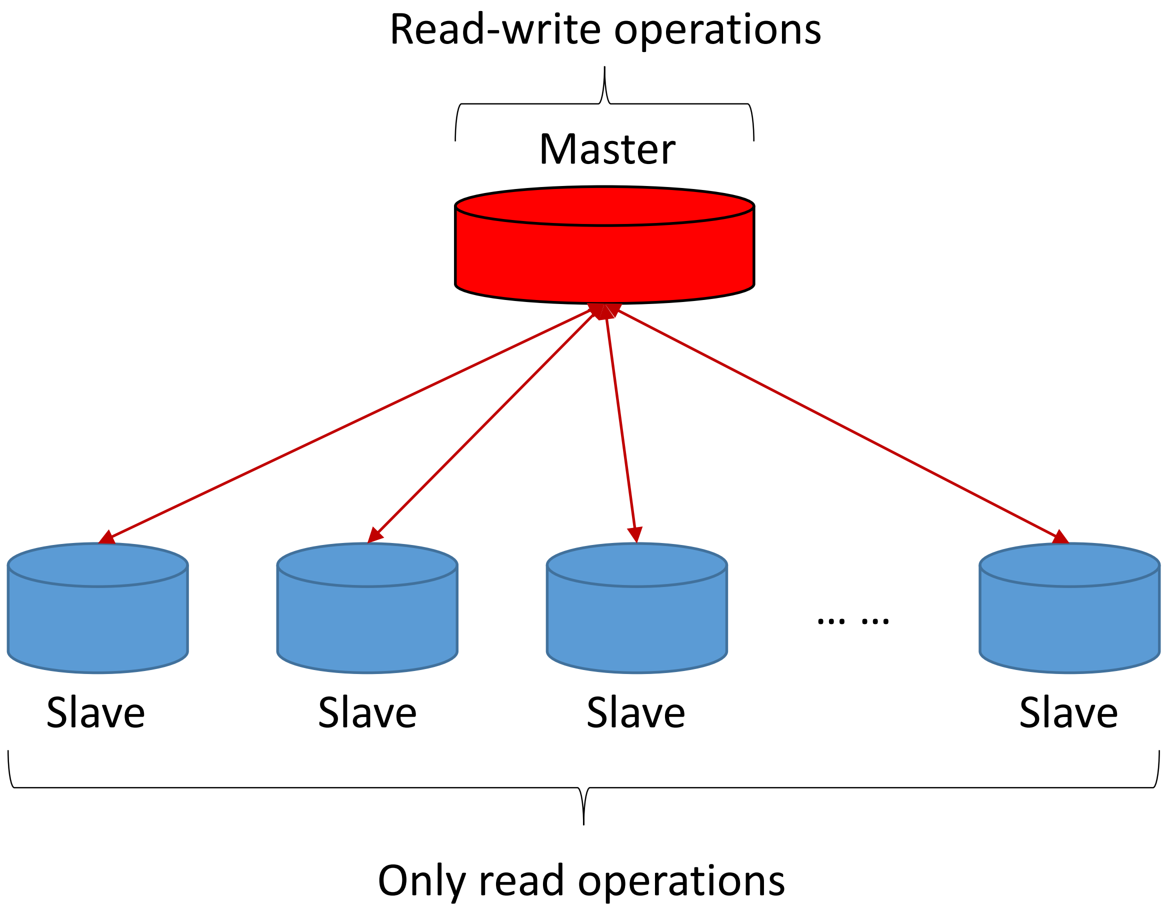
Same data
in **different** places

Replication

- **Same** data
  - portions of the whole dataset (chunks)
- in **different** places
  - local and/or remote servers, clusters, data centers
- Goals
  - Redundancy helps surviving failures (availability)
  - Better performance
- Approaches
  - Master-Slave replication
  - A-Synchronous replication

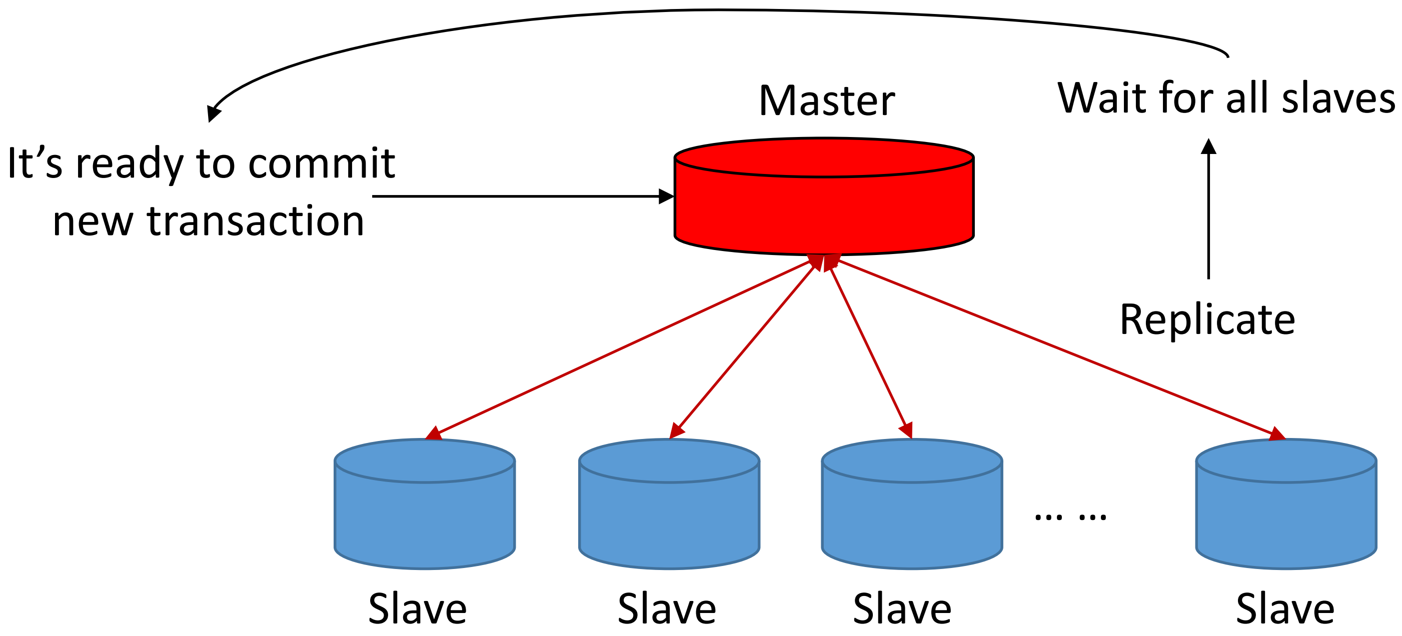
Master-Slave replication

- Master-Slave
  - A **master** server takes all the writes, updates, inserts
  - One or more **Slave** servers take all the reads (they can't write)
  - Only read **scalability**
  - The master is a single point of **failure**
- Some NoSQLs (e.g., CouchDB) support Master-Master replica



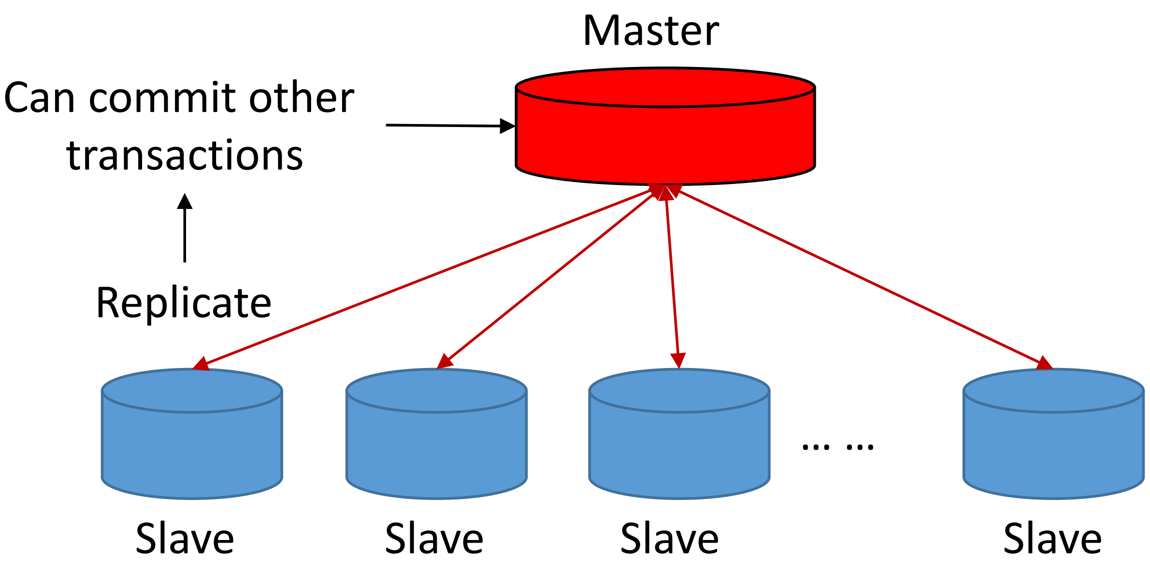
Synchronous replication

- Before committing a transaction, the Master **waits** for (all) the Slaves to commit
- Similar in concept to the **2-Phase Commit** in relational databases
- **Performance** killer, in particular for replication in the cloud
- Trade-off: wait for a subset of Slaves to commit, e.g., the **majority** of them



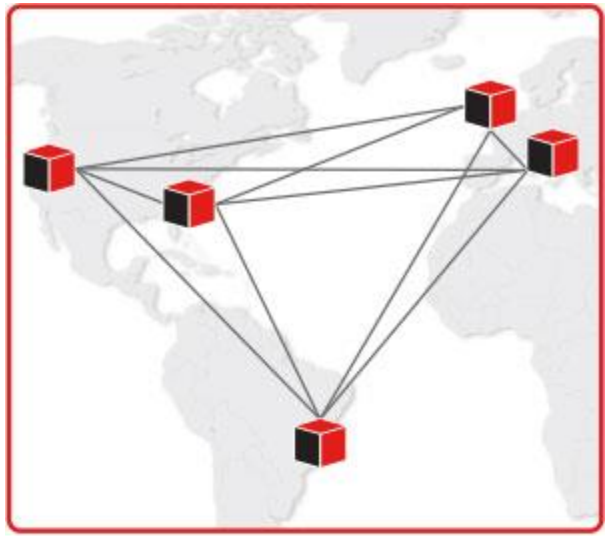
Asynchronous replication

- The Master commits **locally**, it does not wait for any Slave
- Each Slave independently fetches updates from Master, which may **fail**…
  - IF no Slave has replicated, then you've **lost the data** committed to the Master
  - IF some Slaves have replicated and some haven't, then you have to **reconcile**
- Faster and **un**reliable



Distributed databases

Different autonomous machines, working **together** to manage the same **dataset**

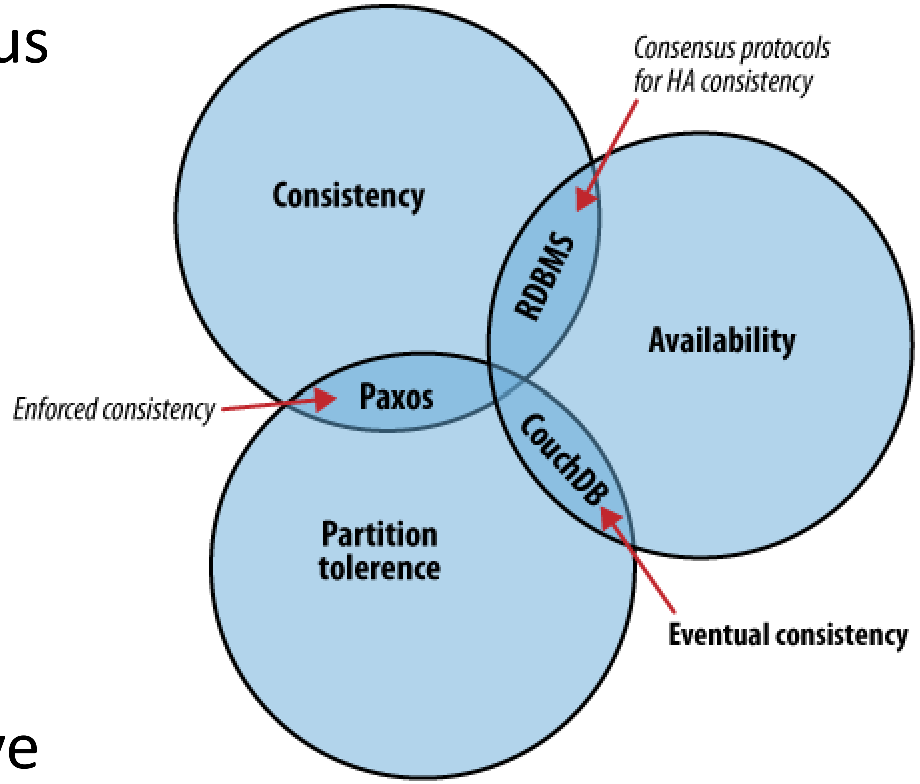


Key features of distributed databases

- There are 3 typical problems in distributed databases:
  - **Consistency**
    - All the distributed databases provide the same data to the application
  - **Availability**
    - Database failures (e.g., master node) do not prevent survivors from continuing to operate
  - **Partition** tolerance
    - The system continues to operate despite arbitrary message loss, when connectivity failures cause network partitions

CAP Theorem

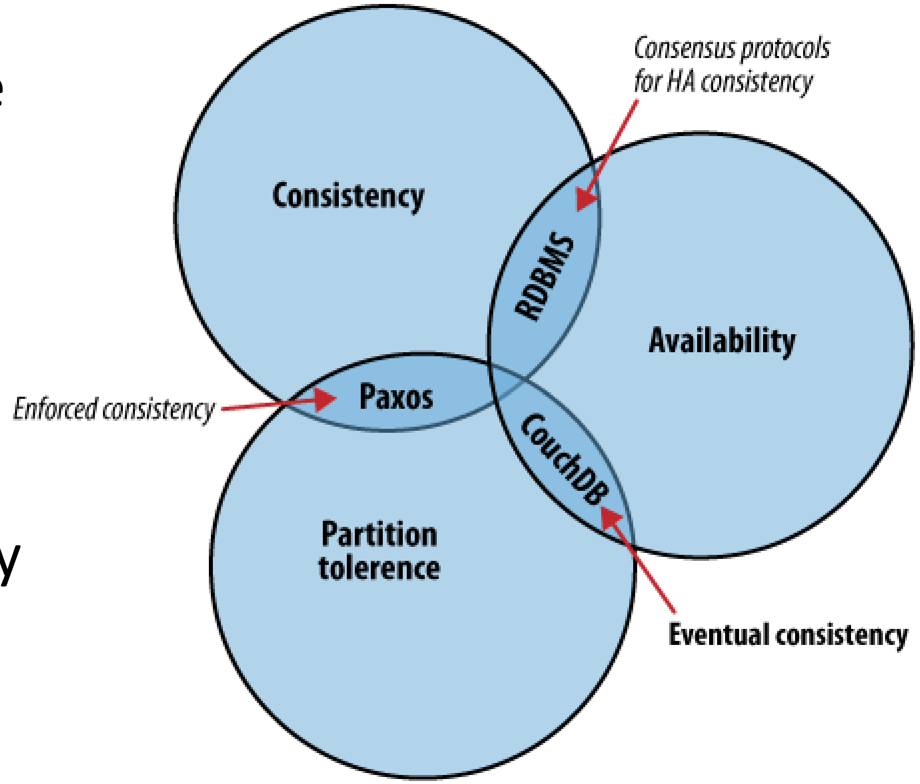
- The CAP theorem, also known as Brewer's theorem, states that it is **impossible** for a distributed system to **simultaneously** provide **all three** of the previous guarantees
- The theorem began as a **conjecture** made by University of California in 1999-2000
  - Armando Fox and Eric Brewer, “Harvest, Yield and Scalable Tolerant Systems”, Proc. 7th Workshop Hot Topics in Operating Systems (HotOS 99), IEEE CS, 1999, pg. 174-178.
- In 2002 a formal proof was published, establishing it as a **theorem**
  - Seth Gilbert and Nancy Lynch, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”, ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59
- In 2012, a follow-up by Eric Brewer, “CAP twelve years later: How the "rules" have changed”
  - IEEE Explore, Volume 45, Issue 2 (2012), pg. 23-29.



http://guide.couchdb.org/editions/1/en/consistency.html#figure/1

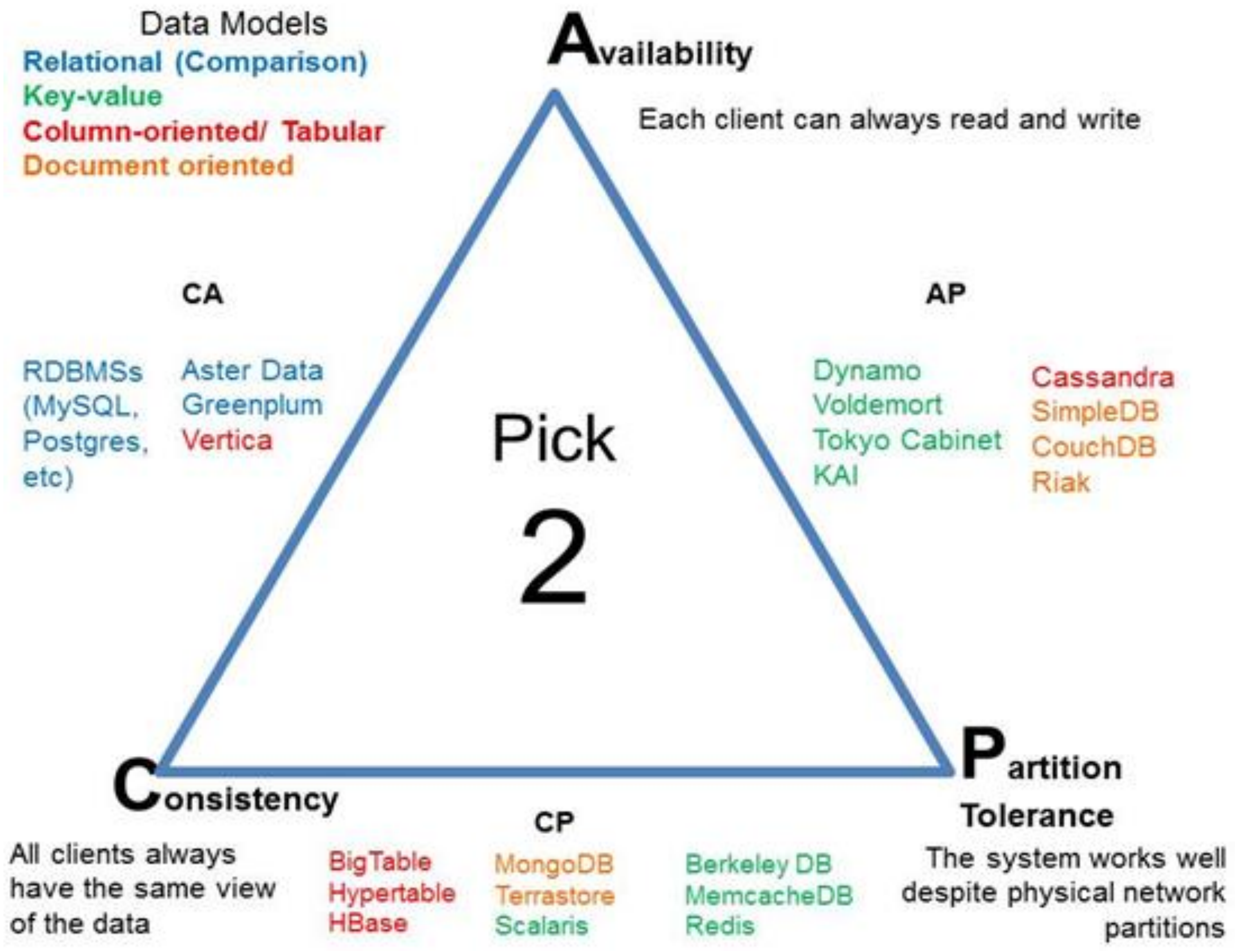
CAP Theorem

- The easiest way to understand CAP is to think of **two nodes** on opposite sides of a **partition**.
- Allowing at least one node to update state will cause the nodes to become **inconsistent**, thus forfeiting C.
- If the choice is to preserve consistency, one side of the partition must act as if it is **unavailable**, thus forfeiting A.
- Only when no network **partition** exists, is it possible to preserve both consistency and availability, thereby forfeiting P.
- The general belief is that for wide-area systems, **designers cannot forfeit P** and therefore have a difficult choice between C and A.



http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

CAP Theorem



http://blog.flux7.com/blogs/nosql/cap-theorem-why-does-it-matter

CA without P (local consistency)

- **Partitioning** (communication breakdown) causes a failure.
- We can still have **Consistency** and **Availability** of the data shared by agents **within each Partition**, by ignoring other partitions.
  - Local rather than global consistency / availability
- Local consistency for a partial system, 100% availability for the partial system, and no partitioning does not exclude several partitions from existing with their own "internal" CA.
- So partitioning means having **multiple independent systems** with 100% CA that do not need to interact.

CP without A (transaction locking)

- A system is allowed to *not* answer requests at all (turn off "A").
- We claim to tolerate **partitioning/faults**, because we simply block all responses if a partition occurs, assuming that we cannot continue to function correctly without the data on the other side of a partition.
- Once the partition is healed and **consistency** can once again be verified, we can restore availability and leave this mode.
- In this configuration there are global consistency, and global correct behaviour in partitioning is to **block access to replica sets** that are not in synch.
- In order to tolerate P at any time, we must sacrifice A at any time for **global consistency**.
- This is basically the **transaction lock**.

AP without C (best effort)

- If we don't care about **global consistency** (i.e. simultaneity), then every part of the system can make available what it knows.
- Each part might be able to answer someone, even though the system as a whole has been broken up into incommunicable regions (**partitions**).
- In this configuration “without consistency” means without the assurance of **global** consistency **at all times**.

A consequence of CAP

"Each node in a system should be able to make decisions purely based on **local state**. If you need to do something under high load with **failures** occurring and you need to reach agreement, you're lost. If you're concerned about **scalability**, any algorithm that forces you to run agreement will eventually become your **bottleneck**. Take that as a given."

Werner Vogels, Amazon CTO and Vice President

Beyond CAP

- The "2 of 3" view is misleading on several fronts.
- First, because **partitions** are rare, there is little reason to forfeit C or A when the system is not partitioned.
- Second, the **choice between C and A** can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved.
- Finally, all three **properties are more continuous than binary**.
    - Availability is obviously continuous from 0 to 100 percent
    - There are also many levels of consistency
    - Even partitions have nuances, including disagreement within the system about whether a partition exists

How the rules have changed

- Any networked shared-data system can have **only 2 of 3** desirable properties at the **same time**
- Explicitly handling partitions, designers can optimize consistency and availability, thereby achieving some **trade-off of all three**
- CAP prohibits only a tiny part of the design space:
  - **perfect** availability (A) and consistency (C)
  - in the presence of partitions (P), which are **rare**
- Although designers need to choose between consistency and availability when partitions are present, there is an incredible range of **flexibility for handling partitions** and recovering from them
- Modern CAP goal should be to maximize combinations of consistency (C) and availability (A) that make sense for the **specific application**

ACID

- The four ACID properties are:
  - **Atomicity (A)** All systems benefit from atomic operations, the database transaction must completely succeed or fail, partial success is not allowed
  - **Consistency (C)** During the database transaction, the database progresses from a valid state to another. In ACID, the C means that a transaction pre-serves all the database rules, such as unique keys. In contrast, the C in CAP refers only to single copy consistency.
  - **Isolation (I)** Isolation is at the core of the CAP theorem: if the system requires ACID isolation, it can operate on at most one side during a partition, because a client's transaction must be isolated from other client's transaction
  - **Durability (D)** The results of applying a transaction are permanent, it must persist after the transaction completes, even in the presence of failures.

BASE

- **Basically Available**: the system provides availability, in terms of the CAP theorem
- **Soft state:** indicates that the state of the system may change over time, even without input, because of the eventual consistency model.
- **Eventual consistency:** indicates that the system will become consistent over time, given that the system doesn't receive input during that time
- Example: DNS – Domain Name Servers
  - DNS is not multi-master

ACID versus BASE

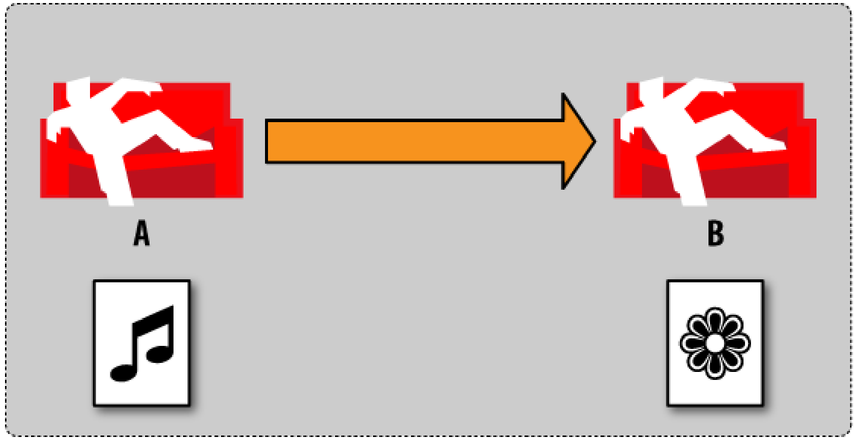
- ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum
- ACID properties focus on **consistency** and are the traditional approach of databases
- BASE properties focus on high **availability** and to make explicit both the choice and the spectrum
- **BASE**: Basically Available, Soft state, Eventually consistent, work well in the presence of **partitions** and thus promote **availability**

Conflict detection and resolution

An example from a notable NoSQL database

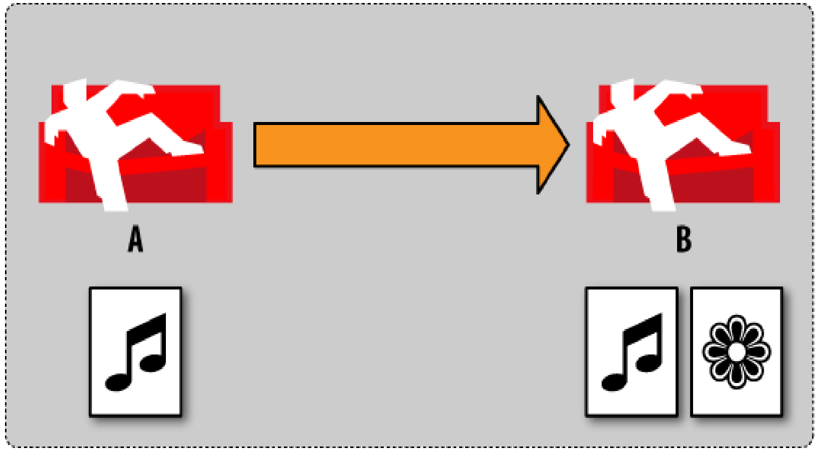
Conflict resolution problem

- There are two customers, **A** and **B**
- **A** books a hotel room, the last available room
- **B** does the same, on a different node of the system, which was **not consistent**



Conflict resolution problem

- The hotel room document is affected by two **conflicting updates**
- Applications should solve the conflict with custom logic (it’s a business decision)
- The database can
  - **Detect** the conflict
  - Provide a local **solution**, e.g., latest version is saved as the winning version



Conflict

- CouchDB guarantees that **each instance** that sees the **same conflict** comes up with the **same winning** and losing **revisions**.
- It does so by running a **deterministic algorithm** to pick the winner.
  - The revision with the longest revision history list becomes the winning revision.
  - If they are the same, the **_rev** values are compared in ASCII sort order, and the highest wins.