ACID, CAP, ~~No~~ NewSQL

A 30k  overview of distributed databases

Vlad Petric

Single-system databases

Most are relational databases

A well-understood problem

- Standardized & formalized in the 90s
- Great implementations in the 2000s, including Free Software (MySQL, Postgres, etc.)
- Work really well, up to some limits

ACID + SQL

ACID

A - atomicity

C - consistency

I - isolation

D - durability

SQL

- Decent query language, based on relational algebra

- Allows non-programmers to write complex queries
  - Filtering
  - Joins
  - Aggregates

- Reasonably well-behaved - e.g., guaranteed polynomial time

Limitations of ACID SQL databases

Hard to scale beyond several systems

- Neither ACID, nor SQL scale well

However, single computer systems scaled well between 1990 and present

- Faster processors (cores), more cores
- Faster memory, higher capacity memory
- Faster storage, higher capacity storage (spinning drives, then solid state)

Then came Big Data …

We need a database to store an entire web crawl …

We need a database to for all our users' (>100M) clicks …

Ad-hoc scaling of single-system databases

First solution - ad-hoc scaling

- Partition large tables based on keys
- Distribute on multiple servers
- All customers starting with “A” go to server 1, “B” -> 2,
    - In practice: use a good hash function

Ad-hoc scaling

One node - probability of going down of 1 in 1000 (.1%)

System of a 100 nodes - what is the probability of at least one node being down?

Even if nothing is down (yeah, right):

- ACID on a node doesn't mean ACID on whole system
- Querying is much more difficult
- Network can still fail

Remainder of talk

Introduction

Distributed databases

- Definition
- Wish list
- Basic Distributed System
- CAP theorem

Real distributed database systems

Conclusions

Distributed databases

System: something with a clearly defined boundary

Distributed systems:

- Collection of computers
- Communicate and *coordinate* via network messages

Distributed database system:

- Distributed system that is a database (write/read/query)
- ***Same data may be accessed from multiple nodes***

What would we like from a distributed database?

Everything from single-system database

- ACID, SQL

Scalability

- Quantity of data
- Read/Write/Complex query bandwidth should scale with the number of systems

Fault Tolerance

- System should hide node failures, network failures

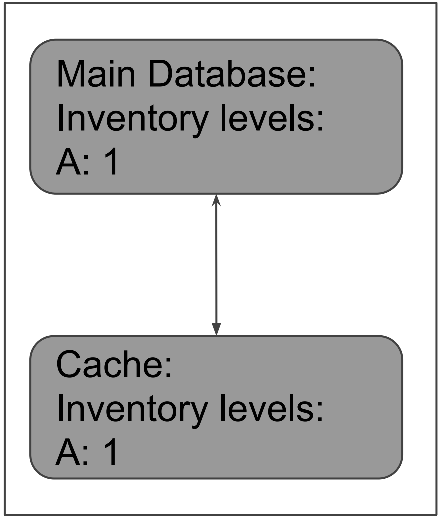
Basic distributed database: inventory system

Inventory system: Main Database + Cache

Distributed database; e.g. A - inventory of book “1984”

Can read from either Main DB or Cache

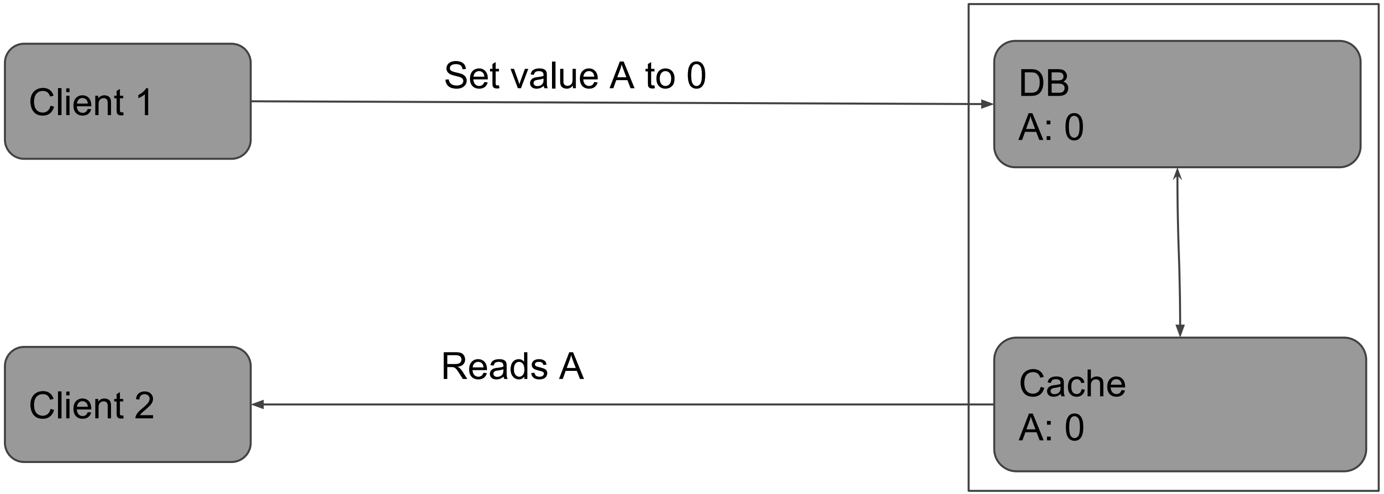
Can only write to main DB



Basic distributed database: inventory system (2)

Good situation:

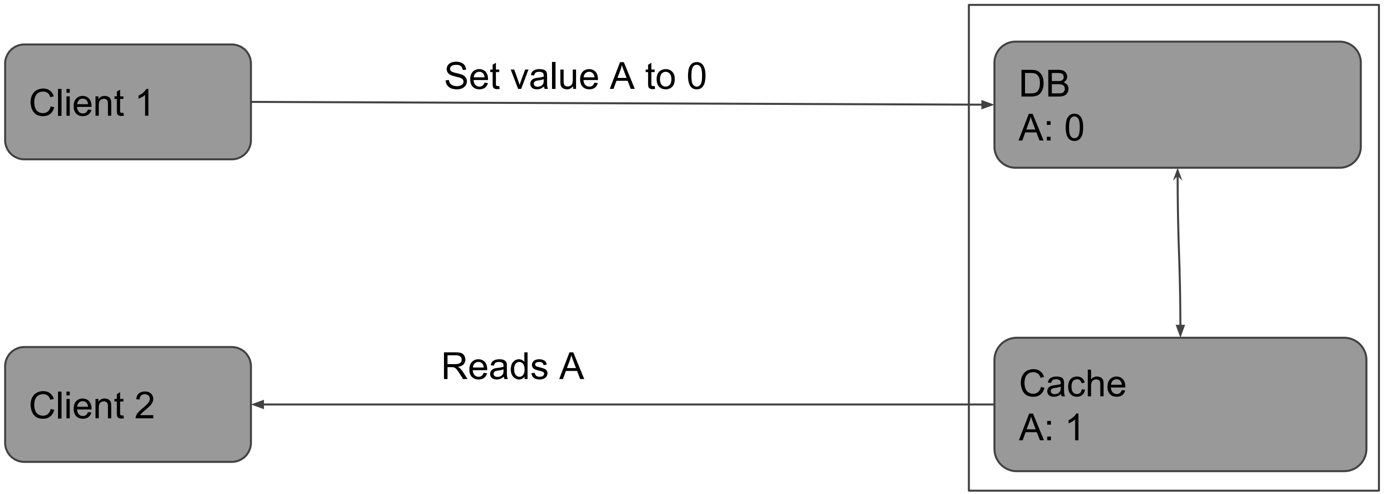
- Client 1 writes A:0
- Writer changes A:0, and propagates value to the RA
- Client 2 reads A:0



Basic distributed database: inventory system (3)

Inconsistent value:

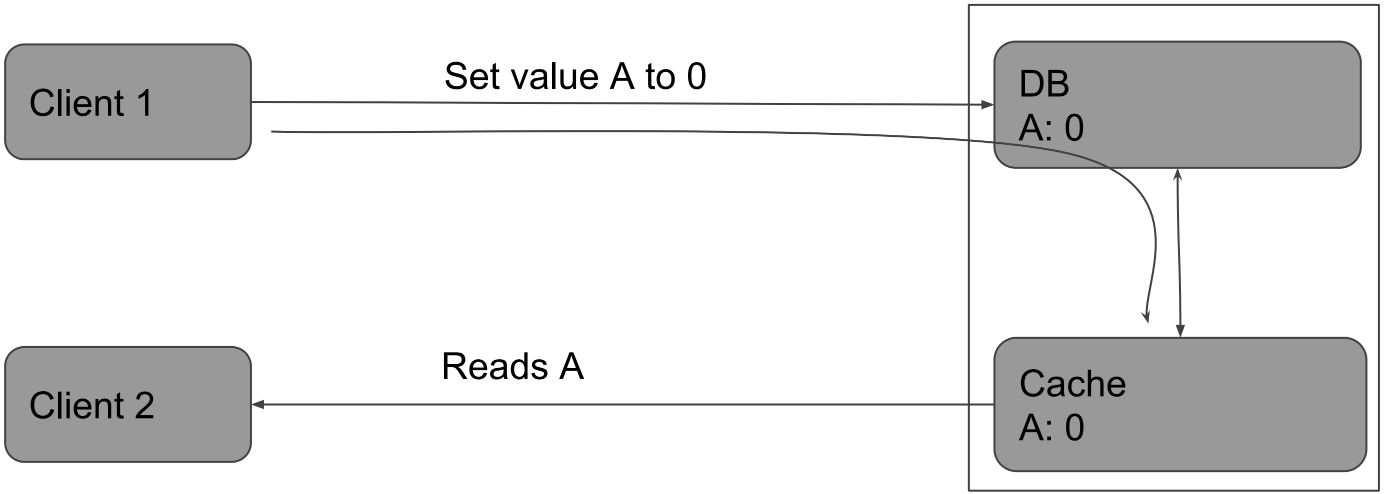
- Client 1 writes A:0
- Client 2 immediately reads A: 1



Basic distributed database: inventory system (4)

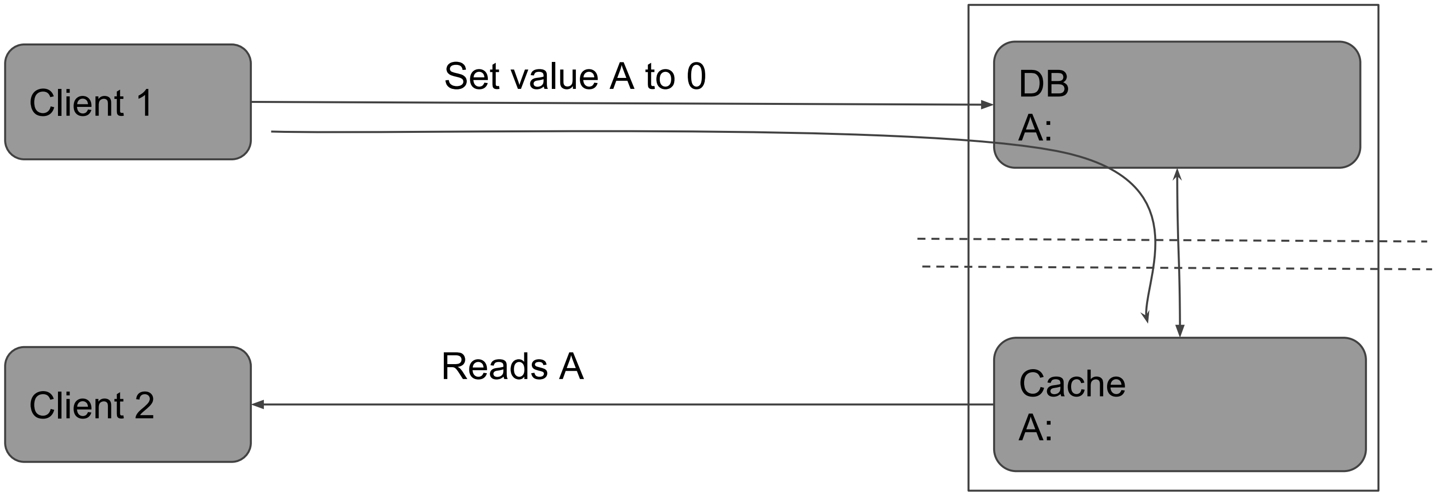
Getting consistency:

- Don’t finish the write until the cache also has the most up to date value



Basic distributed database: inventory system (5)

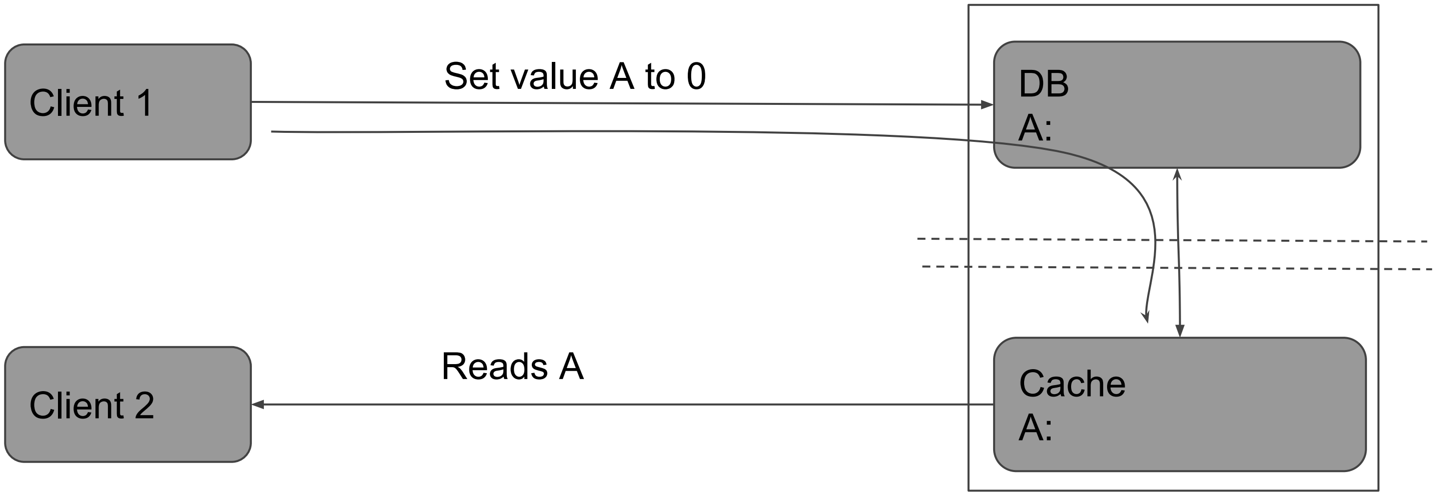
But what if the connection between DB & Cache is severed?



Basic distributed database: inventory system (6)

But what if the connection between DB & cache is severed?

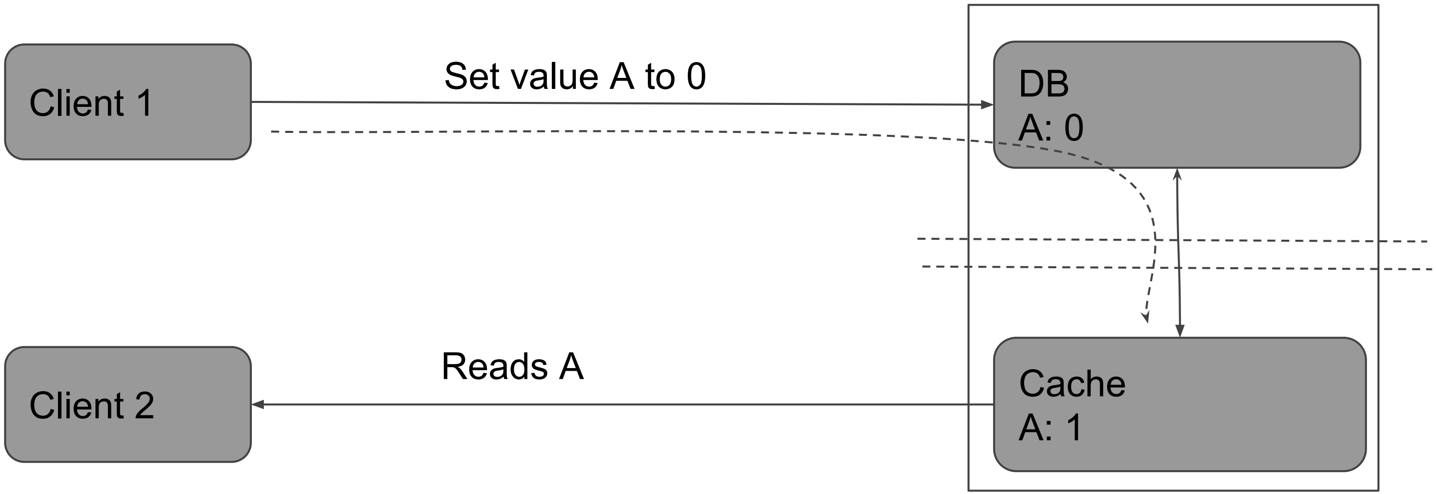
- We could block the write, and eventually fail it (timeout)



Basic distributed database: inventory system (7)

But what if the connection between DB & cache is severed?

- Not propagate the value - inconsistent!



Meet the CAP theorem

CAP (Eric Brewer):

- Consistency
- Availability
- Partition Tolerance

Meet the CAP theorem

CAP:

- Consistency: reads receive the most recent value
  - Once you write something, everyone reads it

- CAP Consistency vs ACID C/I/D (Definitions matter!)

Meet the CAP theorem

CAP:

- Availability: every request receives a non-error response
  - Writes are always accepted
  - Reads see a value (doesn’t necessarily have to be the latest)

Meet the CAP theorem

CAP:

- Partition tolerance - system tolerates an arbitrary number of nodes disconnected from the rest of the system (nodes can’t talk to each other)

CAP Theorem - at most 2 out of 3

```
Consistent and Available => not Partition tolerant

Consistent and Partition tolerant => not Available

Available and Partition tolerant => not Consistent
```

CAP Theorem - in practice

In a distributed system

- Network issues
  - Misconfigurations
  - Power, cooling issues
- JVM stop-the-world garbage collection

P happens!

When P happens - Availability or Consistency?

How do we build useful systems?

---

1. Throw our hands up in the air
    a. “If you want availability, completely give up on consistency and viceversa”
    b. Or pretend that P doesn’t happen

2. Weaken the requirements slightly (change the definitions)

    a. Instead of full availability, high availability
        i. E.g., 99.99999…% requests handled
    b. Weaker consistency models, but stronger than no consistency
    c. CAP theorem only works for strict definitions!

Weakening availability

Networks may fail, but with a complex, redundant topology it’s *far less likely* that they result in failed transactions.

- Geo-redundancy
- No single point of failure

Weakening consistency

Strict serializability (1) - equivalent to C in CAP

Every write is seen immediately in the system.

Implies a *total ordering of operations, reads and writes*, based on time.

Consistency Levels

Serializable (2)

There is a total ordering of operations

- Execution corresponds to some serial order of operations
- … but not completely based on time.

Consistency Levels

Serializable (2)

The following writes happen in order: X, Y, Z

- Y happens after X, Z happens after Y

What will readers see? One of the following:

- Nothing
- X
- X, Y
- X, Y, Z

Consistency Levels

Eventual consistency (3)

If you stop writing, replicas converge

May not seem like much, but still offers a degree of consistency

- Replicas converge within some amount of time
- Not guaranteed, but measurable

Consistency Recap

Strict Serializable

Serializable

Eventually Consistent

Important reminder - this is a 30k feet view!

- Many variants and intermediate levels.

Remainder of talk

Introduction

Distributed databases

Real distributed database systems

- Existing Databases

Conclusions

Strictly serializable systems

Apache Zookeeper, Google Chubby

- Distributed lock service, master election
- Look like a filesystem (hierarchical namespace)
- Can store small pieces of data as well (KiB, not GiB)
- Not suitable to high-throughput

Strict serializable systems

Zookeeper, Google Chubby

- Use N replicas
  - Every write goes to at least round up(N/2 + 1) replicas
  - Generally, odd number of replicas

- Consensus algorithm
  - Replicas agree to *the order of all writes*

- Reads:
  - For strict serializable, read from round up(N/2 + 1)
  - For serializable, read from a single replica

Strict serializable systems - partition example

---

5 replicas

P1: A, B, C split from
P2: D, E

Write to P1, P2?

Read from P1, P2?

Strict Serializable / Serializable systems

Google Spanner, CochroachDB, VoltDB

- Strict serializable as long as clocks synchronized
  - Tens of milliseconds of drift
- Serializable otherwise
- *Wait out the drift*
  - Spanner: wait on write side
  - CockroachDB: wait on read side

Strict Serializable / Serializable systems

3rd party testing was critical

- Jepsen.io found serialization bugs in both CochroachDB and VoltDB, subsequently fixed

CockroachDB, VoltDB - SQL subset

- Including joins!
- NoSQL became NewSQL

Eventual consistency systems

E.g., Cassandra, Big Table, Aerospike

- Any replica may accept writes.
- In case of conflict, timestamp determines who wins.
- Ordering only happens on conflict resolution

Why use eventual consistency systems?

High Throughput, Low Latency

- Easily an order of magnitude better than (strict) sequential system

High Availability of Entire system

- Not the same as CAP availability (binary property)

But … you need to be able to deal with replication delay

Many things I didn't talk about (not an exhaustive list)

PACELC (Pass Elk) - CAP++

- Latency as a trade-off

What is this database suitable for?

- Size/structure of keys/data, Read/Write mix

How easy is it to manage?

- Cassandra - easily add a replica; Zookeeper - restart whole system

How easy is it to program?

Conclusions

- Definitions and Names matter
  - Can’t solve full problem? Come up with a slightly relaxed problem that is solvable

- Don’t trust the marketing department
  - https://jepsen.io/

- Choosing a distributed database means understanding trade-offs

About me

- Full name is Vlad Petric (not Vladimir), and I come from Transylvania (part of Romania). If you Google me, I’m not the bodybuilder
- Worked at Google on Web search and BigTable teams.
- Currently work in the financial sector
- I am the author of Akro build, a C++ build system with automated dependency tracking

Thank you!