Distributed transactions trade-offs

Artem Aliev

Huawei

Artem Aliev

- Huawei Cloud Hybrid Integration Platform
  - Expert and solution architect
- 20+ years in Software Development
  - Big data platforms integrations
  - Apache Hadoop, Spark, Cassandra, TinkerPop
  - Storage optimizations
  - JVM development
- SpbU teacher

artem.aliev@gmail.com

What is transaction

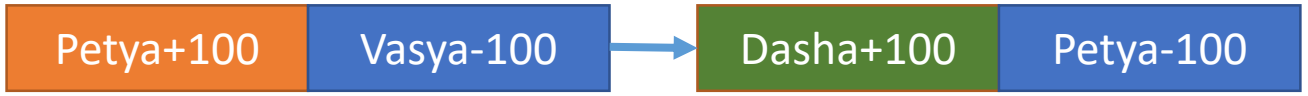
- Transfer many from Vasya to Petya
  - update amount=amount+100 from customers where name = ‘Petya’
  - update amount=amount-100 from customers where name = ‘Vasya’
- Bank state:
  - select sum(amount) as total from customers
- Natural requirement:
  - No money loose or stolen
  - Vasya or any customer can not have negative amount
  - Bank knows how much money it has: ‘total’ should be constant (consistent) if no external transfers
- Transaction can be canceled (rollback) in case of …

More formal: ACID properties

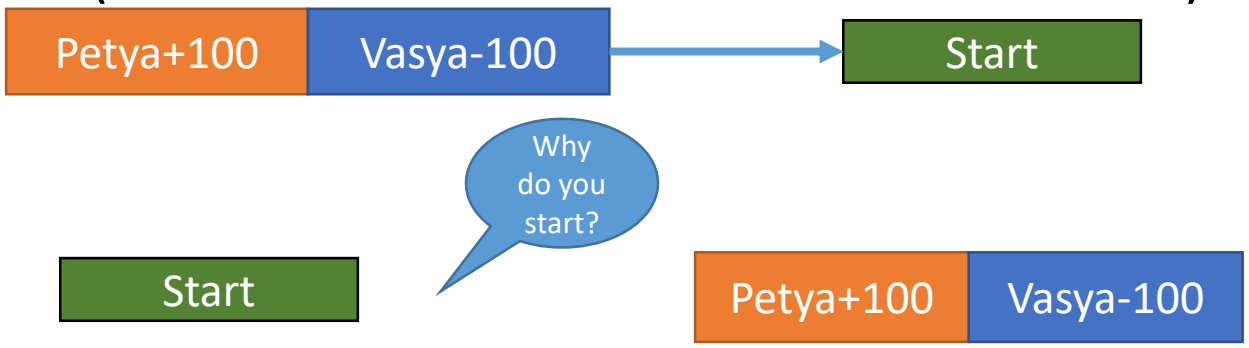
- Transaction is a single unit of logic or work, sometimes made up of multiple operations
- Atomicity
  - Both p=p+100 and v=v-100, either happened or not
  - If only first happened we get money from air
- Consistency
  - Restrictions like constant total, amount >=0, are met at the end of transaction
  - Transaction see only consistent state of the database in proper Isolation level
- Isolation
  - Read uncommitted (no isolation)
  - Read committed: You do not see partial results of other parallel transactions, till it committed
  - Repeatable read
  - Serializable (you can show order)
- Durability
  - If transaction is reported to be committed, it will not be loosed by “any” failure in the future

External Consistency

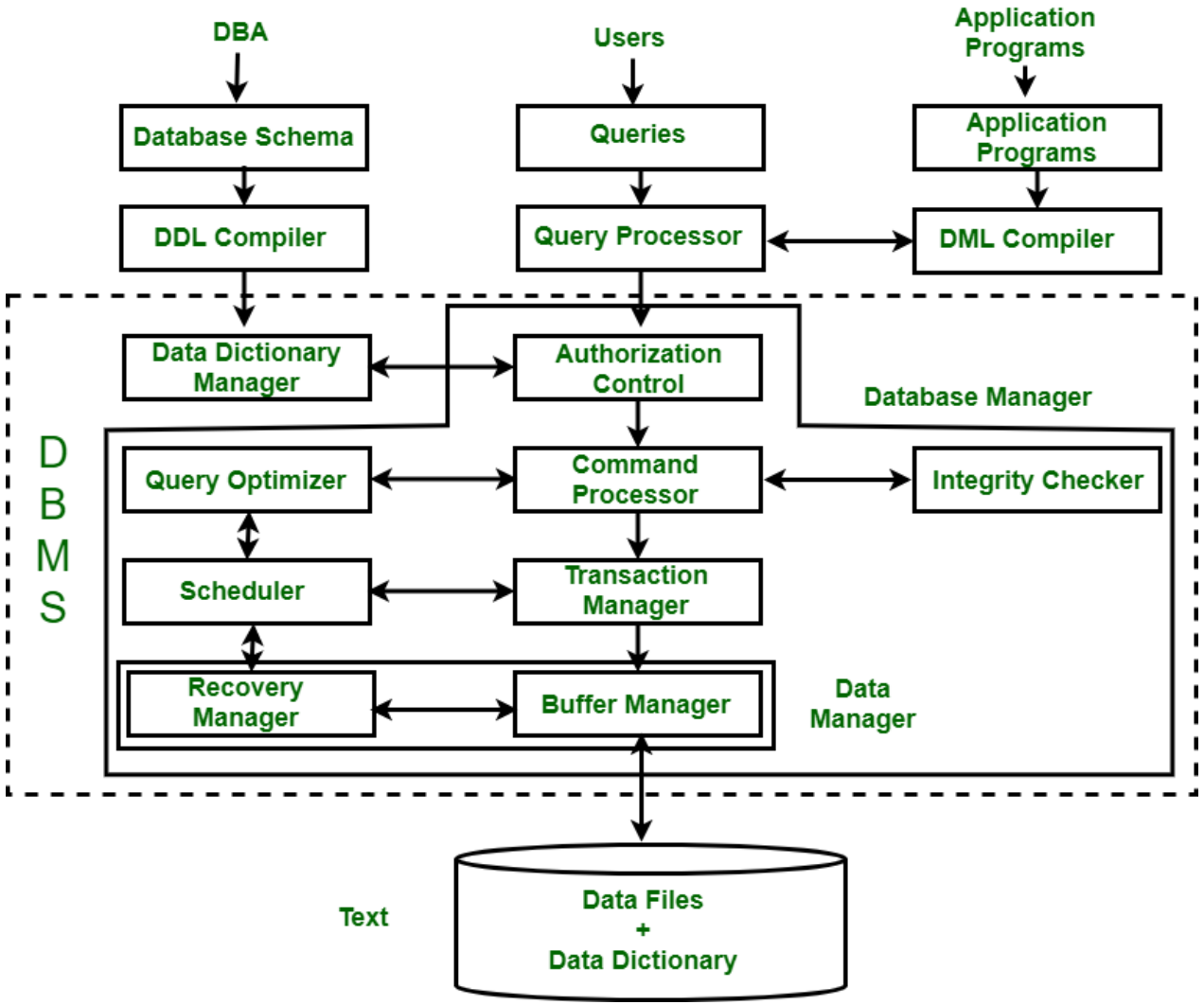
- Simple Consistency:



- Transaction is visible for all participants after commit
  - P: I have transferred you money. V: I do not see
  - (Asynchronous synchronization is not allowed)
- System can not reorder transactions
  - Money sent, start a production
  - (Hidden channel of clients communication)



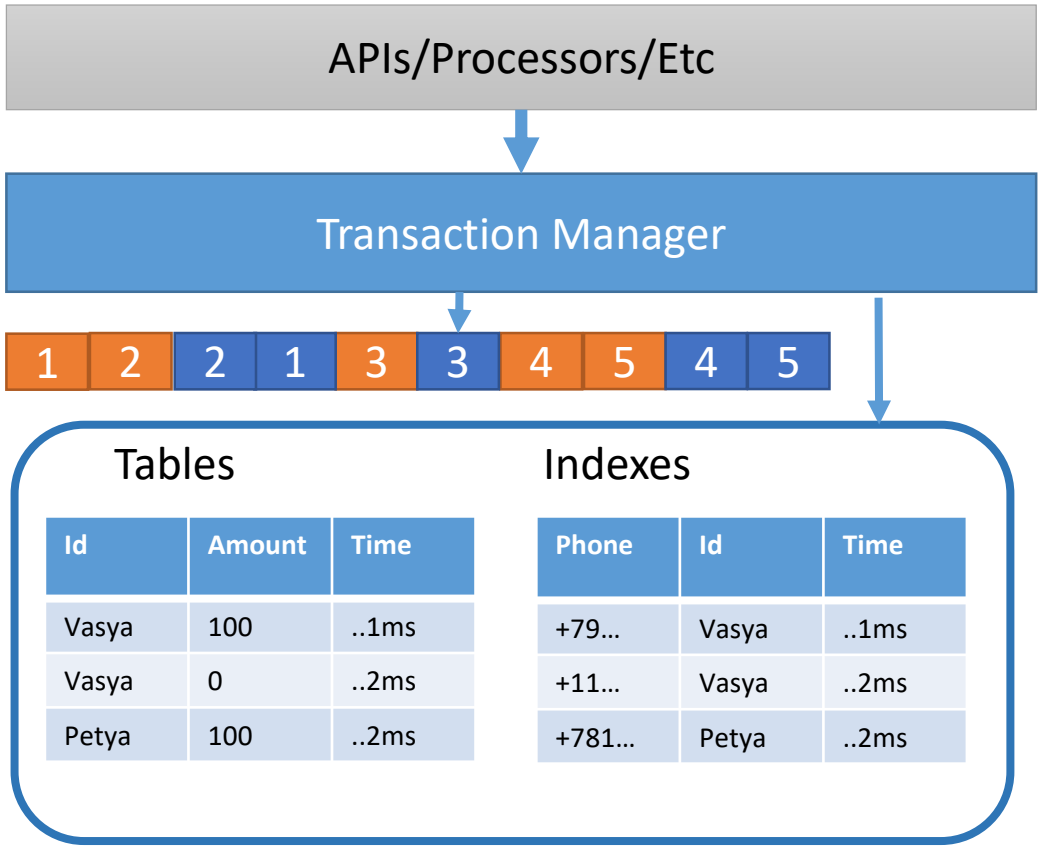
The last century architecture



The last century architecture (simplified)

Commit log

Multi Version Storage



Id	Amount	Time
Vasya	100	..1ms
Vasya	0	..2ms
Petya	100	..2ms

Phone	Id	Time
+79...	Vasya	..1ms
+11...	Vasya	..2ms
+781...	Petya	..2ms

- Parallel transactions
  - Topological order
- Single commit log!
  - Serializable transactions
  - External Consistency
- Multi Version tables
  - Parallel transactions
  - isolation

Transaction steps

- Start transaction (id, start time)
- Log start to Commit log
- Read Lock “P” row
- Read total
- Write Lock “P”
- Write new version of “P”
- Read Lock “V” row
- Read total
- Write Lock “V”
- Write new version of “V”
- Check constrains and fail or continue
- Commit transaction
  - Log finish in commit log
  - Release locks
  - New versions of data is visible

Continue

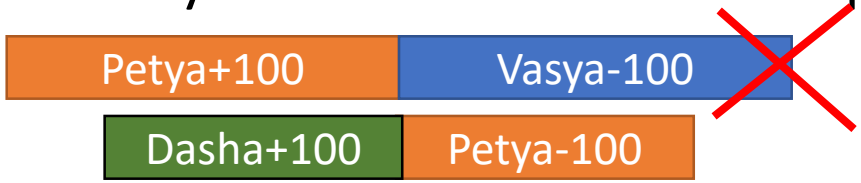
- independent transaction can run in parallel
- Seriliazability requirement is achieved by single Commit log (or timestamp)
- Deadlock is possible (P->V, V->P or any other cycle)
  - Different algorithms to solve them
  - Example: wait and force rollback younger transaction. It restarts

select sum(amount) from customers

- Read lock ALL rows -> stop ALL write transactions
  - Read, sum, release
- Read in the past
  - Fix timestamp
  - Wait till all transaction that starts before the timestamp finished
  - Read values with time <= the timestamp
  - Nor read lock for low isolation level reads

Let’s see how to hack the system

- If system does not support transactions we can stole money
  - Consistency + isolation should be supported together
  - T1:
  - T2

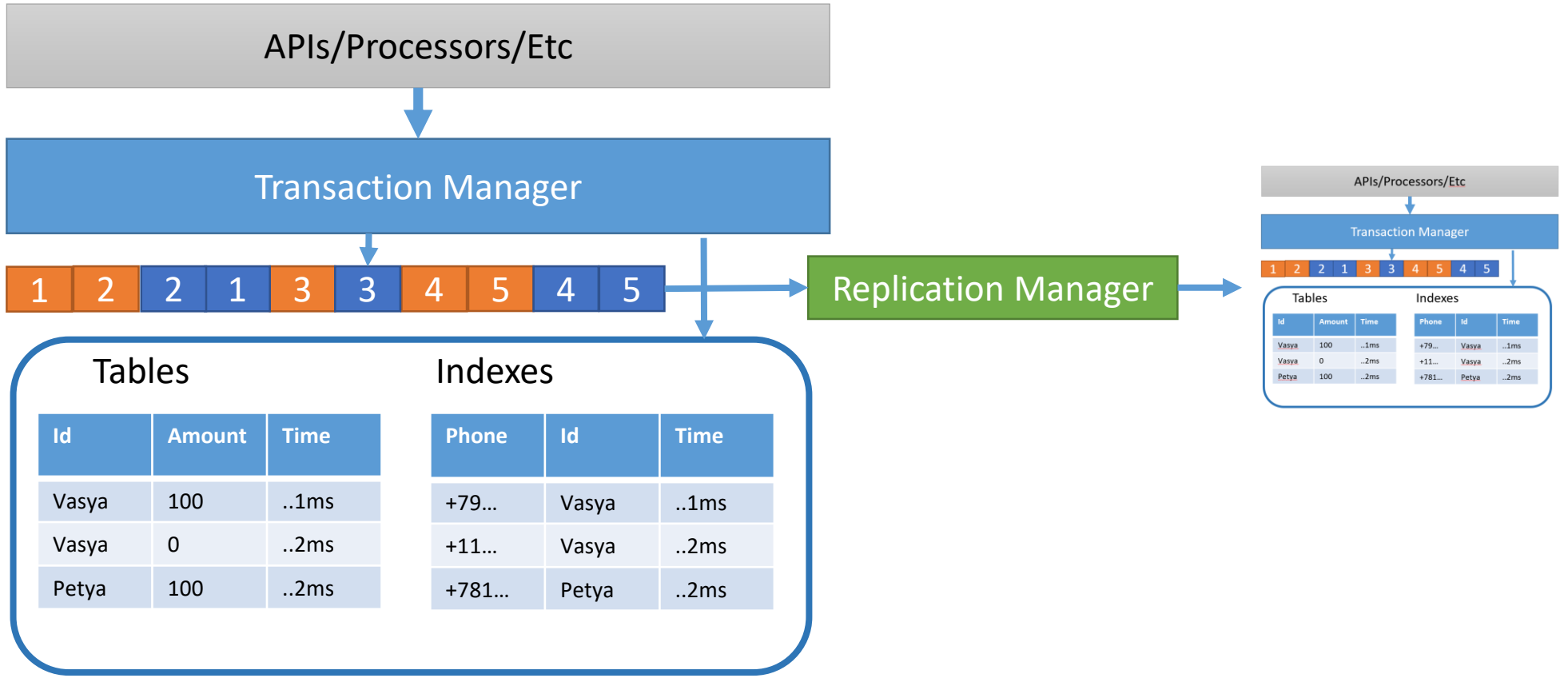


- If system has no replication: (D)DoS the system
  - High Availability or Fault Tolerance
- Durability
  - Report transaction is OK
  - Server dead => no clue about the transaction

Replication for High Availability

Commit log

Multi Version Storage



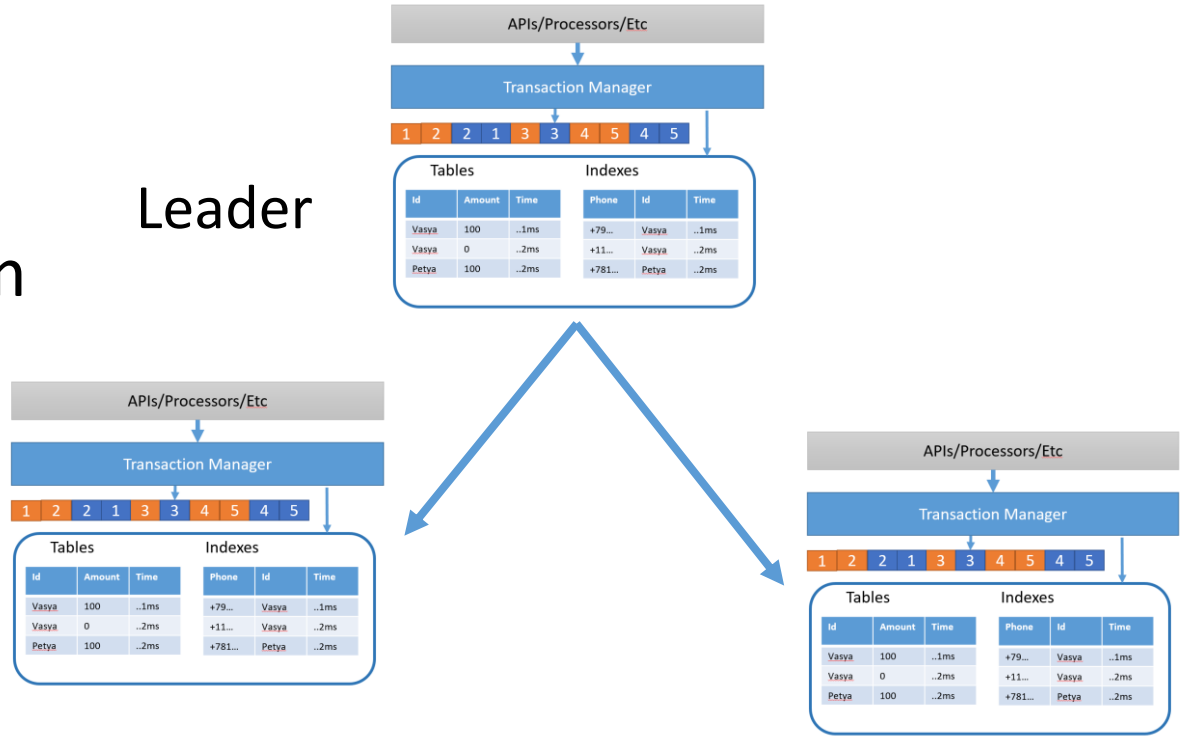
Replication for High Availability

- No single point of failure
- Time to switch or Time to Recover
- Master->slave replication
  - Split brain problem
  - Resolve conflicts
  - Still need external arbiter
- Asynchronous replication
  - Commit log conflicts at restore
  - Consistency is broken
  - Hack the system: send all money, kill the main server, send all money again
- Synchronous replication → Distributed transactions

Modern databases is more like….

Replica Set and Quorum

- Split the brain: Let’s vote for the leader!
- Write to leader
- Leader store to quorum
- Read from leader or from quorum



Commit log replication Leader

- Paxos/RAFT leader election and commit log replication
  - 100-300ms, up to 10 second time to Recover!

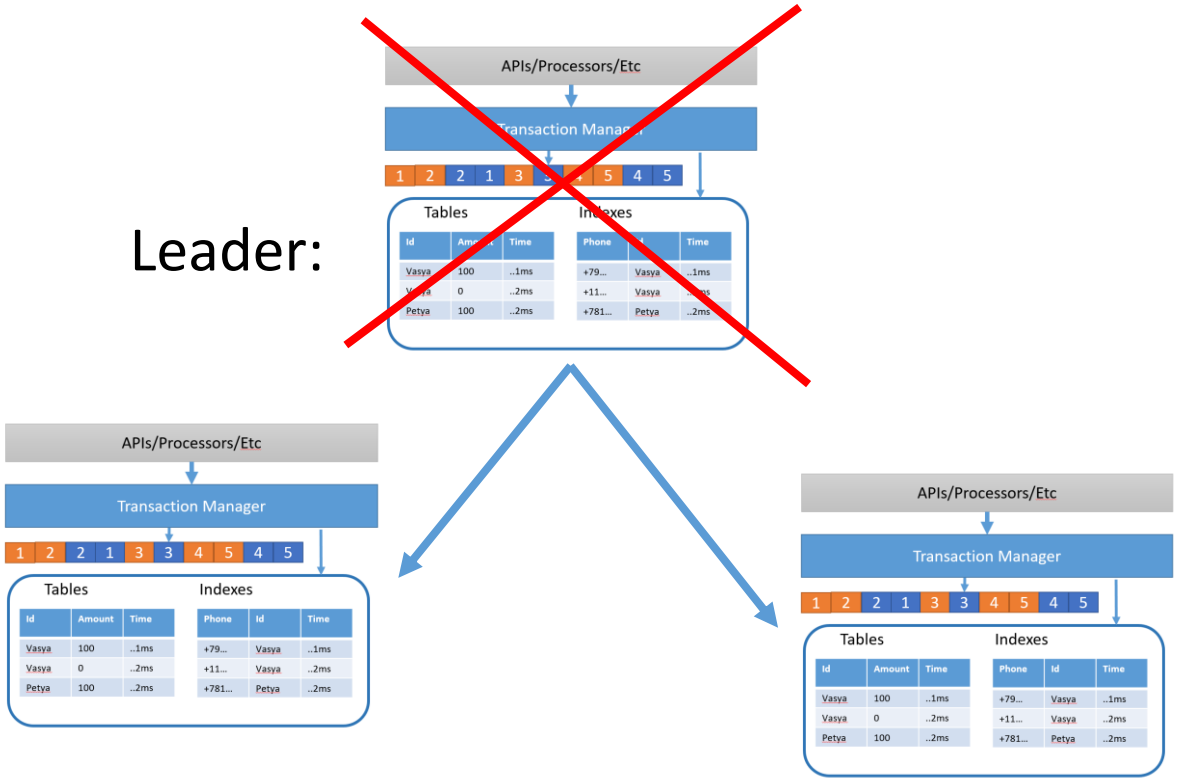
Google spanner

Handling region failure

If the preferred leader region fails or becomes unavailable, Cloud Spanner moves the leaders to another region. There could be a delay of a few seconds (less than 10 seconds) for Cloud Spanner to detect that the leader is no longer available and elect a new leader. During this time, applications could see higher read and write latencies.

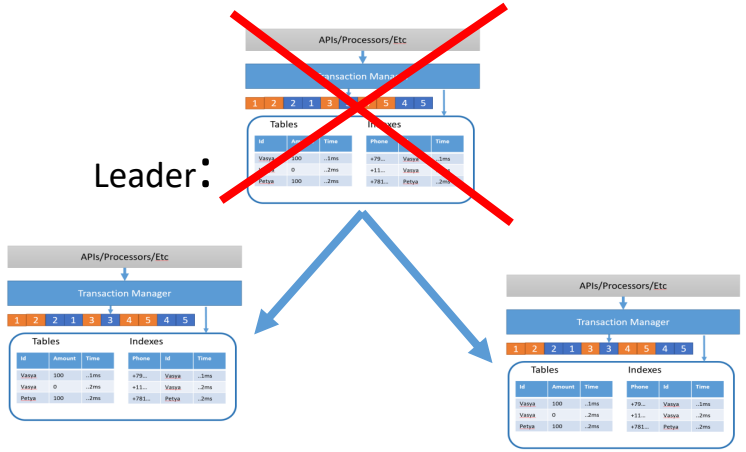
By the way

Leader:

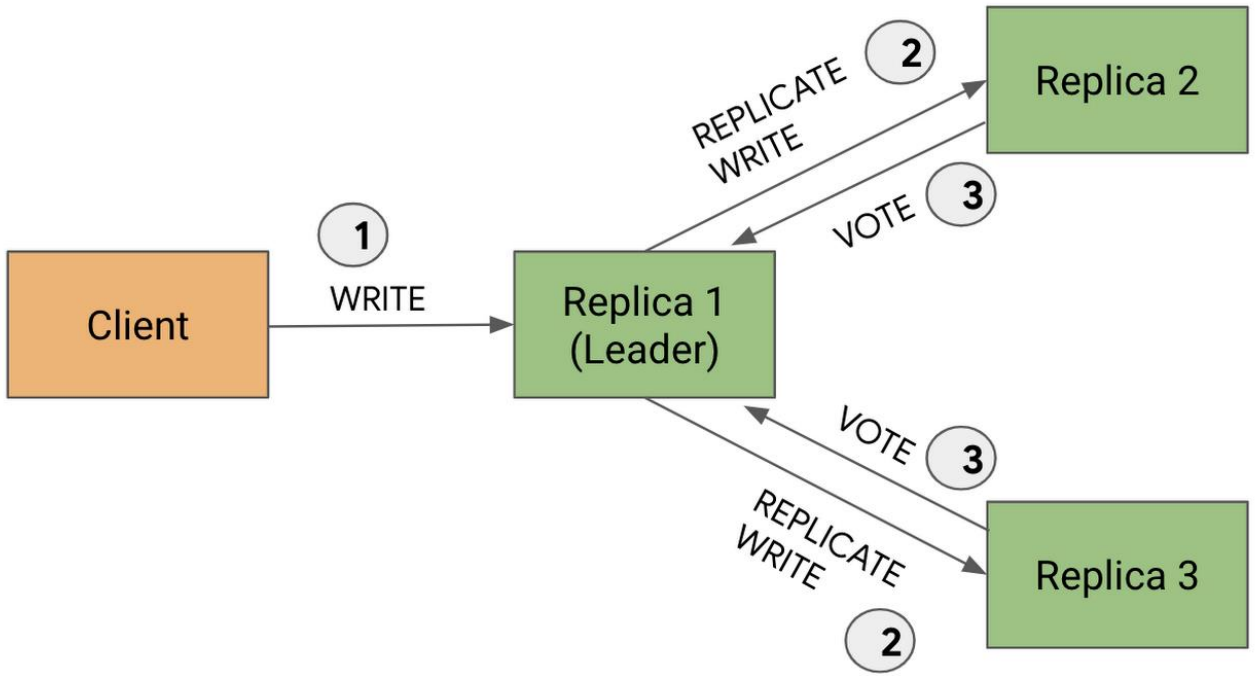


Time is important!

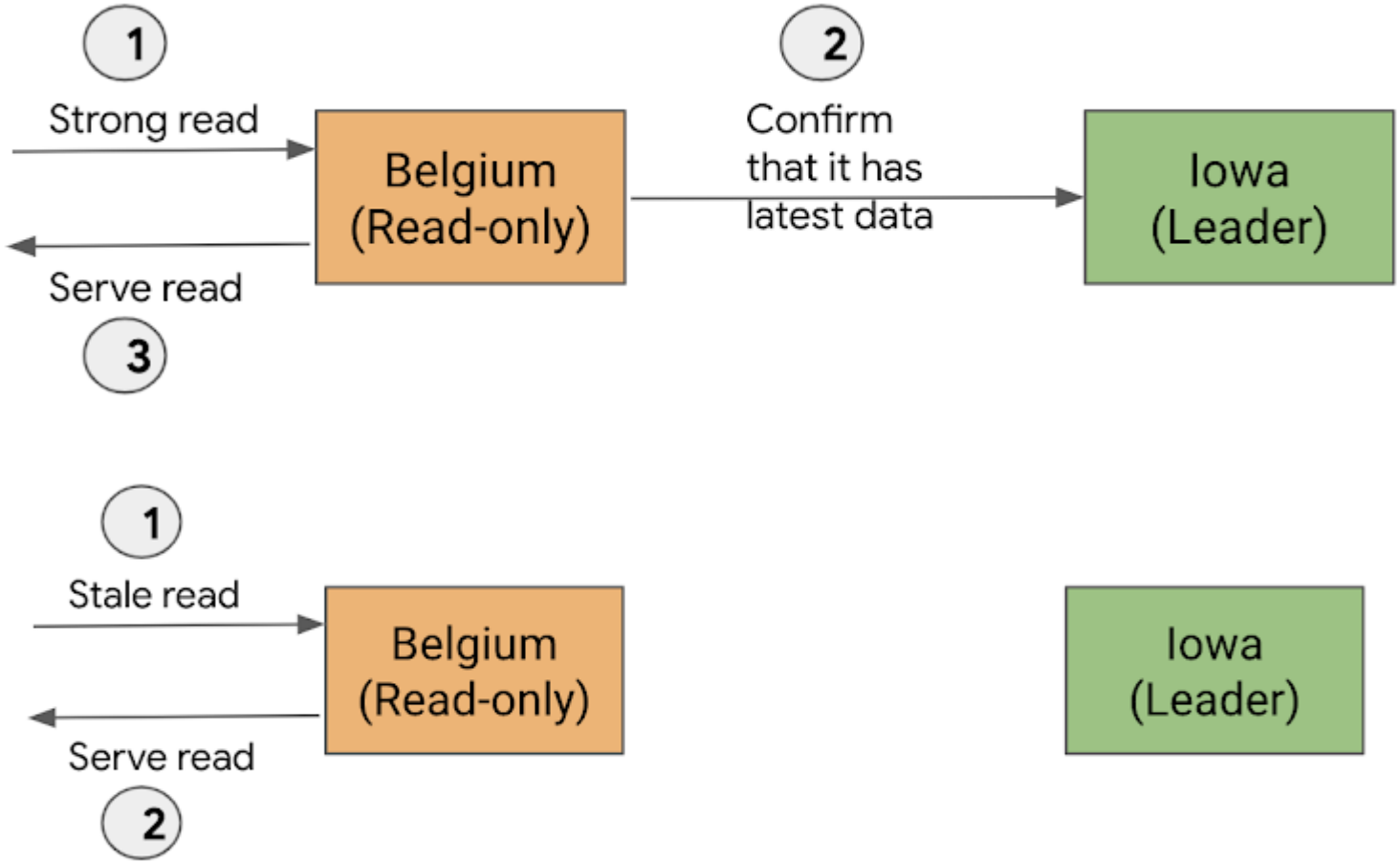
- Select leader candidate with latest changes
  - Quorum write guaranty we have at least one synchronized commit log in case of 3
- Quorum read select latest version of the value



Google Spanner write example



Spanner read Trade-off

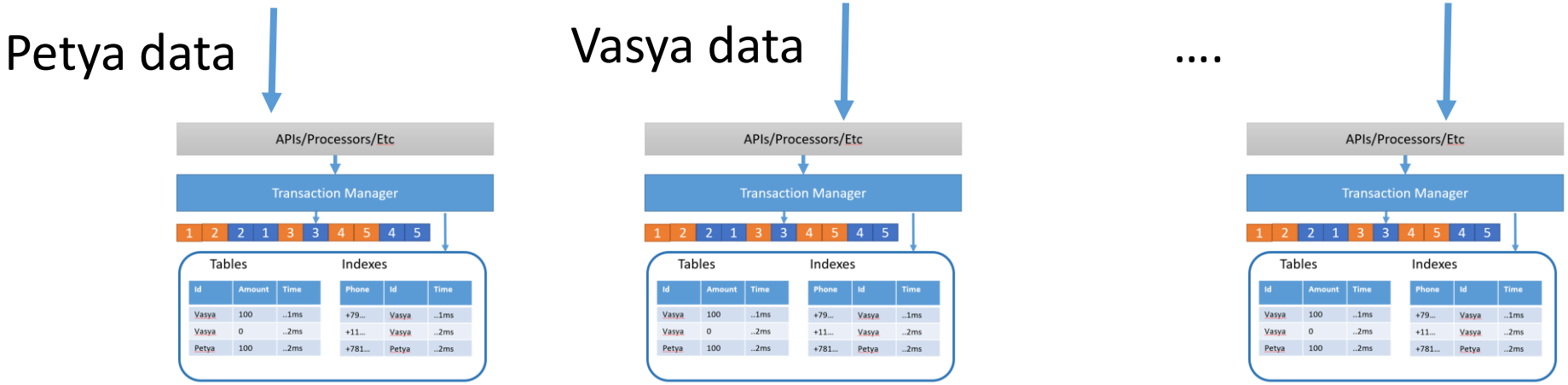


No scalability?

- Client can directly read from replica quorum and select latest result
  - Transactions? No.
  - Consistent “read in the past”. Yes
- Client directly write to all replicas and wait for quorum replies
  - No transactions
  - **Idempotent** updates only! Allows retry.
  - Example: vasya.total=vasya.total+100
    - Some nodes update value but did not response in time.
    - Retry?
    - 200, 100, 0 on different nodes ;)

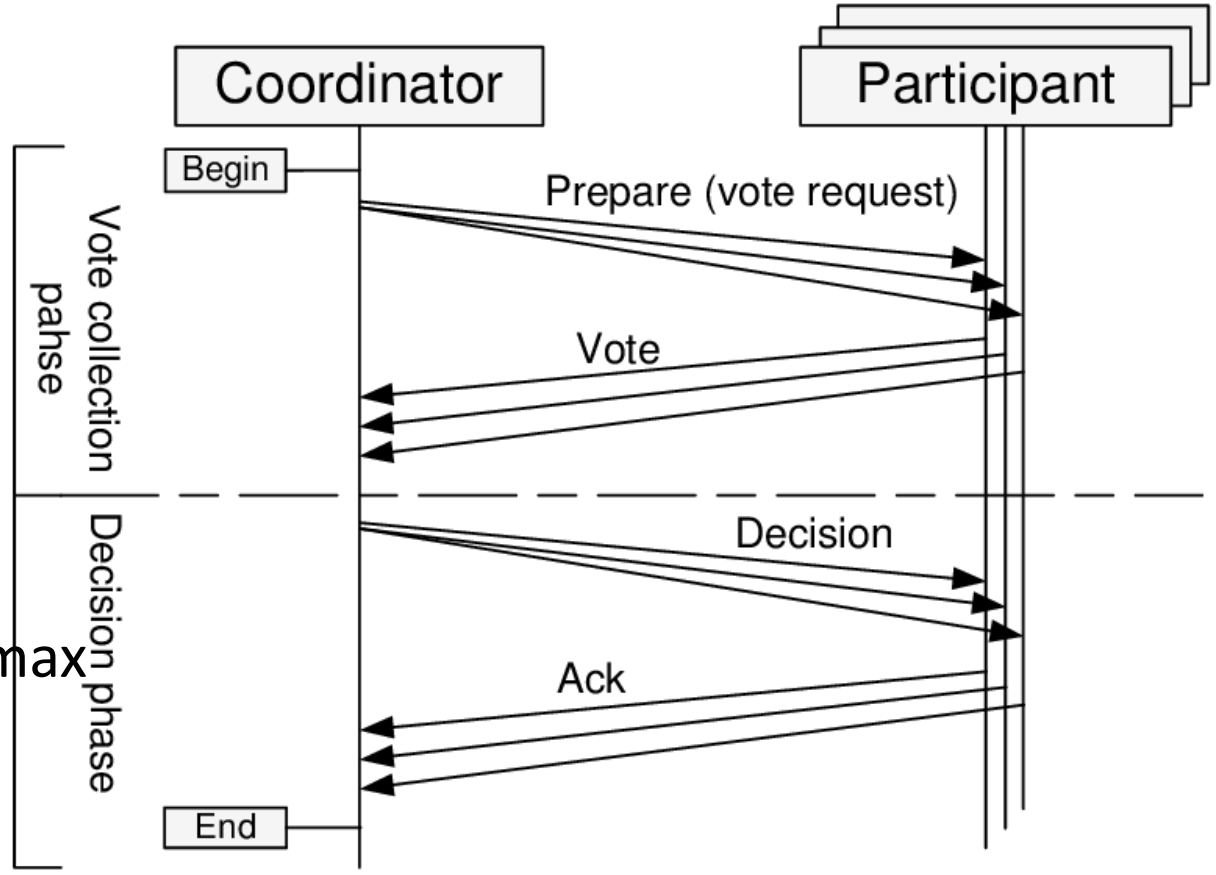
More scalability: Sharding/Partitioning

- For “linear” scalability
- The same as a set of independent databases
- Sharding rules can be applied on client or on transaction coordinator
- No scalability with single global transaction coordinator



2 phase commit (last century)

- Prepare stage
  - Lock data cells at first stage
  - Select max timestamp
- If ok, send commit message
  - With max timestamp
  - (lamport clocks)
- Participants
  - Increase local timestamp to max
  - Commit with the timestamp



2 phase transactions

- What if coordinator fail in the middle of decision phase?
  - Locks, time outs
  - Temporary inconsistent data (if no common clock)
- Coordinator should be in replica set!
- Scale coordinators
  - More then one commit log
  - How to get serializability?
- 2 phase transactions problems examples

The new SQL Server error log has an entry like the following example:

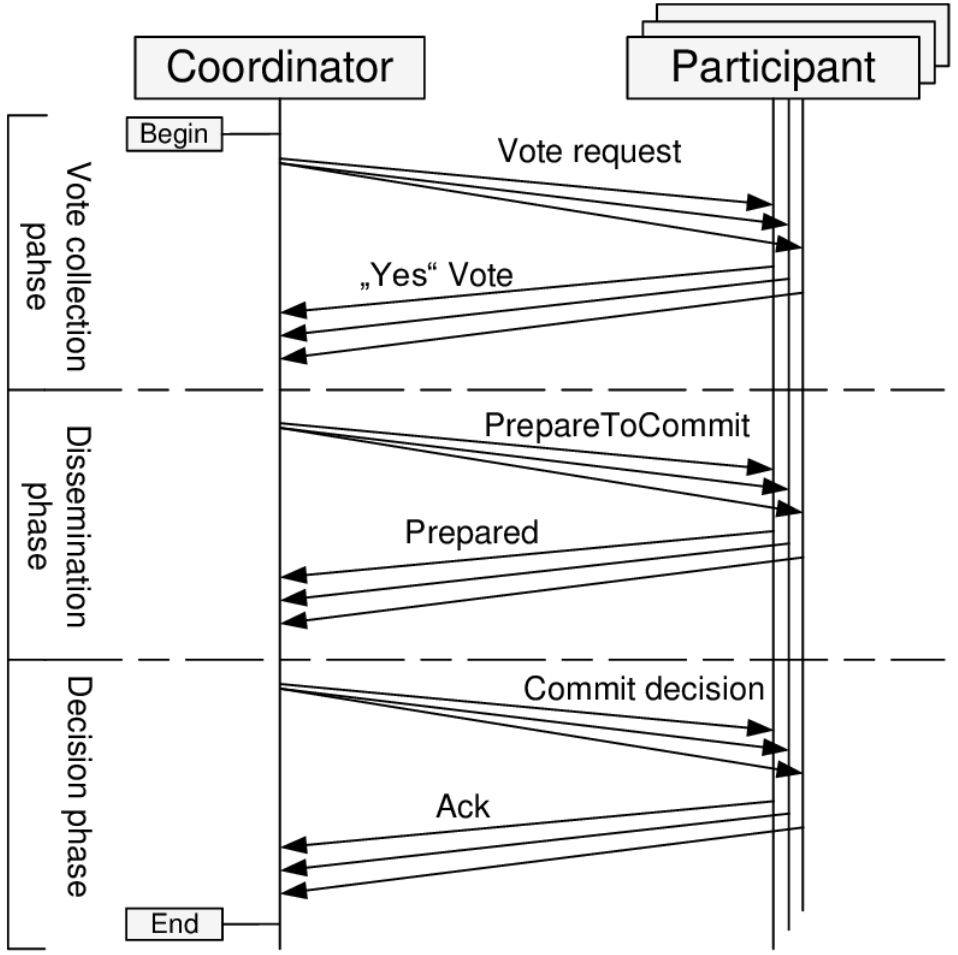
```
Microsoft Distributed Transaction Coordinator (MS DTC)
failed to reenlist citing that the database RMID does
not match the RMID [xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx]
associated with the transaction.  Please manually resolve
the transaction.

SQL Server detected a DTC/KTM in-doubt transaction with UOW
{yyyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyyyy}.Please resolve it
following the guideline for Troubleshooting DTC Transactions.
```

2 phase transactions

- What if coordinator fail in the middle of decision phase?
  - Locks, time outs
  - Temporary inconsistent data (if no common clock)
- Coordinator should be in replica set!
- Scale coordinators
  - More then one commit log
  - How to get serializability?
- 2 phase transactions problems examples

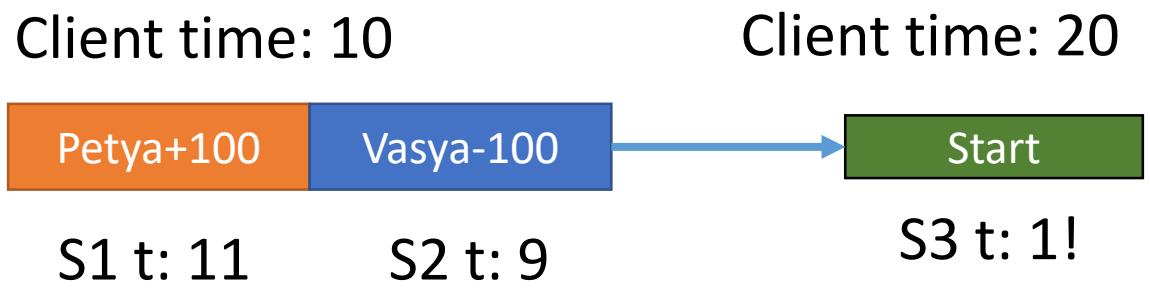
All Fancy DB still use this!

3 Phase commit ;)



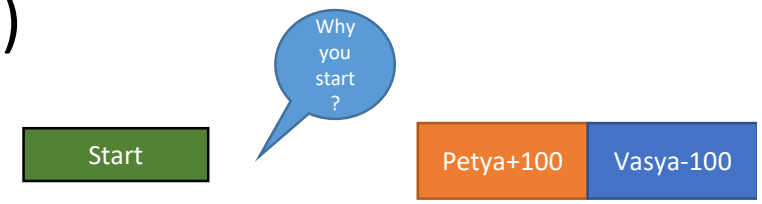
Synchronized time

- Need monolithic increasing values
  - Causality, Time and Happens-Before
  - Global order of transactions
  - Consistent reads
- We can not use server or client time in distributed system



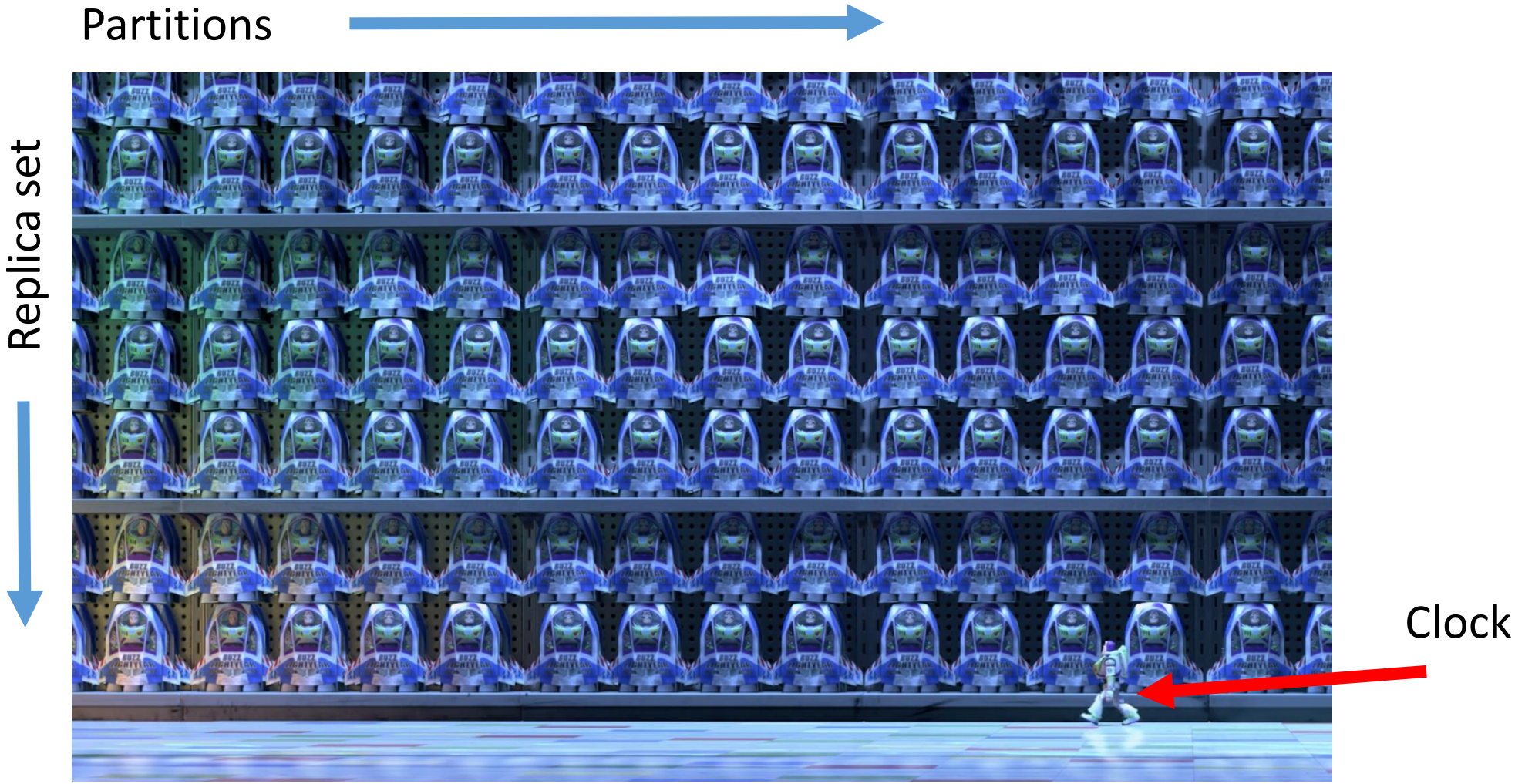
Common clock implementations

- Logical clock (lamport, vector…)
  - Partial order
  - Too many messages
- TrueTime
  - Spanner attempt to synchronize server clocks
  - Use commit timeouts to make sure you get proper order of trans
  - Best solution for geo distributed databases
- Hybrid clocks
  - Base on server clock
  - Additional counter to get unique order value
  - Limited number of message to get correct value
- Single timestamp oracle (server)
  - Surprisely fast solution for local networks



- **No Consistency** - In this mode there are no external consistency guarantees, transactions are assigned timestamps from each server's physical clock and no guarantee is made that reads are consistent or repeatable.
- **HybridTime Consistency** - In this mode our implementation guarantees the global consistency as Spanner, absent hidden channels, but using HybridTime instead of commit-wait. Clients choosing this consistency mode on writes must make sure that the timestamp that is received from the server is propagated to other servers and/or clients. Within the same client process, timestamps are automatically propagated on behalf of the user. If there are hidden channels, i.e. if somehow a write or a read causes another write or read without the timestamp being propagated, there is no guarantee of external consistency.
- **Commit-wait Consistency** - In this mode our implementation guarantees the same external consistency semantics as Spanner by also using commit-wait in the way described in the original paper. However instead of using TrueTime, which is a proprietary and private API, we implemented commit-wait on top of the widely used Network Time Protocol (NTP). Hence, in this consistency mode we support hidden channels.

Modern databases is more like….



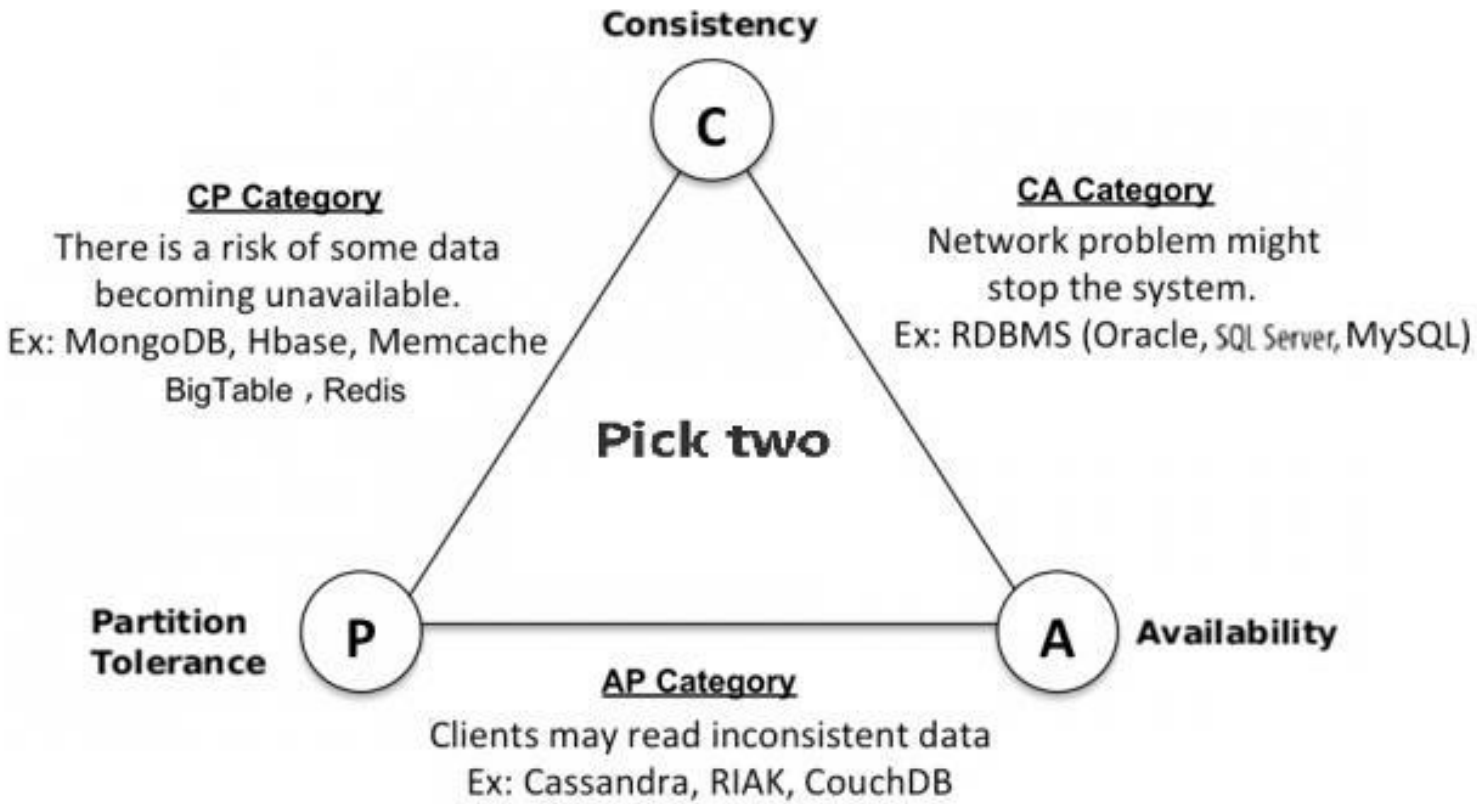
Distributed transactions trade offs

- A lot of DB does not support distributed transactions!
  - “Single partition key transaction”
  - Lightweight transaction (CAS)
  - “Single server/replica group transactions”
    - Adopt your partition key
- No global indexes (required transactions)
  - Each node contains local index, index base request is sent to all partitions
- Use 2ph commits without global clock
  - Temporary no consistency
  - No isolation

Distributed transactions trade offs

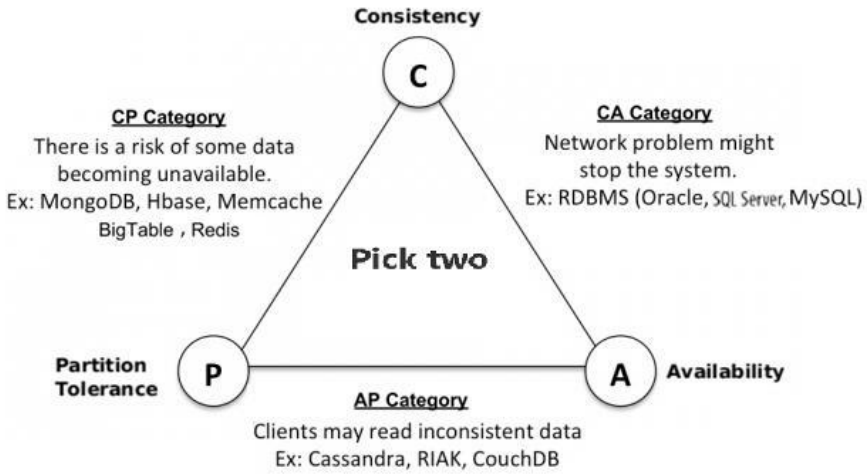
- A lot of DB does not support distributed transactions
  - “Single partition key transaction”
  - Lightweight transaction (CAS)
  - “Single server/replica group transactions”
    - Adopt your partition key
- No global indexes (required transactions)
  - Each node contains local index, index base request is sent to all partitions
- Use 2ph commits without global clock
  - Temporary no consistency
  - No isolation

CAP theorem


Consistency
C
CP Category
There is a risk of some data becoming unavailable.
Ex: MongoDB, Hbase, Memcache BigTable , Redis
CA Category
Network problem might stop the system.
Ex: RDBMS (Oracle, SQL Server, MySQL)
Pick two
Partition Tolerance
P
A
Availability
AP Category
Clients may read inconsistent data
Ex: Cassandra, RIAK, CouchDB

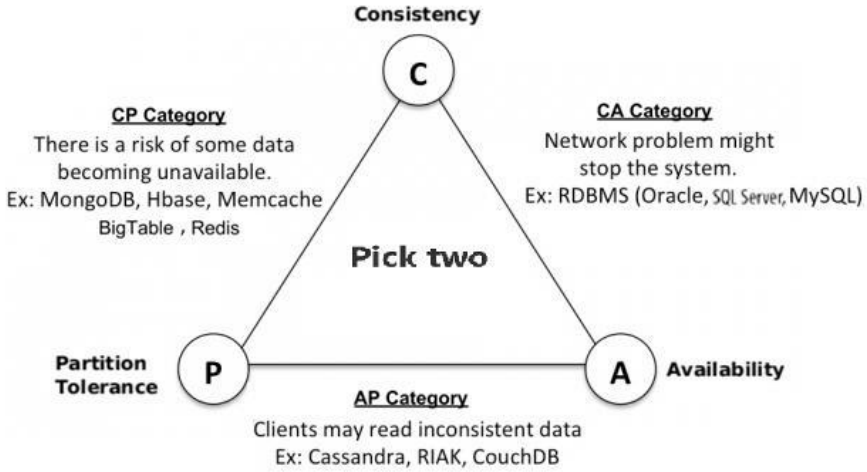
CAP theorem

- Not about transactions!
  - Atomic register read/write only
- Consistency means linearizability
  - Not serializability consistency level
- Partition tolerance
  - Loose connectivity or packages among some part of the distributed system
  - for example connection between Master and Slave: split brain



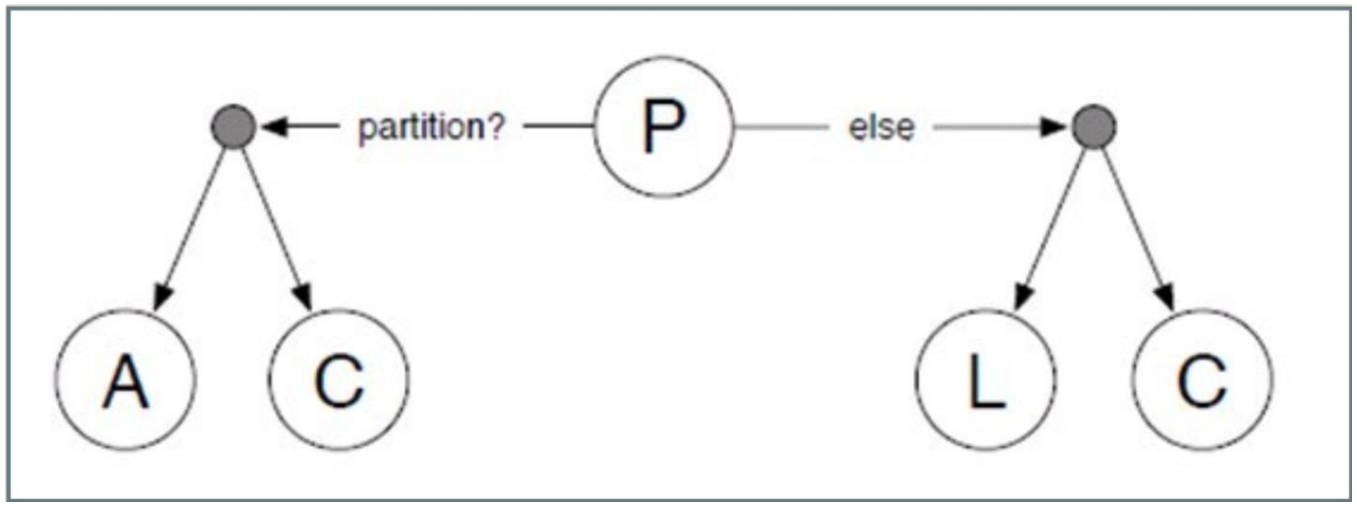
CAP theorem

- Not about transactions!
  - Atomic register read/write only
- Consistency means linearizability
  - Not serializability consistency level
- Partition tolerance
  - Loose connectivity or packages among some part of the distributed system
  - for example connection between Master and Slave: split brain



PACELC – not a theorem

- Partitioned?
  - Availability
  - Consistency
- Else (normal work)
  - Latency
  - Consistency



DDBS	P+A	P+C	E+L	E+C
BigTable/HBase		✓		✓
Cassandra	✓		✓[a]	
Cosmos DB		✓	✓ [b]	
Couchbase		✓	✓	✓
DynamoDB	✓		✓[a]	
FaunaDB[8]		✓	✓	✓
Hazelcast IMDG[5][6]	✓	✓	✓	✓
Megastore		✓		✓
MongoDB	✓			✓
MySQL Cluster		✓		✓
PNUTS		✓	✓	
Riak	✓		✓[a]	
VoltDB/H-Store		✓		✓

Read documentation!

- Most of databases can be tuned for different trade off
  - Check you requirements
  - Properly select configuration
- Do not blindly trust benchmark
  - Read configuration they used
  - If you need transaction, expect much slower results