Having Your Cake, And Eating It Too

Having Your Cake And Eating It Too

An intro to the history of OLTP DBs, NoSQL databases, and an overview of a new breed of DBs that promise the best of both worlds [1]

---

[1] Mostly. Except when it doesn't. There is no free lunc.. err .. cake. Free lunch is a lie.

History of (OLTP) DBs

- **O**nline **T**ransaction **P**rocessing
- Mostly single machine
- Scaled vertically
- Traditionally RDBMs
- Typically fulfill ACID properties

ACID? 🤪

ACID

Atomicity
Transactions succeed completely or fail completely

Consistency
Operations must bring the DB to a *valid* state respecting DB constraints, and future reads

Isolation
No dirty reads, non-repeatable reads, or phantom reads

Durability
Commits stay committed, through network/hardware failure

Problems with RDMBs

Problems with RDMBs

- difficult to horizontally scale, need to shard data in an app aware manner

Problems with RDMBs

- difficult to horizontally scale, need to shard data in an app aware manner
- expensive join operations; very slow when scaling out

Problems with RDMBs

- difficult to horizontally scale, need to shard data in an app aware manner
- expensive join operations; very slow when scaling out
- strong consistency guarantees comes at the expense of blocking writes when reads are happening

Problems with RDMBs

- difficult to horizontally scale, need to shard data in an app aware manner
- expensive join operations; very slow when scaling out
- strong consistency guarantees comes at the expense of blocking writes when reads are happening
- modern apps have missive writes

Problems with RDMBs

- difficult to horizontally scale, need to shard data in an app aware manner
- expensive join operations; very slow when scaling out
- strong consistency guarantees comes at the expense of blocking writes when reads are happening
- modern apps have missive writes
- many apps have more relaxed consistency needs

How do we solve these problems?

Put another way, what tradeoffs can we make?

CAP Theorem

CAP Theorem

- **C**onsistency

CAP Theorem

- **C**onsistency
  - retreive the most recent write or an error

CAP Theorem

- **C**onsistency
  - retreive the most recent write or an error
- **A**vailability (100%)

CAP Theorem

- **C**onsistency
  - retreive the most recent write or an error
- **A**vailability (100%)
  - retreive a non-error response (no guarantee it's the most recent)

CAP Theorem

- **C**onsistency
  - retreive the most recent write or an error
- **A**vailability (100%)
  - retreive a non-error response (no guarantee it's the most recent)
- **P**artition Tolerance

CAP Theorem

- **C**onsistency
  - retreive the most recent write or an error
- **A**vailability (100%)
  - retreive a non-error response (no guarantee it's the most recent)
- **P**artition Tolerance
  - function with network failures or delays between nodes

Clarifiying a common misconception

CAP says during a *partition*, you must choose between *availability* and *consistency*.

However, if there is no networking failure, you can have both consistency and availability.

NoSQL

- We always need parition tolerance in a distributed system
- In order to scale, pick **A**vailability over **C**onsistency
- Accept 'Eventual Consistency'
- Many types, but of the BigTable/Dynamo family:
  - BigTable
  - DynamoDB
  - Cassandra
  - Riak

Problems with SQL/NoSQL systems

- RDBMs give us strong consistency, but poor performance
  - Due to 2PL, writes are blocked while strong reads are happening
- NoSQL DBs give us great performance, but eventual consistency
  - You can get stale reads, because all nodes may not have converged to the latest commit

How do we solve these problems?

🕦 The Arrow of Time

Time is the key to solving these problems: it lets us establish a coherent ordering of events across nodes, and thus pin down causality, and the rules that follow:

🕐 The Arrow of Time

Time is the key to solving these problems: it lets us establish a coherent ordering of events across nodes, and thus pin down causality, and the rules that follow:

- ensure resources that locks are respected (ie, if transaction A has not yet released, stop transaction B from accessing the resource)

🕦 The Arrow of Time

Time is the key to solving these problems: it lets us establish a coherent ordering of events across nodes, and thus pin down causality, and the rules that follow:

- ensure resources that locks are respected (ie, if transaction A has not yet released, stop transaction B from accessing the resource)
- make guarantees about consistency between nodes, regions, and even continents

Timing events in a distributed system

Timing events in a distributed system

1. Logical clock (eg, vector clocks)

Timing events in a distributed system

1. Logical clock (eg, vector clocks)
    - Complex to deal with

Timing events in a distributed system

1. Logical clock (eg, vector clocks)
    - Complex to deal with
2. Global clock

Timing events in a distributed system

1. Logical clock (eg, vector clocks)
   - Complex to deal with
2. Global clock
   - Simple to deal with, but can have drift

Linearizability

The ability to absoutely order events, across nodes, in order to guarantee **Consistency** of data across nodes. Atomic/GPS clocks enable systems to agree on time without having to consult a single source of truth.

Serializability

Transactions should behave as if they had a lock over the data they are reading/writing. Easy in a non-distributed DB, but we need atomic clocks for a distributed environemnt. Provides **Isolation**.

==> External Consistency

With these two properties, you have external consistency. You won't suffer from stale/dirty reads, can read data from one node that was committed in another, and events won't appear out of order.

Google Spanner

- CP
- C(A)P
  - A is effectively more than 9 9s
  - mostly due to config/user errors, only 7.6% due to network (ie, partition) reasons
- serializabity from lock
- external consistency (linearizability) from `TrueTime`

How Spanner does it

How Spanner does it

- Traditional DBs use 2PL for external consistency

How Spanner does it

- Traditional DBs use 2PL for external consistency
  - anti availability, since all nodes have to be up

How Spanner does it

- Traditional DBs use 2PL for external consistency
  - anti availability, since all nodes have to be up
- TrueTime (synchronized clock)

How Spanner does it

- Traditional DBs use 2PL for external consistency
  - anti availability, since all nodes have to be up
- TrueTime (synchronized clock)
  - multi master horizontal scaling

How Spanner does it

- Traditional DBs use 2PL for external consistency
  - anti availability, since all nodes have to be up
- TrueTime (synchronized clock)
  - multi master horizontal scaling
  - GPS receivers and atomic clocks

How Spanner does it

- Traditional DBs use 2PL for external consistency
  - anti availability, since all nodes have to be up
- TrueTime (synchronized clock)
  - multi master horizontal scaling
  - GPS receivers and atomic clocks
  - "if T2 starts to commit after T1 finishes committing, then the timestamp for T2 is greater than the timestamp for T1"

How Spanner does it

- Traditional DBs use 2PL for external consistency
  - anti availability, since all nodes have to be up
- TrueTime (synchronized clock)
  - multi master horizontal scaling
  - GPS receivers and atomic clocks
  - "if T2 starts to commit after T1 finishes committing, then the timestamp for T2 is greater than the timestamp for T1"
  - 7ms wait: nodes must wait before they report a commit

How Spanner does it

- Traditional DBs use 2PL for external consistency
  - anti availability, since all nodes have to be up
- TrueTime (synchronized clock)
  - multi master horizontal scaling
  - GPS receivers and atomic clocks
  - "if T2 starts to commit after T1 finishes committing, then the timestamp for T2 is greater than the timestamp for T1"
  - 7ms wait: nodes must wait before they report a commit
- 2PC, strict two phase locking

How Spanner does it

- Traditional DBs use 2PL for external consistency
  - anti availability, since all nodes have to be up
- TrueTime (synchronized clock)
  - multi master horizontal scaling
  - GPS receivers and atomic clocks
  - "if T2 starts to commit after T1 finishes committing, then the timestamp for T2 is greater than the timestamp for T1"
  - 7ms wait: nodes must wait before they report a commit
- 2PC, strict two phase locking
  - Paxos groups to achieve consensus on updates

🍰 Questions?

Appendix

Spanner, TrueTime & The CAP Theorem

NoSQL (Wikipedia)

NewSQL (Wikipedia)

ACID (Wikipedia)>)

CAP Theorem

Cloud Spanner: TrueTime and external consistency