Distributed Systems

ECE428

Lecture 24

Adopted from Spring 2021

Today’s focus

- Brief overview of key-value stores
- Distributed Hash Tables
  - Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.
- Key-value stores in the cloud
  - How to run large-scale distributed computations over key-value stores?
    - Map-Reduce Programming Abstraction
  - How to design a large-scale distributed key-value store?
    - Case-study: Facebook’s Cassandra

Distributed datastores

- Distributed datastores
  - Service for managing distributed storage.
- Distributed NoSQL key-value stores
  - BigTable by Google
  - HBase open-sourced by Yahoo and used by Hadoop.
  - DynamoDB by Amazon
  - Cassandra by Facebook
  - Voldemort by LinkedIn
  - MongoDB
  - …
- *Spanner is not a NoSQL datastore. It’s more like a distributed relational database.*

Key-value/NoSQL Data Model

- NoSQL = “Not Only SQL”
- Necessary API operations:
  get(key) and put(key, value)
  - And some extended operations, e.g., “CQL” in Cassandra key-value store

- Tables
  - Like RDBMS tables, but …
  - May be unstructured: May not have schemas
    - Some columns may be missing from some rows
  - Don’t always support joins or have foreign keys
  - Can have index tables, just like RDBMSs

How to design a distributed key-value datastore?

Design Requirements

- High performance, low cost, and scalability
  - Performance: high throughput, low latency for reads/writes
  - Low TCO (total cost of operation)
  - Need for fewer system administrators
  - Incremental scalability
    - Scale out: add more machines.
    - Scale up: upgrade to powerful machines.
    - *Cheaper to scale out than to scale up.*

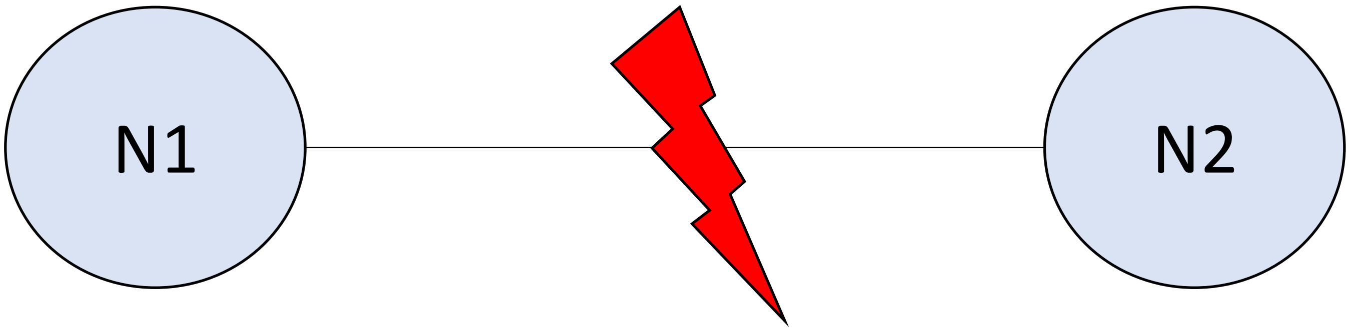
Design Requirements (contd)

- Consistency: reads return latest written value by any client (all nodes see same data at any time).
  - *This is different from the C in ACID properties for transaction semantics!*
- Availability: every request received by a non-failing node must result in a (quick) response.
  - Follows from requirement for high performance.
  - Avoid single-point of failure: replication across multiple nodes.
- Partition-tolerance: the system continues to work in spite of network partitions.

CAP Theorem

- Consistency: reads return latest written value by any client (all nodes see same data at any time).
- Availability: every request received by a non-failing node must result in a (quick) response
- Partition-tolerance: the system continues to work in spite of network partitions.
- In any distributed system, we can only guarantee at most 2 out of the above 3 properties.
  - Proposed by Eric Brewer (UC Berkeley)
  - Subsequently proved by Gilbert and Lynch (NUS and MIT)

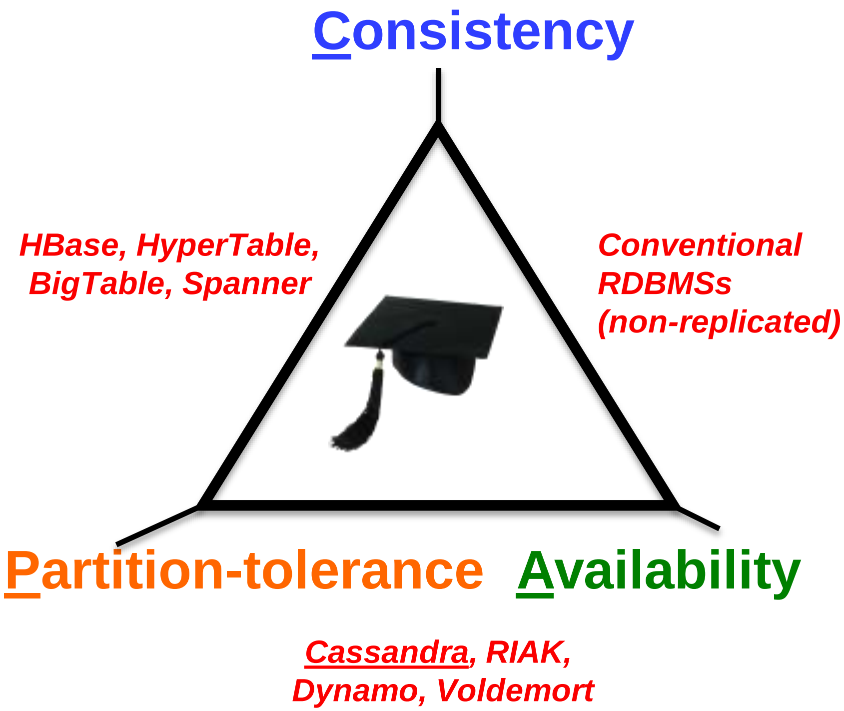
CAP Theorem



- Data replicated across both networks N1 and N2.
- If network is partitioned, N1 can no longer talk to N2.
- Consistency + availability require that N1 and N2 can talk.
  - no partition-tolerance.
- Partition-tolerance + consistency:
  - only respond to requests at N1 (no availability at N2).
- Partition-tolerance + availability:
  - writes at N1 will not be captured by reads at N2 (no consistency).

CAP Tradeoff

- Starting point for NoSQL Revolution
- A distributed storage system can achieve at most two of C-A-P.
- When partition-tolerance is important, you have to choose between consistency and availability.
- Check *Spanner CAP-theorem* white paper on Blackboard



Modern key-value stores vs. RDBMS

- RDBMS provide ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability

- Many modern key-value stores provide BASE
  - <u>B</u>asically <u>A</u>vailable <u>S</u>oft-state <u>E</u>ventual Consistency
  - Prefers Availability over Consistency

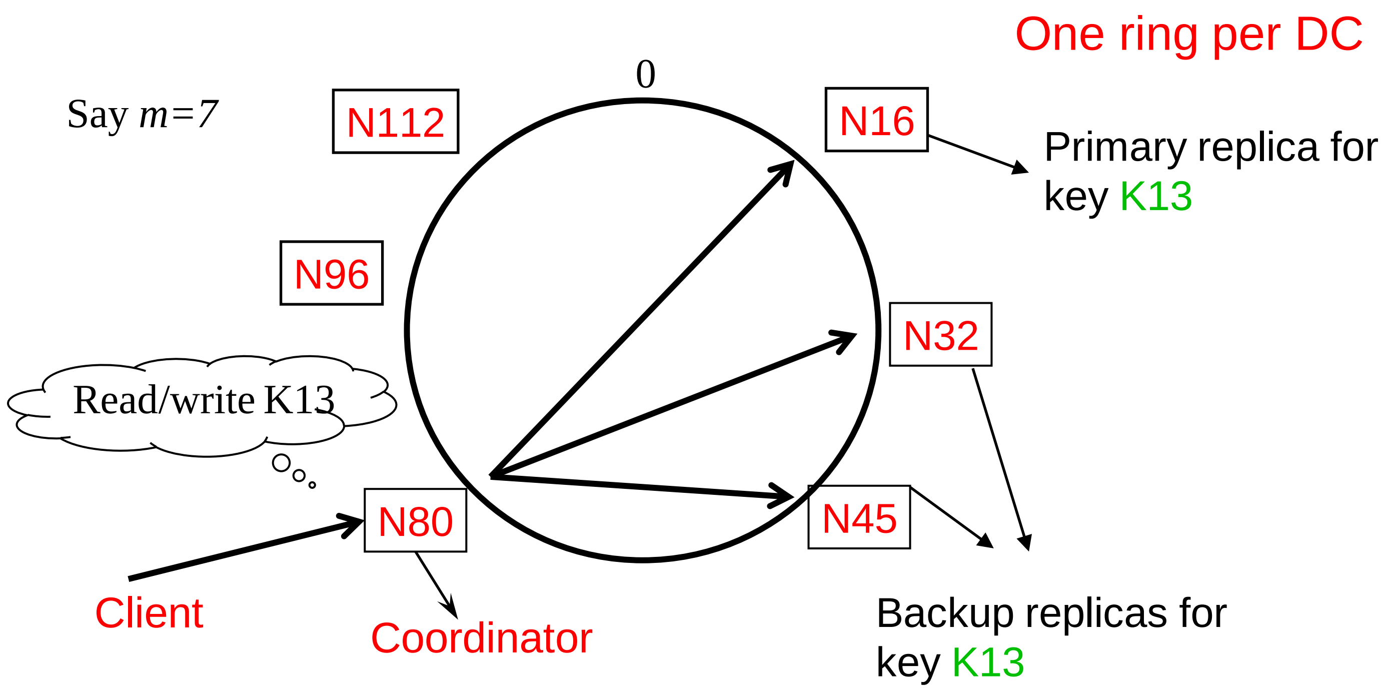
Case Study: Cassandra

Cassandra

- A distributed key-value store.
- Intended to run in a datacenter (and across DCs).
- Originally designed at Facebook.
- Open-sourced later, today it is an Apache project.
- Some of the companies that use Cassandra in their production clusters.
  - IBM, Adobe, HP, eBay, Ericsson, Symantec
  - Twitter, Spotify
  - Netflix: uses Cassandra to keep track of your current position in video you’re watching

Data Partitioning: Key to Server Mapping

- How do you decide which server(s) a key-value resides on?



Cassandra uses a ring-based DHT but without finger or routing tables.

Partitioner

- The component responsible for a key to server mapping (a hash function).

- Two types:
  - *Chord-like hash partitioning*
    - *Murmer3Partitioner* (default): uses *murmer3* hash function.
    - *RandomPartitioner*: uses MD5 hash function.
  - *ByteOrderedPartitioner*: Assigns ranges of keys to servers.
    - Easier for *range queries* (e.g., get me all twitter users starting with [a-d])

- Determines the primary replica for a key.

Replication Policies

Two options for replication strategy:

1. Simple Strategy:
   - First replica placed based on the Partitioner.
   - Remaining replicas placed clockwise in relation to the primary replica.
2. Network Topology Strategy: for the multi-DC deployments
   - Two or three replicas per a DC.
   - The per DC replicas:
     - First replica placed according to Partitioner.
     - Then go clockwise around the ring until reaching the next rack.

Topics to cover next

- Writes.
- Reads.
- Cluster membership.
- Eventual consistency model.

Writes

- Need to be lock-free and fast (no reads or disk seeks).
- Client sends its write request to one coordinator node in the Cassandra cluster.
  - Coordinator may be per-key, per-client, or per-query.
- Coordinator uses Partitioner to send query to all replica nodes responsible for the key.
- When X replicas respond, Coordinator returns the following acknowledgement to the client:
  - X = any one, majority, all….(consistency spectrum)
  - More details later!

Writes: Hinted Handoff

- Always writable: Hinted Handoff mechanism
  - If any replica is down, the coordinator writes to all other replicas, and keeps the write locally until a down replica comes back up.
  - When all replicas are down, Coordinator (the front end) buffers writes (for up to a few hours).

Data Partitioning and Replication

- Partitioner: identifies primary replica for a key
  - hash-based or range based.

- Replication in multi-DC environments
  - replicate across datacenters.
  - replicate across different racks within a datacenter.

- Writes:
  - Client send writes to the *coordinator*.
  - Coordinator sends query to all replicas.
  - Waits for X replicas to respond before returning acknowledgement to client.
    - X determines consistency level. To be discussed.
  - Hinted handoffs ensure writes eventually written to all replicas.

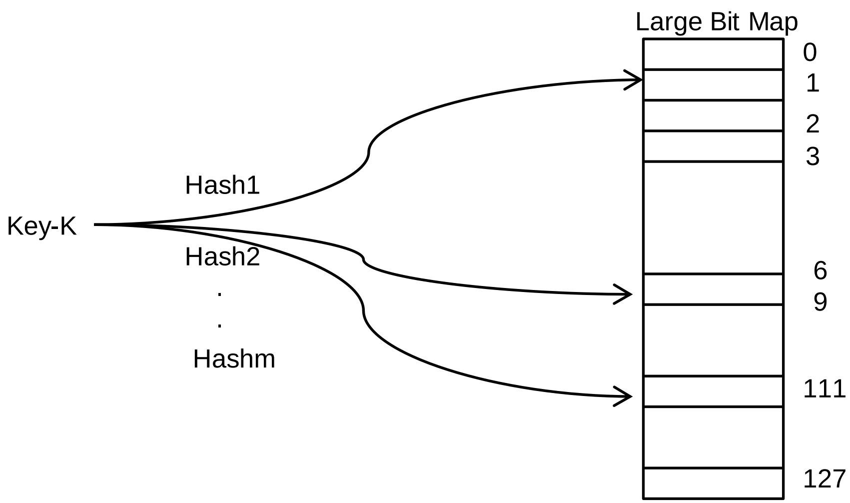
Writes at a replica node

On receiving a write

1. Log it in disk commit log (for failure recovery)
2. Make changes to corresponding Memtables
   - **Memtable** = In-memory representation of key-value pairs
   - Cache which can be searched by a key
   - Write-back cache as opposed to write-through
3. When Memtable is full or old, flush it to the disk
   - Data file: **SSTable** (Sorted String Table, list of key-value pairs, sorted by the key)
   - Index file: SSTable of (key, position in data SSTable) pairs
   - And a Bloom filter (for efficient search) – next slide.

Bloom Filter

- Compact way of representing a set of items.
- Checking for existence in a set is cheap.
- Some probability of false positives: an item not in the set may check true as being in the set.
- No false negatives.



On insert, set all hashed bits.

On check-if-present,
return true if all hashed bits set.

- False positives

False positive rate very low

- m=4 hash function, 100 items, 3200 bits
- FP rate = 0.02%

Compaction

- Data updates accumulate over time and over multiple SSTables.
- Then SSTables need to be compacted.
- The process of compaction merges SSTables, i.e., by merging updates for a key.
- Runs periodically and locally at each server.

Deletes

Delete: don’t delete item right away

- Write a tombstone for the key.
- Eventually, when compaction encounters the tombstone, it will delete the item

Reads

- Coordinator contacts X replicas (e.g., in same rack)
  - Coordinator sends read to replicas that have responded quickest in the past.
  - When X replicas respond, Coordinator returns the latest-timestamped value from among those.
  - X = based on consistency spectrum (more later).
- Coordinator also fetches value from other replicas
  - Checks consistency in background, and initiate a read repair if any two values are different.
  - This seeks to eventually bring all replicas up to date.
- At a replica
  - Read looks at Memtables first, then at SSTables.
  - A row may be split across multiple SSTables => reads need to touch multiple SSTables (reads slower than writes, but still fast).

Cross-DC coordination

- Replicas may span multiple datacenters.
- Per-DC coordinator elected to coordinate with other DCs.
- Election done via Zookeeper which runs a Bully algorithm variant.

Membership

- Any server in cluster could be the leader.
- So every server needs to maintain a list of all the other servers that are currently in the cluster.
- List needs to be updated automatically as servers join, leave, and fail.

Cluster Membership

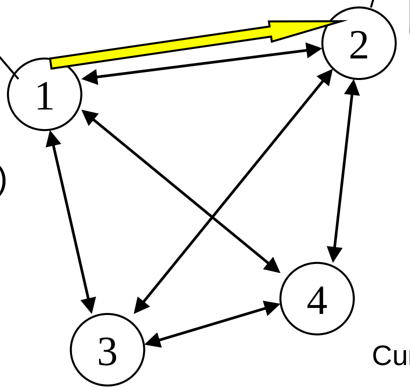
Cassandra uses gossip-based cluster membership

1	10120	66
2	10103	62
3	10098	63
4	10111	65

Address

Heartbeat Counter

Time (local)



1	10118	64
2	10110	64
3	10090	58
4	10111	65

(old)

1	10120	70
2	10110	64
3	10098	70
4	10111	65

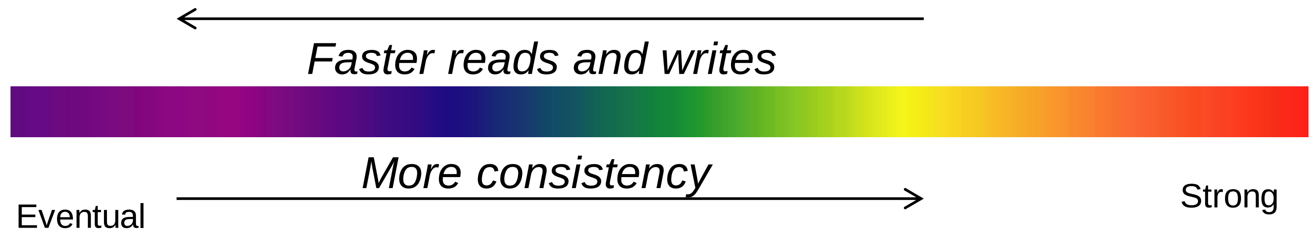
(updated)

Current time : 70 at node 2

(asynchronous clocks)

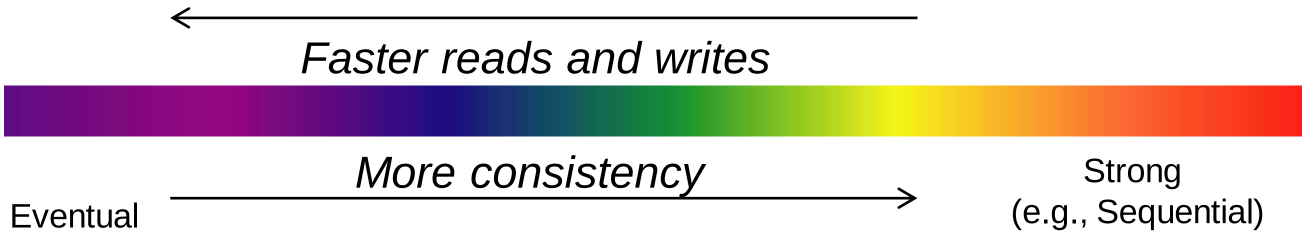
- Nodes periodically gossip their membership list
- On receipt, the local membership list is updated, as shown
- If any heartbeat older than Tfail, node is marked as failed

Consistency Spectrum



Eventual Consistency

- Cassandra offers Eventual Consistency
  - If writes to a key stop, all replicas of key will converge.
  - Originally from Amazon's Dynamo and LinkedIn's Voldemort systems.



Cassandra write and read recap

- Writes
  - Client sends write request to *Coordinator*.
  - Coordinator writes to all replicas.
  - Waits for X replicas to respond before returning acknowledgement to the client.
  - Hinted handoff: if a replica is down, it receives the write request once it comes back up.
- Reads
  - Client sends read request to *Coordinator*.
  - Coordinator contacts X replicas, and returns the latest returned value.
  - Read repair: After returning a response, Coordinator continues with fetching values from other replicas, and initiates repairs to outdated values.

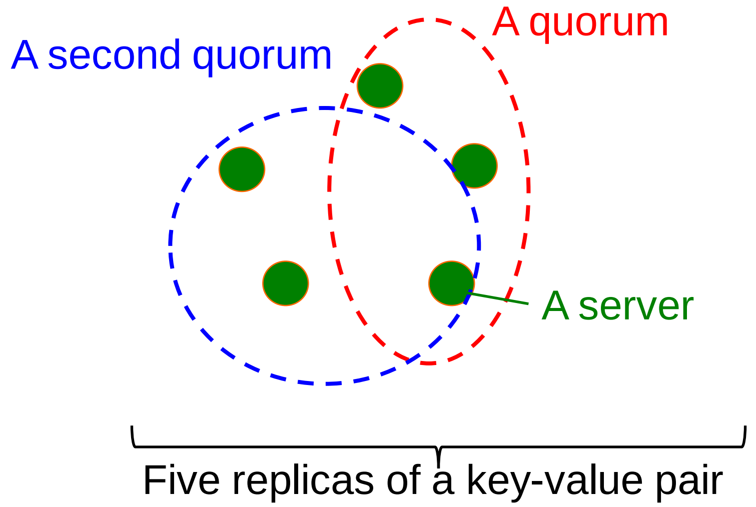
Consistency levels: value of X

- Cassandra has consistency levels.
- Client is allowed to choose a consistency level for each operation (read/write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator caches write and replies quickly to client
  - ALL: all replicas
    - Ensures strong consistency, but slowest
  - ONE: at least one replica
    - Faster than ALL, but cannot tolerate a failure
  - QUORUM: quorum across all replicas in all DCs

Quorums?

In a nutshell:

- Quorum = (typically) majority
- Any two quorums intersect
  - Client 1 does a write in red quorum
  - Client 2 does read in blue quorum
- At least one server in blue quorum returns latest write
- Quorums faster than ALL, but still ensure strong consistency
- Several key-value/NoSQL stores (e.g., Riak and Cassandra) use quorums.



Read Quorums

- Reads
  - Client specifies value of R ($\leq N$ = total number of replicas of that key).
  - R = read consistency level.
  - Coordinator waits for R replicas to respond before sending result to client.
  - In background, coordinator checks for consistency of remaining (N-R) replicas, and initiates read repair if needed.

Write Quorums

- Client specifies W ($\leq$ N)
- W = write consistency level.
- Client writes new value to W replicas and returns when it hears back from all.
  - Default strategy.

Quorums in Detail

- R = read replica count, W = write replica count
- Necessary conditions for consistency:
  1. W+R > N
     - Write and read intersect at a replica. Read returns latest write.
  2. W > N/2
     - Two conflicting writes on a data item don't occur at the same time.
- Select values based on application
  - (W=N, R=1):
    - great for read-heavy workloads
  - (W=1, R=N):
    - great for write-heavy workloads with no conflicting writes.
  - (W=N/2+1, R=N/2+1):
    - great for write-heavy workloads with potential for write conflicts.
  - (W=1, R=1):
    - very few writes and reads / high availability requirement.

Cassandra Consistency Levels

- Client is allowed to choose a consistency level for each operation (read/write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator may cache write and reply quickly to client
  - ALL: all replicas
    - Slowest, but ensures strong consistency
  - ONE: at least one replica
    - Faster than ALL, and ensures durability without failures
  - QUORUM: quorum across all replicas in all DCs
    - Global consistency, but still fast
  - EACH_QUORUM: quorum in every DC
    - Lets each DC do its own quorum: supports hierarchical replies
  - LOCAL_QUORUM: quorum in coordinator's DC
    - Faster: only waits for quorum in first DC client contacts

Eventual Consistency

- Sources of inconsistency:
    - Quorum condition not satisfied R + W < N.
        - R and W are chosen as such.
        - when write returns before W replicas respond.
            - Sloppy quorum: when value stored elsewhere if intended replica is down, and later moved to the replica when it is up again.
    - When local quorum is chosen instead of global quorum.
- Hinted-handoff and read repair achieve *eventual consistency*.
    - If all writes (to a key) stop, then all its values (replicas) will converge eventually.
    - May still return stale values to clients (e.g., if many back-to-back writes).
    - But works well when there a few periods of low writes – system converges quickly.

Cassandra vs. RDBMS

- MySQL is one of the most popular RDBMS (and has been for a while)
- Comparing on 50+ GB data:
- MySQL
  - Writes 300 ms avg
  - Reads 350 ms avg
- Cassandra (orders of magnitude faster)
  - Writes 0.12 ms avg
  - Reads 15 ms avg

Other similar NoSQL stores

- Amazon’s DynamoDB
  - Cassandra’s data partitioning, replication, and eventual consistency strategies inspired from Dynamo.
  - Uses sloppy quorum as the default mechanism for eventual consistency with availability.
  - Uses vector clocks to capture causality between different versions of an object.
  - Dynamo: Amazon’s Highly Available Key-value Store, SOSP’2007.
- LinkedIn’s Voldemort
  - Inspired from DynamoDB.
- …..

Is it a good idea to trade-off consistency for availability?

A recent tweet by a research on Distributed Systems:

> Due to a shopping cart weak consistency error, my mom has found herself with an extra 4 dozen eggs and 4 pounds of beets she didn't mean to order.
>
> Isn't this what I've been warning everyone about for years?
>
> 11 6 94

Summary

- CAP theorem: cannot only achieve 2 out of 3 among consistency, availability, and partition-tolerance.

- Partition-tolerance is required in distributed datastores.
  - Choose between consistency and availability.

- Many modern distributed NoSQL key-value stores (e.g. Cassandra) choose availability, and provide only eventual consistency.