Dynamo

Notes from reviews

- Evaluation doesn't cover all design goals (e.g. incremental scalability, heterogeneity)
- Is it research?
- Complexity?
- How general?

Dynamo: Motivation

- Largest e-commerce site
  - 75K query/sec (my estimation)
    - 500 req/sec * 150 query/req
  - O(10M) users and Many items
- Why not RDBMS?
  - Not easy to scaling-out or load balancing
  - Many components only need primary key access
- Databases are required just for Amazon
  - *Dynamo* for primary key accesses
  - *SimpleDB* for complex queries
  - *S3* for large files
- Dynamo is used for
  - Shopping carts, customer preferences, session management, sales rank, and product catalogs

Dynamo Motivation

- Normal database not the right fit for some applications
  - Strict consistency too expensive, doesn't tolerate all the right failures
  - Applications may be able to use internal knowledge to handle certain inconsistencies
  - Expensive to purchase, to scale to the needed size

Evaluating Commercial Products

- There is a temptation to say "it is really used, it must be good"
  - Not always the case – there can still be bad design decisions
  - But generally shows the different motivations, considerations of industry
  - Example: bigtable from google vs DB
    - DB can be 10x faster in some cases

Dynamo: A Database?

- This is basically a database
- But not your conventional database
- Conventional (relational) database:
  - Data organized in tables
  - Primary and secondary keys
  - Tables sorted by primary/secondary keys
  - Designed to answer any imaginable query
  - Does not scale to thousands of nodes
  - Difficult to replicate
- Amazon's Dynamo
  - Access by primary key only

Dynamo: Features

- Key, value store
  - Distributed hash table
- High scalability
  - No master, peer-to-peer
  - Large scale cluster, maybe O(1K)
- Fault tolerant
  - Even if an entire data center fails
  - Meets latency requirements in the case

ACID Properties

- Atomicity – yes
  - Updates are atomic by definition
  - There are no transactions
- Consistency – no
  - Data is **eventually** consistent
  - Loose consistency is tolerated
  - Reconciliation is performed by the client
  - Database consistency: yes (because only single-value updates)
- Isolation
  - yes isolation – one update at a time
- Durability – yes
  - Durability is provided via replication

High Availability

- Good service time is key for Amazon
- Not good when a credit card transaction times out
- Service-level agreement: the client's response must be answered within 300ms
- Must provide this service for 99.9% of transactions at the load of 500 requests/second.
  - Requires optimistic protocols that can return asynchronously

The Cost of Respecting the SLA

- Loose consistency
  - Synchronous replica reconciliation during the request cannot be done
  - We contact a few replicas, if some do not reply, request is considered failed
- When to resolve conflicting updates? During reads or during writes?
  - Usually resolved during writes
  - Dynamo resolves it during reads
  - Motivation: must have an **always writable** data store (can't lose customer shopping card data)
- QUESTION: what happens under absurd failures? e.g. multiple data centers fail?
  - Cannot handle all possible failures; it is a probability game

Consistency Models

- Server-side:
  - Concerns consistency of data on disks, values that could be returned to clients
  - Depends on quorum protocols & majorities
- Client side: many versions
  - Strong: read previous write
  - Weak: no guarantees
  - Eventual: if wait long enough without failures, get most recent value
  - Causal: if A tells B it updated data x, B will see updated version
    - Isis
  - Read-your-writes: if A writes value x and then reads it, guaranteed to see its write
  - Session: read-your-writes within a "session" that can end
  - Monotonic read consistency: reads only get newer in versions

System Interface

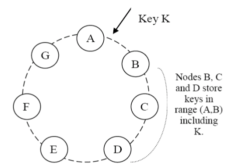
- get (key)
  - Locate object replicas
  - Return:
    - A single object
    - A list of objects with conflicting versions
    - Context (opaque information about object versioning)
- put (key, value, context)
  - Determines where the replicas should be placed
  - Writes them to disk
  - Context helps write things back to same place, help with versioning of data
- Requirements for clients
  - Don't need to know *ALL* nodes, unlike memcache clients
    - Requests can be sent to any node

Design Consideration

- What are design goals?
  - Sacrifice strong consistency for availability **if needed**
  - Conflict resolution is executed during ***read*** instead of ***write***, i.e. "always writeable".
  - Other principles:
    - Incremental scalability.
    - Symmetry.
    - Decentralization.
    - Heterogeneity.
  - ***WHY?***

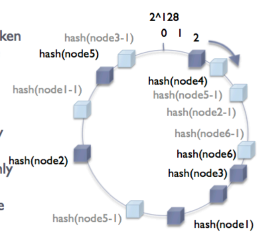
Partition Algorithm

- Consistent hashing: the output range of a hash function is treated as a fixed circular space or "ring".
- "Virtual Nodes": Each node can be responsible for more than one virtual node.



Virtual Nodes

- Virtual nodes
  - Multiple positions are taken by a single physical node
    - O(100) virtual/physical
- Advantages
  - Keys are more uniformly distributed statistically
  - Remapped keys are evenly dispersed across nodes
  - # of virtual nodes can be determined based on capacity of physical node



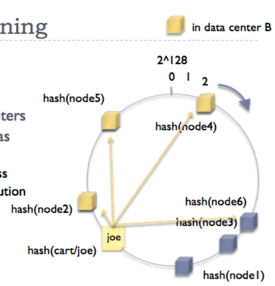
Two virtual nodes per each physical node

How to use consistent hashing

- Try 1: randomly assign nodes numbers (tokens), keys hashing between tokens assigned to next token
  - Problem: finding all keys in a region requires scan (no DB index!)
  - Adding more nodes forces repartitioning of data (a range gets split)
  - Hard to snapshot key space
- Try 2: fixed size buckets, random token numbers, buckets assigned to next token
  - not evenly assigned
- Try 3: Each bucket explicitly assigned to a node, node remembers list of buckets
  - More control over what buckets it steals, where buckets go (no longer rely on hash + numbers)
  - Can store each bucket in a separate file/db
  - Must store mapping

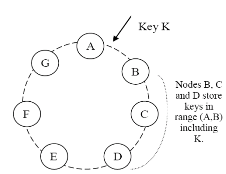
Dynamo: Partitioning

- Physical placement of replicas
  - Each key is replicated across multiple data centers
  - Arrangement scheme has not been revealed
    - Netmask is helpful, I guess
    - Impact on replica distribution is unknown
- Advantages
  - No data outage on data center failures



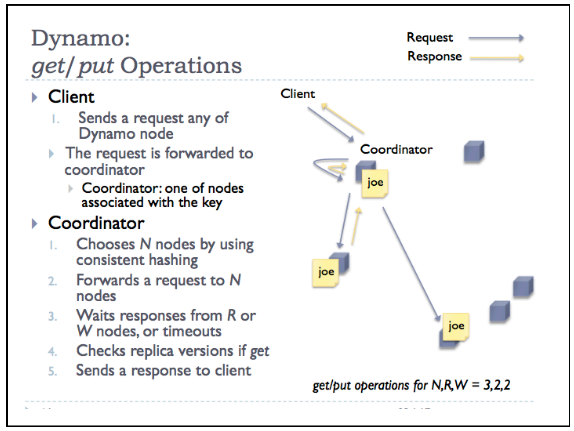
Replication

- Each data item is replicated at next N hosts.
- "*preference list*": The list of nodes that is responsible for storing a particular key.
  - The next N+slop nodes after key (slop for availability)
  - Virtual nodes are skipped to ensure that replicas are located on different physical nodes



Replication

- Each access has a coordinator
- The coordinator hashes the node at N other replicas
  - Anyone can coordinate a read
  - Writes must be done at any of top N nodes so it can assign a timestamp
- N replicas that are next to the coordinator node in the ring in the clockwise fashion



Consistency

- Dynamo client chooses consistency level
  - Chooses N (number of replicas), W (number of writes that must complete), and R (number of reads the at must complete)
  - w+r > N leads to strong consistency
- By default, replication is synchronous
  - async only under failure
- On read: collect results from r nodes

Data Versioning

- A put() call may return to its caller before the update has been applied at all the replicas
- **QUESTION: How do you handle inconsistency**?
  - Allow multiple versions to exist simultaneously
  - Means Dynamo doesn't need to merge versions
- A get() call may return many versions of the same object.
- Challenge: an object having distinct version sub-histories, which the system will need to reconcile in the future.
- Solution: uses vector clocks in order to capture causality between different versions of the same object.

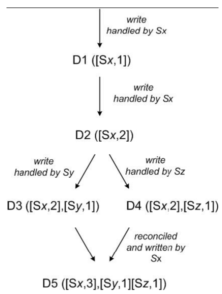
Detecting/resolving conflicts

- Reads return **causally indepdendnt** versions
  - If one version is just older (was overwritten), ignored
- Detect on READ when two conflicting versions come back
  - Fixed by merging in client, writing back new version with clock > all existing versions
- Old versions detected on read when non-conflicting versions returned
  - old one is updated

Vector Clock

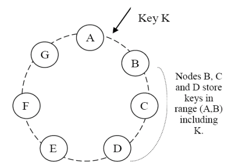
- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.
- *If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.*

Vector clock example



Failed nodes: Hinted handoff

- **What happens on failure?**
- Assume N = 3. When A is temporarily down or unreachable during a write, send replica to D.
  - Write to W "healthy nodes" for fault tolerance
- D is hinted that the replica is belong to A and it will deliver to A when A is recovered.
  - Doesn't just take over for temporarily failed node, but remembers which data to give back
- Again: "always writeable"



Handling Permanent failures

- What happens when hinted replicas are unavailable to the returning node?
  - Q: key problem is getting latest data back to it
  - Answer: use anti-entropy (random synchronization with peers)
- Challenge: detect which data out of date
  - Answer: Merkle trees

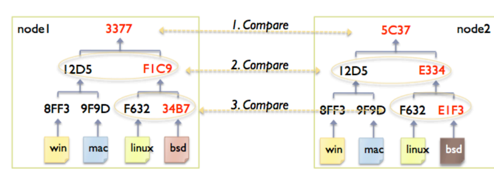
Replica synchronization (Cont'd)

- Structure of Merkle tree:
  - a hash tree where leaves are hashes of the values of individual keys.
  - Parent nodes higher in the tree are hashes of their respective children.
    - Solves the problem of finding small differences in a large data set

Replica synchronization (Cont'd)

- Advantage of Merkle tree:
  - Each branch of the tree can be checked independently without requiring nodes to download the entire tree.
  - Help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas.
- Advantages of Merkle tree
  - Comparisons can be reduced, if most of replicas are synchronized
    - Root checksums are equal, and no more comparison is required

- Sync replicas with Merkle tree
  - Compares Merkle tree with other nodes
    - From root to leaf, until checksum corresponds with each other



Recovery Issues

- What is increased load on system when coming back
  - talk to nodes ~ number of buckets, pull some data from each one
  - low impact on those nodes

Failure detection

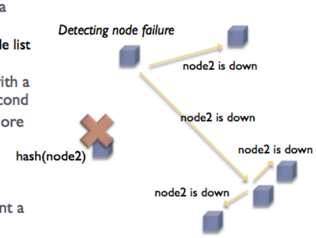
- What do Dynamo nodes have to know about each other?
  - Do they have to know which are live/dead?
  - Do they have to know which are members?
  - ANSWER: only membership + map
- Local failure detection
  - nodes only communicate with next/previous nodes on ring, so can learn during regular communication if they are alive/dead

Membership

- Node add/remove handled through "gossip" protocol
  - nodes randomly communicate with each other
  - Does "anti entropy"
  - New node chooses tokens on ring
    - At first, a node only knows its tokens
  - Anti-entropy distributes this & other mappings to everyone

Dynamo: Membership

- Gossip-based protocol
  - Spreads membership like a rumor
    - Membership contains node list and change history
  - Exchanges membership with a node at random every second
  - Updates membership if more recent one received
- Advantages
  - Robust; no one can prevent a rumor from spreading
  - Exponentially rapid spread



Membership change is spread by the form of gossip

Summary of techniques used in *Dynamo* and their advantages

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.