COMP5338 – Advanced Data Models

Week 8: Key Value Storage Systems

Dr. Ying Zhou
School of Computer Sciences

Outline

- **Overview**
  - **K-V store**
  - **Memcached brief intro**
- **Dynamo**
  - **Overview**
  - **Partitioning Algorithm**
  - **Replication and Consistency**

Key-Value Data Model

- Simplest form of data model
  - Each data “record” contains a key and a value
    - All queries are key based
  - Similar concept in programming language/data structure
    - Associative array, hash map, hashtable, dictionary
  - Basic API similar to those in hashtable
    - `put key value`
    - value = **get** key
- There are many such systems
  - Some just provides pure K-V form
    - The system treats **`value`** as uninterpretable string/byte array
  - Others may add further dimensions in the value part
    - The value can be a json document, e.g HyperDex
    - The value can contain columns and corresponding values, e.g. Bigtable/HBase

From Keys to Hashes

- In a key-value store, or key-value data structure, the key can be of any type
  - String, Number, Date and Time, Complex object,
- They are usually hashed to get a value of fixed size
  - Hash function is any function that can be used to map data of arbitrary size to data of a fixed size
    - E.g. any Wikipedia page’s content can be hashed by SHA-1 algorithm to produce a message-digest of 160-bit value
  - The result is called a hash value, hash code, hash sum or hash
- Hash allows for efficient storage and lookup

key	Hash
Alice	1
Tom	3
Bob	4

Hash Entry	Data
1	(Alice, 90)
3	(Tom, 85)
4	(Bob, 87)

Memcached: motivation

- **`Memcached`** is a very early in-memory key-value store
- The main use case is to provide caching for web application, in particular, caching between the database and application layer
- Motivations:
  - Database queries are expensive, cache is necessary
  - Cache on DB nodes would make it easy to share query results among various requests
    - But the raw memory size is limited
  - Caching more on Web nodes would be a natural extension but requests by default have their own memory spaces and do not share with each other
- Simple solution
  - “Have a **global hash table** that all Web processes on all machines could access simultaneously, instantly seeing one another’s changes”

https://www.linuxjournal.com/article/7451

Memcached: features

- As a cache system for query results
  - Durability is not part of the consideration
    - Data is persisted in the database
    - No replication is implemented
  - Mechanisms for freshness guarantee should be included
    - Expiration mechanism
  - Cache miss is a norm
    - But want to minimize that
- Distributed cache store
  - Utilizing free memories on all machines
  - Each machine may run one or many Memcached server instances
  - It is beneficial to run multiple Memcached instances on single machine if the physical memory is larger than the address space supported by the OS

Memcached: the distributed caching

- The keys of the global hash table are distributed among a number of **`Memcached`** server instances
  - How do we know the location of a particular key
- Simple solution
  - Each **`memcached`** instance is totally independent
  - A given key is kept in the same server
  - Client library knows the list of servers and can use a predetermined partition algorithm to work out the designated server of a key
- Client library runs on web node
- There are many implementations
- May have different ways to partition keys
  - Simple hash partition or consistent hashing

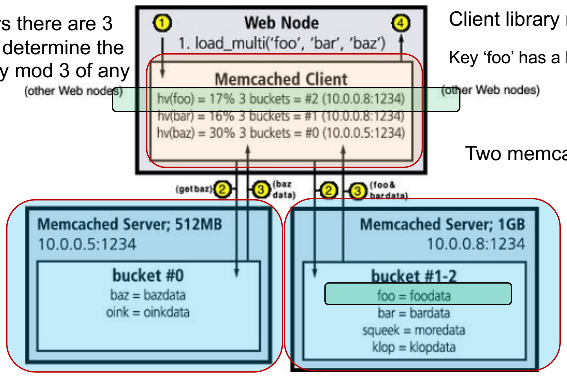
Memcached: basic hash partition

- Treat **`Memcached`** instance as buckets
  - Instance with larger memory may represent more buckets
- Use modulo function to determine the location of each key
  - Hash(key) **mod** #buckets

Client library knows there are 3 buckets, so would determine the location of a key by mod 3 of any key’s hash value

Client library runs on web node

Key ‘foo’ has a hash value of 17, mod 3 is 2



Two memcached servers

One server has 512M memory for Memcached, and is considered as one bucket with id 0

Another server has 1G memory for Memcached, and is considered as two bucket with id 1 and 2

This key should be stored on the server managed bucket id 2

Outline

- Overview
  - K-V store
  - Memcached brief intro
- **Dynamo**
  - **Overview**
  - Partitioning Algorithm
  - Replication and Consistency

Dynamo

- Motivation
  - Many services in Amazon _only need primary key access_ to data store
    - E.g. shopping cart
  - Both scalability and availability are essential terms in the service level agreement
    - Always writable (write never fails)
    - Guaranteed latency
    - Highly available
- Design consideration
  - Simple key value model
  - Sacrifice strong consistency for availability
  - Conflict resolution is executed during **read** instead of write
  - Incremental scalability

Dynamo Techniques Summary

- Dynamo is a decentralized **peer-to-peer** system
  - All nodes taking the same role
  - There is no master node

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.

Outline

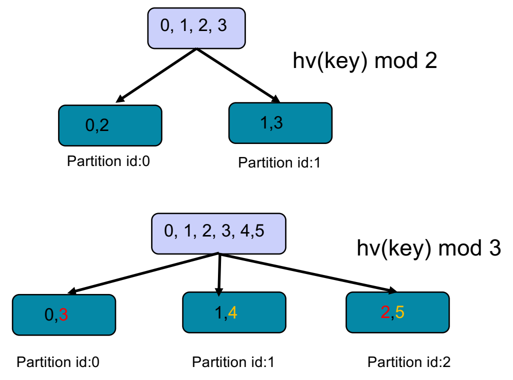
- Overview
  - K-V store
  - Memcached brief intro
- **Dynamo**
  - Overview
  - **Partitioning Algorithm**
  - Replication and Consistency

Partitioning Algorithm

- Partition or shard a data set
  - There is a partition or shard key
  - There is an algorithm to decide which data goes to which partition
    - Range partition vs. Random(Hash) partition
- Range partition requires keys to be of same type
  - So we can define a range of keys for each partition
- Hash partition is more flexible
  - Key distribution is based on its hash value instead of raw value
  - There are many options to determine the key placement
    - Modulo function as in previous section
    - Predefined ranges on hash value as in MongoDB hashed sharding
    - Consistent hashing in Dynamo

Modulo Based Hash Partition- Repartition

- Repartition may involve a lot of data movement
  - The modulo function of key's hash value will redistribute large number of keys to different partitions



Consistent Hashing

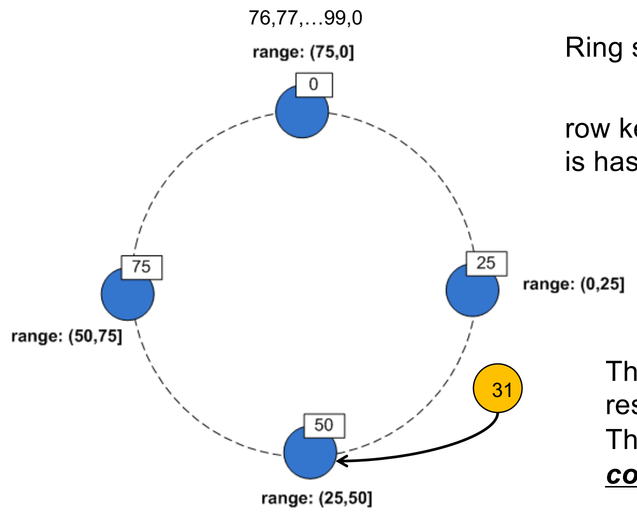
- Consistent hashing
  - " a special kind of hashing such that when a hash table is resized, only **K/n** keys need to be remapped on average, where **K** is the number of keys, and **n** is the number of slots."

    [Wikipedia: Consistent Hashing]
  - It does not identify each partition as a number in **[0,n-1]**
  - The output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value).
  - Each partition represents a range in the ring space, identified by its position value (token)
  - The hash of a data record's key will uniquely locate in a range
  - In a distributed system, each node represents one partition or a number of partitions if "virtual node" is used.

Consistent Hashing in Dynamo

- Each node in the Dynamo cluster is assigned a "token" representing its position in the "ring"
- Each node is responsible for the region in the ring between it and its predecessor node
- The ring space is the MD5 Hash value space (128 bit)
  - 0 to 2^{127} -1
- The MD5 Hash of the key of any data record is used to determine which node is the coordinator of this key. The coordinator is responsible for
  - Storing the row data locally
  - Replicating the row data in N-1 other nodes, where N is the replication factor

Consistent hashing example



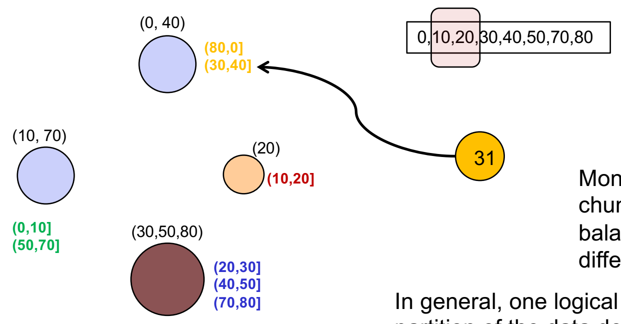
Ring space: 0~99

row key: “bsanderson”
is hashed to a number **31**

The node with token 50 is responsible for this key
This is called the ***coordinator*** of this key

Virtual Nodes

- Possible issue of the basic consistent hashing algorithm
  - Position is randomly assigned, cannot guarantee balanced distribution of data on node
  - Assume nodes have similar capacity, each is handling one partition
- Virtual nodes and multiple partitions per node

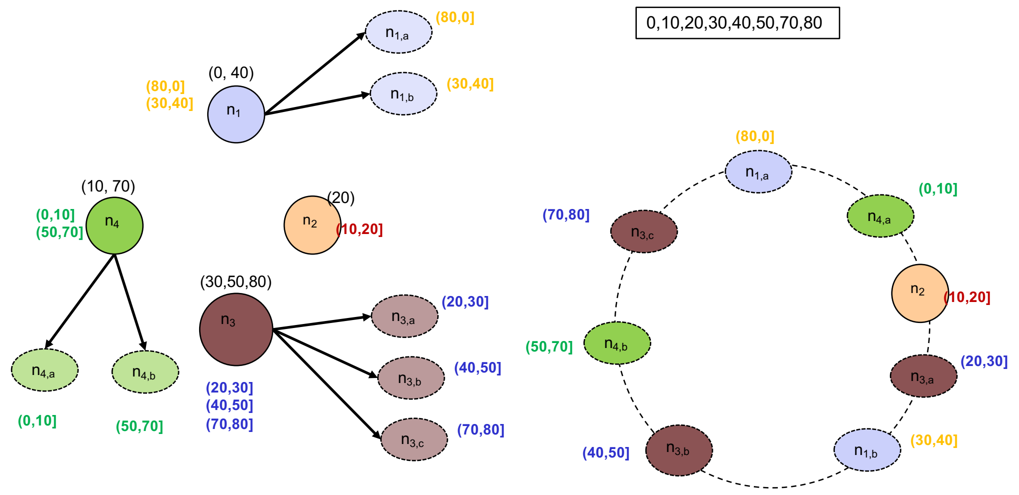


Memcached has similar design consideration: an instance with a lot of available memories can be allocated more than one buckets

MongoDB uses smaller sized chunk to achieve shard load balance. Shards may contain different number of chunks

In general, one logical partition of the data does not map to one physical machine

Virtual Nodes Ring

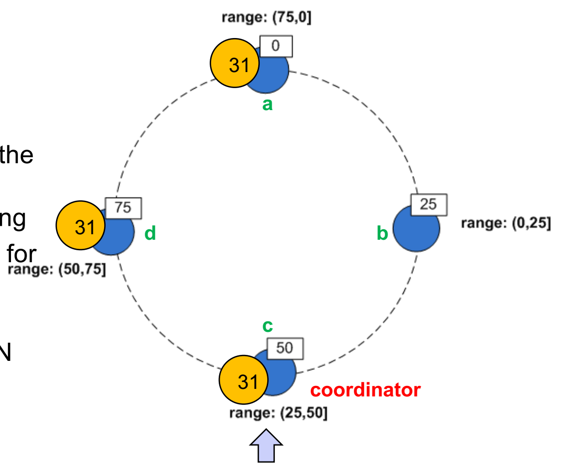


Outline

- **Overview**
  - **K-V store**
  - **Memcached brief intro**
- **Dynamo**
  - **Overview**
  - **Partitioning Algorithm**
  - **Replication and Consistency**

Replication

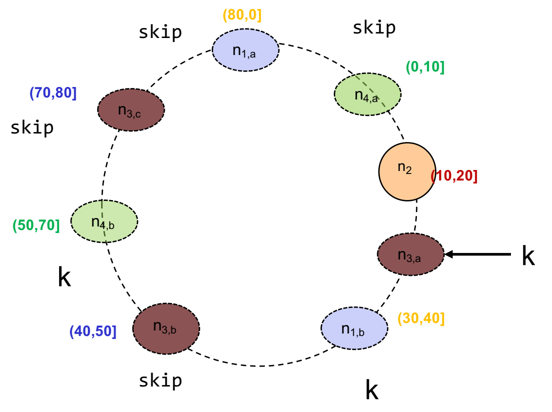
- Replication is essential for high availability and durability
  - Replication factor (N)
  - Coordinator
  - Preference list
- Each key (and its data) is stored in the coordinator node as well as N-1 clockwise successor nodes in the ring
- The list of nodes that is responsible for storing a particular key is called the *preference list*
- Preference list contains more than N nodes to allow for node failures
  - Some node are used as temporary storage.
  - Can be computed on the fly by any node in the system



Preference list for this key: {c,d,a,b}

Replication with Virtual Nodes

- "Note that with the use of virtual nodes, it is possible that the first N successor positions for a particular key may be owned by less than N distinct physical nodes (i.e. a node may hold more than one of the first N positions). To address this, the preference list for a key is constructed by skipping positions in the ring to ensure that the list contains only distinct physical nodes."

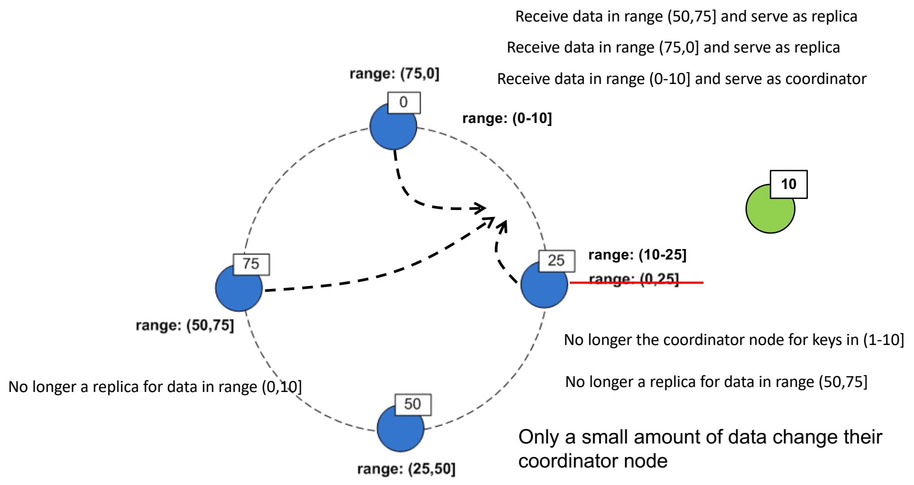


Preference list of K is: {n3, n1,n4, *n2*}

Membership and Failure Detection

- Each node in the cluster is aware of the token range handled by its peers
  - This is done using a gossip protocol
  - New node joining the cluster will randomly pick a token
  - The information is gossiped around the cluster
- Failure detection is also achieved through gossip protocol
  - Local knowledge
  - Nodes do not have to agree on whether or not a node is "really dead".
  - Used to handle temporary node failure to avoid communication cost during read/write
  - Permanent node departure is handled externally

Adding Storage Node


Receive data in range (50,75] and serve as replica
Receive data in range (75,0] and serve as replica
Receive data in range (0-10] and serve as coordinator
range: (75,0]
0
range: (0-10]
10
75
25
range: (10-25]
~~range: (0,25]~~
range: (50,75]
No longer the coordinator node for keys in (1-10]
No longer a replica for data in range (0,10]
No longer a replica for data in range (50,75]
50
range: (25,50]

Only a small amount of data change their coordinator node

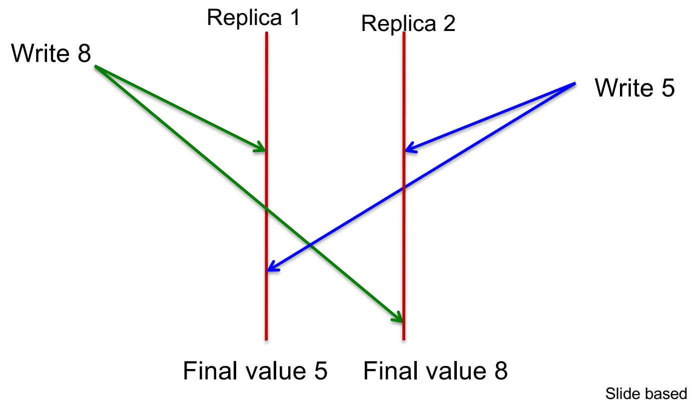
http://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2

Read and Write with Replication

- When there are replicas, there are many options for read/write
- In a typical Master/Slave replication environment
  - Write happens on the master and may propagate to the replica immediately and wait for all to ack before declaring success, or lazily and declare success after the master finishes write
  - Read may happen on the master (strong consistency) or at one replica (may get stale data)
- In an environment where there is no designated master/coordinator, other mechanisms need to be used to ensure certain level of consistency
  - Order of concurrent writes
  - How many replica to contact before answering/acknowledging

Concurrent Write

- Different clients may try to update an item simultaneously
- If nothing special is done, system could end with "split-brain"

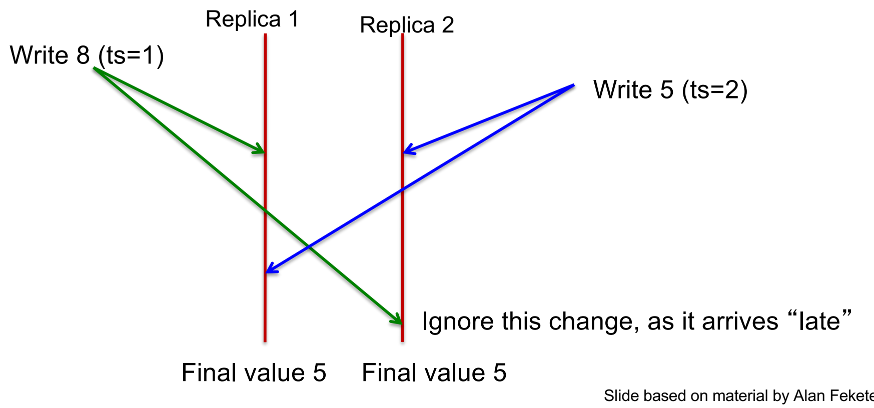


Timestamp is a typical way to order concurrent writes

Slide based on material by Alan Fekete

Ordering concurrent writes

- Associate timestamps on the writes, and let higher timestamp win
  - Node ignores a write whose timestamp is lower than the value already there (Thomas Write Rule)



Slide based on material by Alan Fekete

Quorums

- Suppose each write is *initially* done to W replicas (out of N)
  - Other replicas will be updated later, after write has been acked to the client
- How can we find the current value of the item when reading?
- Traditional "quorum" approach is to look at R replicas
  - Consider the timestamp of value in each of these
  - Choose value with highest timestamp as result of the read
- If W>N/2 and R+W>N, this works properly
  - any write and read will have at least one site in common (quorums intersect)
- Any read will see the most recent completed write,
  - There will be at least one replica that is BOTH among the W written and among the R read

Slide based on material by Alan Fekete

Dynamo Mechanisms

- Vector Clock
  - A way to implement the "timestamp" concept in distributed system
  - Aid for resolving version conflict
    - E.g. when read receives R copies, how do we decide if there is causal order among the copies or if they are on different branches?
- Sloppy Quorum + hinted hand off
  - Achieve the "always writable" feature
  - Eventual consistency

Dynamo Read/Write Route

- *Any node* is eligible to receive client read/write request
  - get (key) or put (key, value)
- The node receives the request can direct the request to the node that has the data and is available
  - Any node knows the token of other nodes
  - Any node can compute the hash value of the key in the request and the preference list
  - A node has local knowledge of node failure
- In Dynamo, "A node handling a read or write operation is known as the coordinator. Typically, this is the first among the top N nodes in the preference list.", which is usually the coordinator of that key unless that node is not available.
  - Every node can be the coordinator of some operation
  - For a given key, the read/write is usually handled by its coordinator or one of the other top N nodes in the preference list

Data Versioning

- A **put()** call may return to its caller before the update has been applied at all the replicas
  - $W < N$
- A **get()** call may return many versions of the same object/value.
  - $R > 1$ and $R < N$
- Typically when $N = 3$, we may have $W = 2$ and $R = 2$
- Challenge: an object may have 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.

Vector Clock: definition

- "A vector clock is effectively a list of **(node, counter)** pairs. One vector clock is associated with every *version* of every *object*."
  - E.g. $[(n_0,1),(n_1,1)]$, $[(n_1,2),(n_2,1),(n_3,2)]$
  - *Node* refers to the node that coordinate the write
  - *Counter* remembers how many times this node has coordinated the write
- $\mathbf{[(n_0,1),(n_1,1)]}$ means the associated version of the object has been updated twice: once coordinated by $\mathbf{n_0}$ and once coordinated by $\mathbf{n_1}$;
- $\mathbf{[(n_1,2),(n_2,1),(n_3,2)]}$ means the associated version of the object has been updated 5 times coordinated by three different nodes;
- Coordinator variation is caused by temporary node failure
- Vector clock is not designed for deriving the coordination history of a single version.
- It is designed to determine causal ordering between a pair of clocks belonging to two versions of the same object.

Vector Clock: causal ordering

- "One can determine whether two versions of an object are on *parallel branches* or have a *causal ordering*, by examine their vector clocks. 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. Otherwise, the two changes are considered to be in conflict and require reconciliation. "
  - [(n0,1),(n1,1)] **<** [(n0,2),(n1,1),(n3,**1**)]
    - The second version is ~~3~~ 2 updates ahead of the first one
  - [(n0,1),(n1,1)] ?? [(n0,2),(n2,**1**)]
    - The two versions branch off after the initial insert with [(n0,1)],
      - the left version is then updated once, and this is coordinated by n1
      - the right version is updated twice: once by n2 and once by n0; we don't know which happens first

Update with vector clock

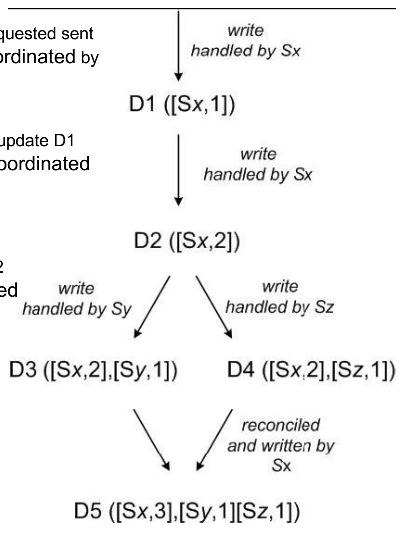
- Data replication revisits
  - Each replica stores a copy of an object
  - The copy stored on different replica may be of different version, as indicated by its vector clock
  - This happens because any machine hosting the replica may be temporarily unavailable and miss some update requests
  - Both read/write may contact multiple replica and obtain multiple versions
- Update with vector clock mechanism
  - "In Dynamo, when a client wishes to update an object, it must *specify which version it is updating*. This is done by passing the context it obtained from an *earlier read operation*, which contains the vector clock information. Upon processing a read request, *if Dynamo has access to multiple branches that cannot be syntactically reconciled, it will return all the objects at the leaves, with the corresponding version information in the context*. An update using this context is considered to have reconciled the divergent versions and the branches are collapsed into a single new version."

Vector Clock Example

An insert requested sent by ***c1*** is coordinated by Sx

c1 wants to update D1 and this is coordinated by Sx

c1 wants to update D2 and this is coordinated by Sy



c2 reads the object, and if it receives both D1 and D2, it can tell the replica with D1 that it should be garbage collected. And D2 is the latest version.

c2 wants to update D2 and this is coordinated by Sz. It is not aware of D3

C3 wants to update the object, when it performs a read, it may receive both D3 and D4; **both** will be included in the context. Assume the write is handled by Sx again. It will include all previous in its vector clock. So it is the descendant of both D3 and D4.

Vector Clock Size

- The size of vector clock is relatively small
  - Only the node coordinates the write request has an entry in the vector clock
  - If the preference list contains 4 nodes, most vector clocks contain up to 4 entries
  - In rare situation when all nodes in the preference list is not available, another node will coordinate the write and add an entry in the vector clock
  - A vector clock truncate mechanism is used to remove the oldest entry
    - Additional timestamp required to determine the 'oldest entry'
  - Truncating may remove histories that needed in reconciliation
    - It has never occurred in real life so is not explored extensively

Vector Clock More Examples

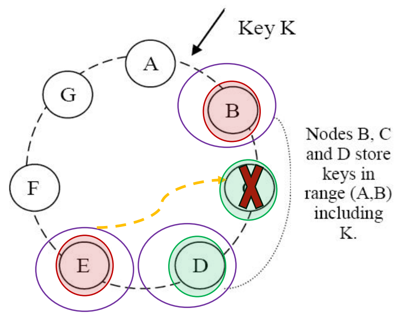
- In a system with 6 nodes: n0~n5, for a key with preference list {n1~n4} which of the following vector clock pair(s) has(have) causal order:
  - [(n1, **1**), (n2,**2**)] and [(n1,**1**)]
  - [(n1, **3**), (n3,**1**)] and [(n1,**2**),(n2,**1**)]
  - [(n1,**2**),(n3,**1**)] and [(n1,**4**),(n2,**1**),(n3,**1**)]

Sloppy Quorum

- Quorum members may include nodes that do not store a replica
  - Preference list is larger than N
  - Read/Write may have quorum members that do not overlap
- Both read and write will contact the first N *healthy* nodes from the preference list and wait for **R** or **W** responses
- Write operation will use hinted handoff mechanism if the node contacted does not store a replica of the data

Hinted Handoff

- Assume N = 3. When C is temporarily down or unreachable during a write, send replica to E.
- E is hinted that the replica belongs to C and it should deliver to C when C is recovered.
- Sloppy quorum does not guarantee that read can always return the latest value
  - Write set: (<u>**B**</u>,D,<u>**E**</u>)
  - Read set: (<u>**C**</u>,<u>**D,**</u> E)



Write contacts nodes B,D,E and acks to client after B and E reply

Read contacts nodes C,D,E and replies the client a merged value based on replies from C and D

References

- Brad Fitzpatrick, “Distributed Caching with Memcached” Linux Journal” [Online]. August 1, 2004. Available: https://www.linuxjournal.com/article/7451.

- Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. **Dynamo: amazon's highly available key-value store**. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (SOSP '07). 205-220.