PART 3.
DATA STORAGE AND FLOW MANAGEMENT

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435

Today’s topics

• FAQs
• BigTable: Column-based storage system

FAQs

• PA3 help session
  • April 7th, 3:00PM ~ 3:50PM CSB130

Data Storage and flow management
Column Family Stores
Google Big Table

This material is built based on,


Topics in BigTable
1. Data model
2. Locating tablet
3. Data Compaction
4. Data Compression
5. Caching and prefetching
Column Family Stores: Big Table

2. Locating Tablet

System Structure

BigTable master
Performs metadata ops + load balancing

BigTable tablet server
Serves data

BigTable tablet server
Serves data

BigTable tablet server
Serves data

Cluster scheduling system
Handles failover, monitoring

GFS
Holds tablet data, logs

Lock service
Holds metadata, handles master-collection

Building blocks (1/2)

- Memtable: in-memory table
  - writes go to log then to in-memory table
  - Periodically data are moved from memory table to disk (using SSTable file format)

- The Google SSTable (Sorted String Table) file format
  - Internally used to store the contents of a part of table (Tablet)
  - Persistently ordered immutable map from key to values
  - Keys and values are arbitrary byte strings

- Tablet
  - All of the SSTables for one key range + memtable

Building blocks (2/2)

- SSTable contains a sequence of blocks
  - 64KB, configurable

- Block index
  - Stored at the end of SSTable
  - Index is loaded into memory when the SSTable is opened

- SSTable is used by: Cassandra, Hbase, LevelDB
  - Open-source implementation
  - http://code.google.com/p/leveldb/

SSTable: Sorted String Table

Reading and writing data can dominate running time
Random reads and writes are critical features

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Key</th>
<th>Value</th>
<th>...</th>
</tr>
</thead>
</table>

Index

<table>
<thead>
<tr>
<th>Key</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>Offset</td>
</tr>
</tbody>
</table>

Access to the block

- In-memory map of keys to {SSTables, memtable}

- Lookup can be performed with a single disk seek
  - Find the block by performing a binary search of the in-memory index
  - Read the block from disk
Locating tablets [1/2]

- Since tablets move around from server to server, given a row, how do clients find the right machine?
  - Need to find tablet whose row range covers the target row

Using the BigTable master

- Central server almost certainly would be bottleneck in large system
  - Instead: store special tables containing tablet location info in BigTable cell itself

Locating tablets [2/2]

- 3-level hierarchical lookup scheme for tablets
  - Location is part of relevant server
  - 1st level: bootstrapped from Chubby (lock service), points to the root tablet
  - 2nd level: Uses root tablet data to find owner(node) of appropriate metadata tablets
  - 3rd level: metadata table holds locations of tablets of all other tables
    - Metadata tablet itself can be split into multiple tablets
  - Aggressive prefetching+caching
    - Most ops go right to proper machine

Caching the tablet locations [1/4]

- Root tablet is never split
  - To ensure that the tablet location hierarchy has no more than 3 levels

Metadata tablet

- Stores the location of a tablet under a row key
  - Tablet’s identifier and its end row
  - Each metadata row stores approximately 1KB of data in memory
  - Average limit of 128MB Metadata tablets
  - 2nd tablets are addressed

Caching the tablet locations [2/4]

- If the client’s cache is empty?
  - One read from Chubby
  - One read from root tablet
  - One read from metadata tablet
  - Three network round-trips is required to locate the tablet

Caching the tablet locations [3/4]

- If the client’s cache is stale?
  - With given information, client could not find the data
  - What is the maximum round-trips needed (If the root server has not changed)?
Caching the tablet server locations

- If the client's cache is stale?
  - With given information, client could not find the data
- Up to 5 round trips (if the root server has not changed)
  - First round: user accesses tablet and misses data (arrow 1)
  - If only the tablet information is stale
    - 2 additional rounds to locate tablet info from the metadata tables (a-1, a-2)
  - If the metadata table info is also stale
    - 4 additional rounds
      - To the metadata table (it misses tablet info due to the stale info) (b-1)
      - To the root server to retrieve the location of the metadata table (b-2)
      - To the metadata table to retrieve the tablet server location(b-3)
      - Locate tablet from the tablet server(b-4)

Prefetching tablet locations

- Client library reads the metadata for more than one tablet
  - Whenever it reads the metadata table
- No GFS accesses are required
  - Table locations are stored in memory

Tablet Assignment (1/2)

- Each tablet is assigned to one tablet server at a time
  - The master keeps track of:
    - The set of live tablet servers
    - Which tablets are assigned
- New tablet assignment
  - The master assigns the tablet by sending a tablet load request to the tablet server

Tablet Assignment (2/2)

- A tablet server starts
  - Chubby creates a uniquely-named file in a specific Chubby directory
  - Exclusive lock
  - Master monitors this directory to discover tablet servers
- A tablet server terminates
  - Release its lock
  - Master will reassign its tablets more quickly

Tablet status

- The persistent state of a tablet is stored in GFS

Tablet Representation

- SSTable on GFS
- Append-only log on GFS
- Write buffer in memory (random-access)
**write operation**

- The tablet server checks,
  - If the data is well-formed
  - If the user is authorized to mutate data
- Operation is committed to a log file
- The contents are inserted into the MemTable

**read operation**

- Tablet server checks
  - If the request is well-formed
  - If the user is authorized to read data
- Merged view of MemTable (in memory) and SSTable (in disk)
- Read operation is performed

**Data Compaction and Compression**

- What is the difference between data compaction and data compression?

**Minor Compactions**

- As write operations executed
  - The size of the memtable increases
- Minor compaction
  - When the memtable size reaches a threshold
    - The memtable is frozen
    - A new memtable is created
    - A frozen memtable is converted to an SSTable (stored in GFS)
  - Shrinks the memory usage in the tablet server
  - Reduces the amount of data that has to be read from the commit log during recovery (if the server dies)

**Merging Compaction**

- New SSTable from the minor compaction will increase
  - Read operations need to merge updates from large number of SSTables
- Merging Compaction
  - Bounds the number of such files periodically
  - Reads the contents of a few SSTables and the memtable and writes out a new SSTable
  - Input SSTables and memtable can be discarded as soon as the merging compaction has finished
Major Compaction
- Rewrites multiple SSTables into exactly one SSTable
- No deletion information or deleted data included

Data Compaction:
Log-Structured Merge (LSM) Trees

Background
- Sequential access to disk (magnetic or SSD) is at least three orders of magnitude faster than random IO
  - Journaling, logging or a heap file is fully sequential
  - 200-300 MB/s per drive
- But transitional logs are only really applicable to “SIMPLE” workloads
  - Data is accessed entirely
  - Data is accessed by a known offset

Sequential IO vs. Random IO

Existing approaches to improve performance
- Hash
- B+ tree
- External file: create separate hash or tree index

- Adding index structure improves read performance
  - It will slow down write performance
  - Update structure and index

- Log-structured merge trees
  - Fully disk-centric
  - Small memory footprint
  - Improved write performance
  - Read performance is still slightly poorer than B+ tree
Basic idea of LSM trees

- LSM trees manage batches of writes to be saved
- Each file contains a batch of changes covering a short period of time
- Each file is sorted before it is written
- Files are immutable
- New updates will create new files
- Reads inspect all files
- Periodically files are merged

In-memory buffer for LSM (MemTable)

- Data is stored as a tree (Red-Black, B-tree etc) to preserve key-ordering
- MemTable is replicated on disk as a write-ahead-log
- When the MemTable fills the sorted data is flushed to a new file on disk
- Only sequential IO is performed
- Each file represents a small, chronological subset of changes (sorted)
- Periodically the system performs a compaction

Conceptual view of rolling merge

Locality groups

- Clients can group multiple column families together into a locality group
  - Separate SSTable is generated for each locality group in each tablet

  Example
  - Locality group 1: Page metadata in Webtable
    - Language and checksum
  - Locality group 2: Contents of the page
  - Application reading the metadata does not need to read through all of the page content

Compression

- Compression is required for the data stored in BigTable
  - Similar values in the same row/column
  - With different timestamps
  - Similar values in different columns
  - Similar values across adjacent rows

- Clients can control whether or not the SSTables for a locality group are compressed
  - User specifies the locality group to be compressed and the compression scheme
  - Keep blocks small for random access (~64KB compressed data)
  - Low CPU cost for encoding/decoding
  - Server does not need to encode/decode entire table to access a portion of it
Two-pass compression scheme

- Data to be compressed
  - Keys in BigTable (row, column and timestamp)
    - Sorted strings
  - Values in BigTable
    - BMDiff (Bentley and McIlroy’s Scheme) across all values in one family
    - BMDiff output for values 1..N is dictionary for value N+1
  - Zippy is used for final pass over whole block
    - Localized repetitions
    - Cross-column-family repetition, compresses keys
  - First pass: BMDiff
  - Second pass: Zippy (now called as snappy)

BMDiff

- Adapted to VCDiff (RFC3284)
  - Shared Dictionary Compression over HTTP (SDCH)
  - Chrome browser

Example of the Constitution of the US and the King James Bible

<table>
<thead>
<tr>
<th>File</th>
<th>Text</th>
<th>gzip size</th>
<th>Relative compressed size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>49523</td>
<td>13936</td>
<td>1.0</td>
</tr>
<tr>
<td>Const+Const</td>
<td>99046</td>
<td>26631</td>
<td>1.911</td>
</tr>
<tr>
<td>Bible</td>
<td>4480056</td>
<td>1321495</td>
<td>1.0</td>
</tr>
<tr>
<td>Bible+Bible</td>
<td>8920112</td>
<td>2642389</td>
<td>1.9995</td>
</tr>
</tbody>
</table>

http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=755678&isnumber=16375

The compression algorithm

- Representing the common string
  - start, length
    - start: initial position
    - length: size of the common sequence
- e.g. “the Constitution of the United States PREAMBLE We, the people of the United States, in order to form a more perfect Union…”
  - “the Constitution of the United States PREAMBLE We, the people <16.21>, in order to form a more perfect Union…”

Implementation of Data compression

- Representing the common string
  - Hash table of fingertable
  - Text
  - Calculate hash value
  - Lookup the hash table
  - If there is no match, this value is added
What if we find a match?

- \( b = 100 \)
- The current block of length \( b \) matches block 56
- We could encode that single block as \( <5600, 100> \)
- This scheme guarantees not to encode any common sequences less than \( b \)

Results

<table>
<thead>
<tr>
<th>Compression</th>
<th>Bible</th>
<th>Bible+Bible</th>
</tr>
</thead>
<tbody>
<tr>
<td>input gzip</td>
<td>4460556</td>
<td>9029112</td>
</tr>
<tr>
<td>com50</td>
<td>1321495</td>
<td>2642389</td>
</tr>
<tr>
<td>com20</td>
<td>4384403</td>
<td>4384414</td>
</tr>
<tr>
<td>com50</td>
<td>gzip</td>
<td>3906771</td>
</tr>
<tr>
<td>com20</td>
<td>gzip</td>
<td>1318687</td>
</tr>
</tbody>
</table>

Snappy

- Based on LZ77
  - Dictionary coders
  - Sliding window
- Very fast and stable but not high compression ratio
  - 20~100% lower compression ratio than gzip

BigTable and data compressions

- Large window data compression
  - BMDiff (~ 100MB/s for write, ~1000MB/sec for read)
  - Identify large amounts of shared boilerplate in pages from same host

- Small window data compression
  - Looks for repetitions in 16KB window
  - Snappy
  - e.g. 45.1TB of crawled dataset (2.1B pages)
  - 4.2 TB compressed size

Caching for read performance

- Tablet servers use two levels of caching
  - Scan cache
    - Higher-level cache
      - Caches the key-value pairs returned by the SSTable interface in the table server
  - Block cache
    - Lower-level cache
      - Caches SSTables blocks that were read from GFS
Bloom filters

- Read operation has to read from all SSTables that make up the state of a tablet
  - SSTables in disk results many disk accesses
- Bloom filter
  - Detects if an SSTable might contain any data for a specified row/column pair
- Probabilistic data structure
  - Tests whether the element is a member of a set
  - The element either definitely is not in the set or may be in the set

Key Value Stores: Dynamo

This material is built based on,


What Amazon needs (1/2)

- Amazon’s architecture
  - A highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services
- Storage technologies that are always available
- Customer should be able to view and add items to the shopping cart even if:
  - The disks are failing
  - Network routes are flapping or,
  - Data centers are being destroyed by tornados

If you design a data storage system for,

- Amazon.com to store transaction data for the shopping cart management, how would you prioritize properties: Consistency, Availability, or Partition tolerance? And Why?

What Amazon needs

- Highly available system with failure resilience
  - Small but significant number of servers
  - Network components
- Failure handling should not impact availability or performance
Overview of Dynamo (1/2)
- Partitions and replicates data using consistent hashing
- Tracks object version to provide consistency
- Uses a decentralized synchronization protocol
  - Storage nodes can be added and removed from Dynamo without any manual partitioning or redistribution
- Gossip-based distributed failure detection and membership protocol

Overview of Dynamo (2/2)
- Underlying storage technology
  - Amazon’s e-commerce platform
- Handles peak loads,
  - Over 3 million checkouts in a single day
  - Hundreds of thousands of concurrently active sessions

Why Amazon does not store their state in relational database?
- Functionality of RDBMS is not required for Amazon’s tasks
  - Store and retrieve data by primary key
  - Does not require complex querying and management
- Excess functionality requires
  - Expensive hardware
  - Highly skilled personnel
- Hard to achieve scale-out databases

System Assumptions (1/2)
- Query model
  - Simple read and write operations to a data item
  - Uniquely identified by a key
  - Usually less than 1MB
  - No operations span multiple data items
- ACID Properties
  - Dynamo targets applications that operate with weaker consistency if this results in high availability
  - Dynamo does not provide any isolation guarantees

System Assumptions (2/2)
- Efficiency
  - The system needs to function on a commodity hardware infrastructure
  - Stringent latency requirements

Summary of techniques
<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Partitioning</td>
<td>Consistent Hashing</td>
</tr>
<tr>
<td>(2) High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
</tr>
<tr>
<td>(3) Handling temporary failures</td>
<td>Stoppable Quorum and hinted handoff</td>
</tr>
<tr>
<td>(4) Recovering from permanent failures</td>
<td>Anti-entropy using Merkle tree</td>
</tr>
<tr>
<td>(5) Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection</td>
</tr>
</tbody>
</table>
Key Value Stores: Dynamo
(1) Partitioning
(2) High Availability for writes
(3) Handling temporary failures
(4) Recovering from permanent failures
(5) Membership and failure detection

Partitioning algorithm
- Dynamically partitions the data over the set of nodes
- Distributes the load across multiple storage hosts
- Dynamo uses consistent hashing
  - Relies on the Chord Distributed Hash Table

Hash function
- What is a hashtable?
  - Hash function
    - Takes a hash-key value as an argument
    - Produces a bucket number as a result
  - Bucket number
    - $0 \sim (B-1)$, where $B$ is the number of buckets
  - Selecting prime number as the bucket number will reduce the probability of collisions

Hash function and indexing (1/2)
- Index
  - Data structure that makes it efficient to retrieve objects given the value of one or more elements of those objects
  - Index can be one of the fields
    - (name, address, phone) triples and index on the phone field
    - Given phone number, the index allows you to find the record quickly

Hash function and indexing (2/2)
- Hash function is useful to implement indexing
  - A hash function is applied to the value of the hash-key
    - One of the fields
  - Record is placed in the bucket whose number is determined by the hash function
    - The bucket could be a list of records in main-memory, or a disk block
What if hash-keys are not integers?
- String
  - Convert common types of integers using ASCII or Unicode equivalent
- Multi-dimension?
  - e.g. a geospatial coordinate?

Case study: Geohash
- Latitude/longitude geocode system
- Provides arbitrary precision
  - By selecting the length of the code gradually

Colorado State University
- 40.5748° N, 105.0810° W
- Geohash: 9jqqdqj5h3y1

Resolutions with length of geohash string
- Length: 0
  - 01
  - 00

Encoding (1/4)
- LAT: 40.5747652 LON:-105.0865006 (CSU)
- Phase 1. Create interleaved bit string geobits[]
  - Even bits are from longitude code, LON
  - Odd bits are from latitude code: LAT
- Step 1.
  - If -180<=LON<=0 set geobits[0] as 0
  - If 0<LON<=180 set geobits[0] as 1
- Step 2.
  - If -90<=LAT<=0 set geobits[1] as 0
  - If 0< LAT<90 set geobits[1] as 1
Encoding (2/4)

- LAT: 40.5747652 LON:-105.0865006 (CSU)

- Step 3.
  - Since geobits[0] = 0,
  - If -180<=LON<=-90 set geobits[2] as 0
  - If -90<LON<0 set geobits[2] as 1

- Step 4.
  - Since geobits[1] = 1
  - If 0<=LAT<=45 set geobits[3] as 0
  - If 45< LAT<90 set geobits[3] as 1

Encoding (3/4)

- LAT: 40.5747652 LON:-105.0865006 (CSU)

- Step 3.
  - Since geobits[2] = 0,
  - If -180<=LON<=-135 set geobits[4] as 0
  - If -135<LON<90 set geobits[4] as 1

- Step 4.
  - Since geobits[3] = 0
  - If 0<=LAT<=22.5 set geobits[5] as 0
  - If 22.5< LAT<45 set geobits[5] as 1

Encoding (4/4)

- LAT: 40.5747652 LON:-105.0865006 (CSU)

- Repeat this process until your precision requirements are met

- Currently geohash bit string is 010011

- Phase 2.
  - First five bits "00011" are mapped to "9"

Consistent Hashing

- When a hash table is resized
  - Non-consistent hashing algorithm requires to re-hash complete table
  - Consistent hashing algorithm requires only partial records of the table
Consistent hashing (2/3)

Identifiers: $2^m$ identifiers

- Machine B is the successor node of key 1: $\text{successor}(1) = 3$
- Key 2 will be stored in machine C: $\text{successor}(2) = 5$
- Key 3 will be stored in machine B: $\text{successor}(3) = 5$

Consistent hashing assigns keys to nodes:
- Key $k$ will be assigned to the first node whose identifier is equal to or follows $k$ in the identifier space.

Consistent hashing (3/3)

- If machine C leaves circle, $\text{successor}(5)$ will point to A.
- If machine N joins circle, $\text{successor}(2)$ will point to N.

Scalable Key location

- In consistent hashing:
  - Each node need only be aware of its successor node on the circle.
  - Queries can be passed around the circle via these successor pointers until it finds resource.
- What is the disadvantage of this scheme?
  - It may require traversing all $N$ nodes to find the appropriate mapping.

Scalable Key location in Chord

- Let $m$ be the number of bits in the key/node identifiers.
- Each node $n$ maintains:
  - A routing table with (at most) $m$ entries.
  - Called the finger table.
- The $i^{\text{th}}$ entry in the table at node $n$, contains the identity of the first node, $x$.
  - Succeeds $n$ by at least $2^{i-1}$ on the identifier circle.
  - I.e., $x = \text{successor}(n \times 2^{i-1})$, where $\text{successor}$ (and all arithmetic is modulo $2^m$).
- The $i^{\text{th}}$ entry finger of node $n$. 

Chord
Definition of variables for node \( n \), using \( m \)-bit identifiers

- \( \text{Finger}[i].\text{start} = (n+2^{i-1}) \mod 2^m \), \( 1 \leq i \leq m \)
- \( \text{Finger}[i].\text{interval} = [\text{finger}[i].\text{start}, \text{finger}[i+1].\text{start}) \)
- \( \text{Finger}[i].\text{node} = \text{first node} \geq n.\text{finger}[i].\text{start} \)

- Finger table
  - The Chord identifier
  - The IP address of the relevant node
- First finger of \( n \) is its immediate successor on the circle

Finger tables

Lookup process (1/3)
- Each node stores information about only a small number of other nodes
- A node’s finger table generally does not contain enough information to determine the successor of an arbitrary key \( k \)
- What happens when a node \( n \) does not know the successor of a key \( k \)?
  - If \( n \) finds a node whose ID is closer than its own to \( k \), that node will know more about the identifier circle in the region of \( k \) than \( n \) does

Lookup process (2/3)
- \( n \) searches its finger table for the node \( j \)
  - Whose ID most immediately precedes \( k \)
  - Ask \( j \) for the node it knows whose ID is closest to \( k \)
- Go clockwise
- Never overshoot

Lookup process (3/3)
Lookup process: example 1

0. Request comes into node (machine) 1 to find the successor of id 4.
1. Node 3 wants to find the successor of identifier 4
2. Identifier 4 belongs to [3,5)
3. Check succ: 3
4. Node 1 asks node 3 to find successor of 4
5. Successor of 4 is 0

Lookup process: example 2

0. Request comes into node 3.
1. Node 3 wants to find the successor of identifier 0
2. Identifier 0 belongs to [7,3)
3. Check succ: 0
4. Node 3 asks node 0 to find successor of 1
5. Machine is using identifier 0 as well -> succ is 0.

Dynamo’s partitioning

- Inspired by Consistent Hashing and Chord
- When a node starts for the first time
  - Chooses its set of tokens (virtual nodes in the consistent hash space)
  - Maps nodes to their respective token sets
  - Stores both tokens and nodes onto disk
- Repeated reconciliation of the membership change
- Partitioning and placement information are propagated via the gossip-based protocol
  - Token ranges handled by its peers
- Direct forwarding of read/write operations are possible

Replication (1/3)

- Dynamo replicates its data on multiple hosts
  - Each data item is replicated at R hosts, where R is a parameter configured “per-instance”
- Each key k is assigned to a coordinator node
  - The coordinator is managing the replication of the data items that fall within its range
  - Stores at the k-th clockwise successor nodes in the ring
  - Each node is responsible for the region of the ring between it and its R-th predecessor

(2) High Availability for writes

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Partitioning</td>
<td>Consistent Hashing</td>
</tr>
<tr>
<td>(2) High Availability for writes</td>
<td>Member checks with reconciliation during reads</td>
</tr>
<tr>
<td>(3) Handling temporary failures</td>
<td>Bi LinkedList and hot chain fallback</td>
</tr>
<tr>
<td>(4) Recovering from permanent failures</td>
<td>Anti-entropy using thread pool</td>
</tr>
<tr>
<td>(5) Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection</td>
</tr>
</tbody>
</table>
Replication (2/3)

Machine D will store the keys, \([A, B), [B, C),\) and \([C, D)\).

Replication (3/3)

- Preference list
  - The list of nodes that is responsible for storing a particular key
- Node failures
  - Preference list contains more than \(R\) nodes
- Virtual nodes can reduce actual number of machines in \(R\) nodes
  - The preference list for a key is constructed only by distinct physical nodes

Data Versioning

- Dynamo provides eventual consistency
  - Allows for updates to be propagated to all replicas asynchronously
  - A put call may return to its caller before the update has been applied to all the replicas

“Add to Cart” example

- The shopping cart application requires that an “Add to Cart” operation can never be forgotten or rejected
  - If the most recent cart is not available and user makes changes to an old version of the cart
    - Still the change should be meaningful and preserved
  - “Add to cart” and “delete item from cart” should be translated into put operation to Dynamo
  - The divergent versions are reconciled later

Maintaining vector clock

- Dynamo treats the result of each modification as a new and immutable version of the data
  - Multiple version of data can be in the system
- Version branching
  - Due to the failure(s) in node(s), there are conflicting versions of an object
- Merging
  - Collapses multiple branches of data evolution back into one
  - Semantic reconciliation
  - e.g. merging different versions of shopping cart and preserving all of the items those client put into the cart

Vector clocks

- Used to capture causality between different versions of same object
  - Two versions of object are on parallel branches or have a causal ordering
- Vector clock
  - A list of (node, counter) pairs
  - One vector clock is associated with every version of every object
Definition of the vector clocks

- \( VC(x) \) denotes the vector clock of event \( x \)
- \( VC(x)_z \) denotes the component of that clock for process \( z \)
- \( x \rightarrow y \) denotes that event \( x \) happens before event \( y \)
- If \( x \rightarrow y \), then \( VC(x) < VC(y) \)

\[
VC(x) < VC(y) \iff \forall z [VC(x)_z \leq VC(y)_z] \land \exists z' [VC(x)_z < VC(y)_{z'}]
\]

Examples

- \( VC(D1)=(Sx, 3), (Sy, 2), (Sz, 2), (Sq, 2) \)
- \( VC(D2)=(Sx, 3), (Sy, 2), (Sz, 2), (Sq, 1) \)
- \( VC(D3)=(Sx, 3), (Sy, 2), (Sq, 1) \)
- \( VC(D4)=(Sx, 3), (Sy, 3), (Sz, 2), (Sq, 1) \)

Properties of the vector clocks

- If \( VC(a) < VC(b) \), then \( a \rightarrow b \)
- Antisymmetry:
  - If \( VC(a) < VC(b) \) then \( NOT VC(b) < VC(a) \)
- Transitivity:
  - If \( VC(a) < VC(b) \) and \( VC(b) < VC(c) \), then \( VC(a) < VC(c) \)

Execution of get and put operations

- Users can send the operations to any storage node in Dynamo
- Coordinator
  - A node handling a read or write operation
  - The top \( N \) nodes in the preference list
- Client can select a coordinator
  - Route its request through a generic load balancer
  - Use a partition-aware client library
  - Directly access the coordinators

Using quorum-like system

- \( R \)
  - Minimum number of nodes that must participate in a successful read operation
- \( W \)
  - Minimum number of nodes that must participate in a successful write operation

- Setting \( R \) and \( W \)
  - \( R + W > N \)
  - \( W > N/2 \)

- The latency of a get (or put) operation is dictated by the slowest of the \( R \) (or \( W \)) replicas
  - \( R \) and \( W \) are configured to be less than \( N \)
put request
- Coordinator node
1. Generates the vector clock
   --For the new version
2. Writes the new version locally
3. Sends the new version to the N highest-ranked reachable nodes
   --Along with the new vector clock

get request
- The coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes
  - In the preference list
- Waits for R responses
- If multiple versions of the data are collected
  - Returns all the versions it deems to be causally unrelated
- The reconciled version superseding the current versions is written back

(3) Handling temporary failures

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Partitioning</td>
<td>Consistent hashing</td>
</tr>
<tr>
<td>(2) High availability for writes</td>
<td>Vector clocks with reconciliation during adds</td>
</tr>
<tr>
<td>(3) Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff</td>
</tr>
<tr>
<td>(4) Recovering from permanent failures</td>
<td>Anti-entropy using Merkle tree</td>
</tr>
<tr>
<td>(5) Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection</td>
</tr>
</tbody>
</table>

Example

```
Example

The data will be sent to the node D

If it is temporarily down

This data contains a hint in its metadata

-- node where it was supposed to be stored
```

What if W is 1?
- Applications that need the highest level of availability can set W as 1
  - Under Amazon’s model
  - A write is accepted as long as a single node in the system has durably written the key to its local store
  - A write request is rejected,
    - Only if all nodes in the system are unavailable
FAQs

- Programming Assignment 1 has been posted.
  - Due on Oct. 13
- Midterm 1
  - Oct. 2
  - Dynamo
  - Start with slides
  - Try examples
  - Read papers
  - 6 problem sets
    - Each of them will contain several sub-problems
    - 150 points (15% of cumulative course grade)
- Term project: Phase II

(4) Recovering from permanent failures

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Partitioning</td>
<td>Consistent hashing</td>
</tr>
<tr>
<td>(2) High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
</tr>
<tr>
<td>(3) Handling temporary failures</td>
<td>Sleepy Guacamole and hinted handoff</td>
</tr>
<tr>
<td>(4) Recovering from permanent failures</td>
<td>Anti-entropy using Merkle tree</td>
</tr>
<tr>
<td>(5) Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection</td>
</tr>
</tbody>
</table>

Anti-entropy protocol

- Replica synchronization protocol
- Hinted replica can be lost before they can be returned to the original replica node
- Detect inconsistencies between the replicas faster
- Minimize the amount of transferred data
- Dynamo uses Merkle tree

Merkle tree

- Hash tree where leaves are hashes of the values of individual keys
  - Parent nodes are hashes of their respective children
- Each branch of the tree can be checked independently
  - Without requiring nodes to download the entire tree or dataset
- If the hash values of the root of two trees are equal
  - The values of the leaf nodes in the tree are equal
  - No synchronization needed

Uses of Merkle tree

- Merkle trees can be used to verify any kind of data stored, handled and transferred.
  - Used in a peer-to-peer network
- Trusted computing systems
  - Sun’s ZFS (Zeta File System)
- Google’s Wave protocol
- Git
- Cassandra and Dynamo
- BitTorrent protocol

How Merkle tree works

- $H_0 = H_2$ means concatenating two values
- $H_3$ = Hash value of $H_0 + H_1$
- $H_4$ = Hash value of $(H_0 + H_1 + H_2)$
How Dynamo uses Merkle tree

- Each node maintains a separate Merkle tree for each key range
- Two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common
- Node performs tree traversal and determines if there is any difference
- Perform the appropriate synchronization action

Disadvantage
- When a new node joins or leaves
  - Tree needs to be recalculated

How Merkle tree works for Dynamo

1. Top Hash A
   - H5
   - H6

2. Top Hash B
   - H10
   - H11

3. If H6 vs. H10 did not match
   - Compare H5 vs. H13
   - Compare H6 vs. H14

4. If H6 and H14 did not match
   - Compare H3 vs. H12
   - Compare H4 vs. H13

Ring Membership

- A node outage should not result in re-balancing of the partition assignment or repair of the unreachable replicas
- A node outage is mostly temporary

- Gossip-based protocol
  - Propagates membership changes
  - Maintains an eventually consistent view of membership

- Each node contacts a peer every second
  - Random selection
  - Two nodes reconcile their persisted membership change history

Logical partitioning

- Almost concurrent addition of two new nodes
- Node A joins the ring
- Node B joins the ring

- A and B consider themselves members of the ring
- Yet neither would be immediately aware of each other
- Logical partitioning

External Discovery

- Seeds
  - Discovered via an external mechanism
  - Known to all nodes
  - Statically configured (or from a configuration service)

- Seed nodes will eventually reconcile their membership with all of the nodes
Communication failure

- Attempts to
  - Communicate with unreachable peers during a get or put operation
  - Transfer partitions and hinted replicas

- Detecting communication failures
  - When there is no response to an initiated communication

- Responding to communication failures
  - Sender will try alternate nodes that map to failed node’s partitions
  - Periodically retry failed node for recovery