Manage Big Data in the Cloud -- NoSQL Databases

SQL and NoSQL

- Why NoSQL
  - SQL
    - Some applications require the use of relational databases
    - But many applications does not require the features that are source of performance issues, such as the join operation, foreign keys, …
    - It is harder for RDB to achieve horizontal scalability (sharding)
    - So, whenever RDB is not necessary, use NoSQL DB
- NoSQL system categories
  - Key/value store
  - Wide Column store
  - Document store
  - Graph store

NoSQL Categories

- Key-value store
  - Schema-less value field
  - Example: Amazon Dynamo, Yahoo Pnuts
- Wide Column store
  - Key-value store, but value fields can have schema definition
  - Similar to table columns, but allow nested column definitions
  - Major examples
    - Google Big table, Hbase, Cassandra

NoSQL Categories

- Document store
  - Each row is a document, also schemaless
    - Generally semi-structured (like JSOM) in the document, though no requirement for that
  - Support retrieval based on keywords in the documents
  - Major examples: MongoDB, CouchDB
- Graph store
  - Emphasize the relations between nodes (each row)
  - Best for social networks
  - Major examples, Neo4j

NoSQL History

- Forerunners
  - GBT, Dynamo, Pnuts
  - Each has its own special applications
    - Found that RDB is overkill, and cannot scale
- New systems
  - General design for wide usability
  - Some introduces new design concepts
    - Cassandra, MongoDB, Redis
- Memory based storage
  - Redis, MemCacheDB, etc.

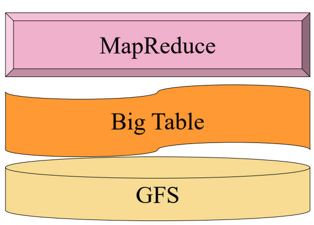
Design Issues Revisited

- CAP (old tech, new terminology)
  - Consistency
    - Sequential consistency
  - Availability
    - Always available for read and write
  - Partitioning tolerant
    - Tolerance of the network partitioning problem
  - Conflict between availability and consistency
    - To assure sequential consistency, read/write may be blocked
    - To avoid blocking and assure availability, the system should tolerate a certain degree of inconsistency
    - If no network partitioning, quorum approach can work
      - Still require bounded computation and communication

Google GBT

Google Cloud Technology Stack

- Big data solution stack from Google



  - GFS stores unstructured data
  - Bigtable stores structured data
  - MapReduce handles batch processing over both
- Big table on top of file system?
  - Besides GBT and HBase, no other nosql DB follows this solution stack

Google Big Table -- Data Model

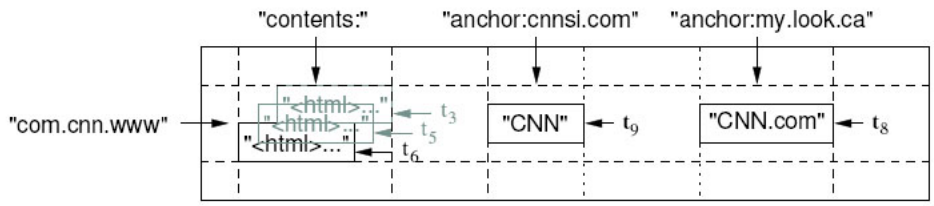
- Data model
  - (key, columns, time) → string
    - Key
      - Supports up to 64 KB, but mostly10-100 B
      - Table is sorted by row key
    - Data type of the columns are all strings, no size limit
    - Time: version of the cell
      - 64 bit integers, real-time in microseconds
      - Can be assigned by Bigtable or the application
      - Application specifies the number of versions to be kept for the cell
      - Bigtable provides garbage collection to remove obsolete versions
    - Transaction support:
      - Changes to multiple columns with the same row key are performed atomically, no cross-row transactional support

Google Big Table -- Data Model

- Data model
  - Column family
    - Column <family: qualifier>
      - One or more columns can be grouped into one column family
    - Locality group
      - Group multiple column families into a locality group
      - Separate storage (SSTable) for each locality group in each tablet
      - Segregating column families that are not typically accessed together, enabling more efficient reads
    - The basic unit of access control
    - The basic unit for compression
      - No need to compress all fields of a row, but can optimize compression for multiple rows of the same column

Google Big Table -- Data Model

- Google usage
  - Keep copies of a large collection of web pages
  - Use URL as the row key
  - Various aspects of web page as column names
  - Content of the web page is stored in the <contents:column> with the timestamps when they were fetched



Google Big Table -- Storage Basics

- Storage mechanism
  - Split a table into tablets for storage
    - Each is a group of row keys (consecutive in the table)
    - Facilitates range search: Reside on a few platforms
  - A tablet is a unit of storage, stored on one tablet server
    - Each tablet is ~100MB to 200MB   Why a range, not a fixed size?
    - Each tablet server stores 10-1000 tablets
    - Support fast recovery
      - E.g., 100 nodes 1000 tablets, each picks up 10 tablets of a failed node
    - Support fine-grained load balancing
      - Transferring 100-200MB won't create much traffic
    - Make directory management reasonable

Google Big Table -- System Architecture

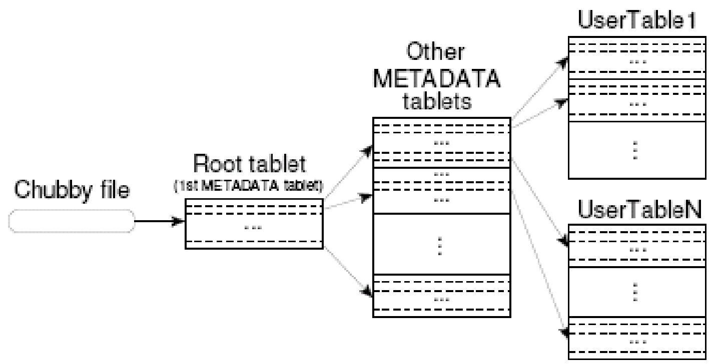
- One master server
  - Table creation/deletion
  - Assigning tablets to tablet servers
  - Maintain the METADATA table (tablet directory)
  - Manage the status of tablet servers (alive, etc.)
    - Use Chubby to achieve synchronization with tablet servers
  - Balance the load of the tablet servers
  - Merge tablets
    - Better done at the master (global knowledge, knowing two tablets)
  - Garbage collection (maintain max of N versions)

Google Big Table -- System Architecture

- Many tablet servers
  - Store tablets and process read and write requests on them
  - Splits tablets that have grown too large
    - Merge is initiated by the master, but split can be initiated by the tablet server locally
    - Tablet server: Split a tablet → Notify the master to update the METADATA table
    - In case of failure (tablet server or master died)
      - Master will detect the new tablets when it asks a tablet server to load the tablet that has been split
- Library linked into every client
  - Client directly communicate with tablet server for R/W
  - Only communicate with Master to find the tablet location

Google Big Table -- Sharding

- Directory of the tablets
  - Three tier directory tables
    - Root table, metadata table, user table
    - Each directory level table are also stored as tablets

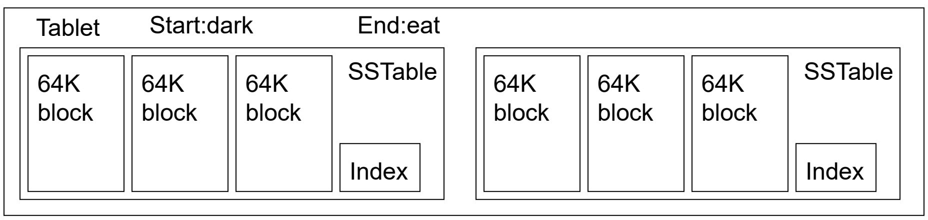


Google Big Table -- Sharding

- Directory of the tablets
  - The root table is a fixed, single tablet, never splits
    - So that the entry to the root table is always known
    - Metadata table tablets and user table tablets may be added, deleted
  - Client library caches tablet locations
    - Include the chain from root to metadata to user tables
    - Client first uses the locally cached pointers to locate the tablet server; If fails (not cached or outdated), use the locally cached pointers to locate the user table tablet, … (recursively up the hierarchy)
      - Save search time, does not need to always start from the root, which is most busy and the search path can be long
    - Prefetch multiple tablet rows when access the User Table

Google Big Table -- Tablet Storage

- A tablet is stored in the form of SSTables
  - SSTable (Sorted String Table)
    - Immutable storage units for tablets
    - Stores data blocks, each is of 64KB
    - Keeps an index: keys and corresponding offsets (block, offset)
    - Stored as a GFS file
- Replication
  - By GFS, tablet server has no knowledge of it

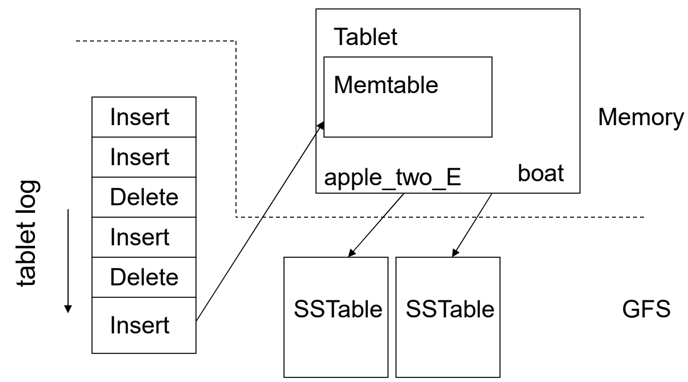


Google Big Table -- Tablet IO

- Write
  - Updates are first logged in memory, then flushed to disk
    - Server side "memTable" (fixed size), not the client memory
    - Logs are structured in a tree, log structured merge tree (LSM)
  - When one memTable is full
    - Logs in the memTable are written to disk as one SSTable
    - Group commit all logged writes after written to disk
    - New writes are logged in a new memTable
    - When written to disk, memTable is appended at the end of the log file (use record append in GFS) ⇒ Yields a faster sequential write
- Read
    - Read from logs in memory and on disk (SSTalbes)
    - Read from the data SSTables (ordered)
    - Efficient write (even for random writes), slower read

Google Big Table -- Tablet IO

- Group commit on log
  - Changes are logged in-memory "memtable"
  - Group commit the logs



Google Big Table -- Tablet IO

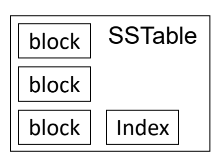
- After a while, log may become too long
    - Difficult to recover, inefficient to read (search) ⇒ Compact
  - Minor compaction
    - Compact the memTable before writing to disk SSTable
  - Merge compaction
    - Compacts a few "logging" SSTables to create a new "logging" SSTable (discard the original ones after compact)
    - Done periodically by a background process
  - Major compaction
    - Merges the data SSTable and all its logging SSTables into one SSTable, remove all deleted entries from the original SSTable
    - Done periodically by a background process

Google Big Table -- Tablet IO

- One log file per tablet or one per tablet server
  - On a tablet server, there may be concurrent writes to multiple tablets
  - One log file per tablet could cause inefficiency
  - Use one log file for all tablets
    - More efficient in handling writes
    - Harder to manage and can have inefficient recovery
    - GBT still uses this optimization

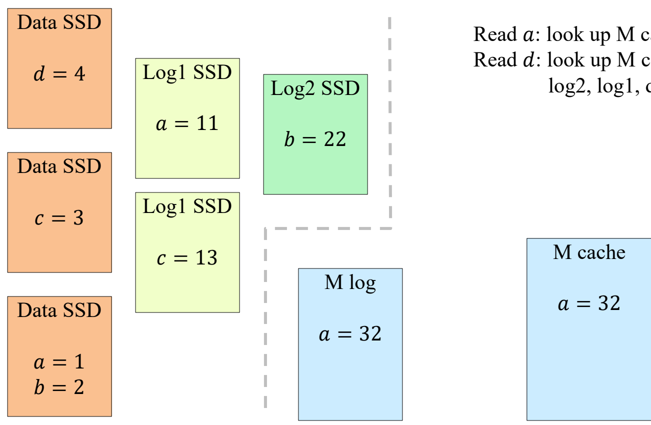
Google Big Table -- Tablet IO

- Read some key(s) from a tablet
  - Need to search for the keys
    - Sequential search will be too inefficient
  - Solution: In-memory index tree for the tablet
    - Each tablet has metadata, which points to all data and log SSTables
    - Read all index tables of all data SSTables of the tablet
    - Read all the log SSTables and apply the updates to the index tree
    - Search the index tree, then retrieve data for the specific keys
  - Cache of data
    - Scan cache: cache the rows recently read
    - Block cache: cache the blocks recently read
      - Server side cache, in memory
    - Scan the log SSDs



Google Big Table -- Tablet IO

- Read some key(s) from a tablet
  - If it is cached in memory, great, but if not?



Read a: look up M cache
Read d: look up M cache, M log, log2, log1, data SSDs

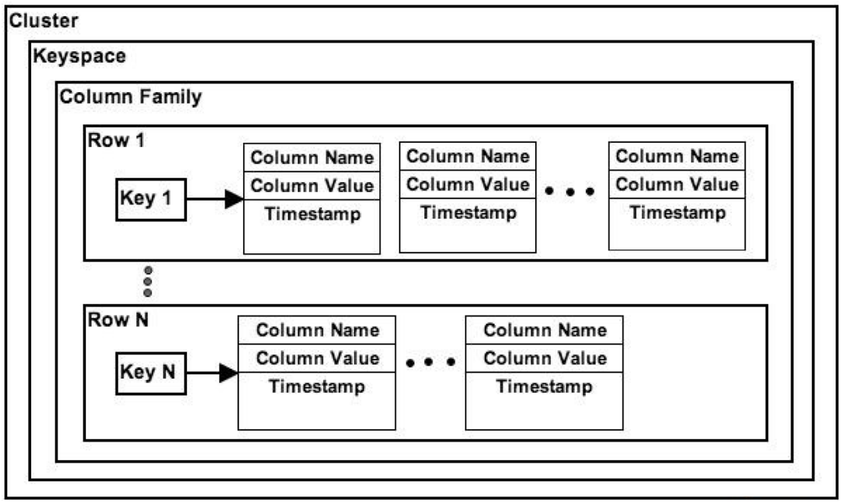
Google Big Table -- Tablet IO

- Read requires search for the key(s)
  - A tablet has multiple SSTables
  - Search may need to load the indices of multiple SSTables
    - Multiple disk blocks to be loaded
  - Use Bloom filters (tablet level)
    - User specifies whether to use a Bloom filter
      - One Bloom filter per SSTable, to help quick prediction of the existence of keys in each SSTable
        - » Bloom filters for all SSTables of a tablet are together, so load once
      - Most useful for identifying non-existing keys
    - Bloom filter
      - Hash the key to multiple hash values, say 3
      - Probability of collision in all hash tables is slim

Hadoop and Google

	Google	Apache
Sponsor	Google	Yahoo, Amazon
Open-ness	Open document	Open source
Computing PaaS	MapReduce	Hadoop MapReduce
File System	GFS	HDFS
Storage System (for structure data)	Bigtable	HBase
Search Engine	Google	Nutch

Cassandra – Data Model

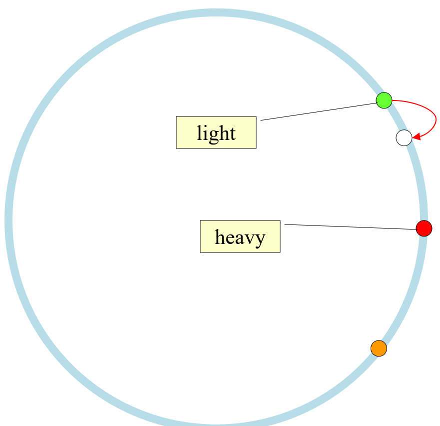


Cassandra – Sharding

- Data placement
  - Use DHT ring, offer two placement options
    - Regular consistent hashing
    - Order preserving partitioning
  - Order-preserving partitioning
    - The plain key, not the hash, is used for placement
    - Continuous key space may be retrieved together (range query)
    - Key space may not have a uniform distribution
      - Sometimes, user may have to specify the physical node location to ensure balanced loads on nodes
  - Replicas placement policies
    - On N successors
    - The first successor is the coordinator, who is responsible for data replication for the key

Cassandra – Sharding

- Data placement
  - Load balancing: Lightly loaded nodes advance on the ring



Only need to exchange load information with neighbors

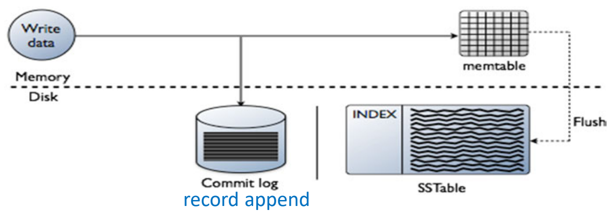
Equivalent to:
Heavily loaded nodes go backwards in the ring

Cassandra – Consistency

- Offer multiple consistency models
  - Zero
    - Immediately respond to client, update is asynchronously forwarded to the secondaries in the background
  - One
    - Respond to the client only after assuring that the update got written to the commit log (on disk) of at least one data server
  - Any
    - Same as one, but can be a hinted handoff (one has to be a replica node)
  - Quorom
    - Read/Write quorums both include (N+1) / 2 nodes
  - All
    - Only respond to client after the write goes to the memTable and disk of all nodes
    - Node failure will cause the write to fail

Cassandra – Consistency

- Read/write protocols
  - Quorum based read/write
    - Access request is by the key, and routed to one replica, say X
    - X sends the request to other replicas and other replica sends ack back to X after logging (in commit log)
    - X responds to the client after the quorum is reached
- Efficient write by logging
  - Same as GBT, use SSTable, but add a commit log



Cassandra – Performance

- Performance evaluation
  - Retrieve > 50 GB data
  - MySQL
    - Writes 300 ms, Reads 350 ms
  - Cassandra
    - Writes 0.12 ms, Reads 15 ms
    - Read is slower than write
    - 1000 times faster than SQL DB
  - No evaluation about the impact and performance of the distributed membership protocol

Cassandra – Membership

- Membership
  - Cassandra system maintains the list of data servers
    - When there are node failures or additions, use a gossip protocol (scuttlebutt) for membership updates
    - Member list is maintained distributedly by all data servers
      - Unlike Swift and Ceph, no central server
  - Use a fuzzy failure detection mechanism
  - One node x may consider another node y has failed if x does not receive any response from y
    - But also associate a fuzzy value with it
    - It is time-out based, the longer the time has expired, the higher fuzzy value will be for considering that the node has failed

Cassandra – Membership

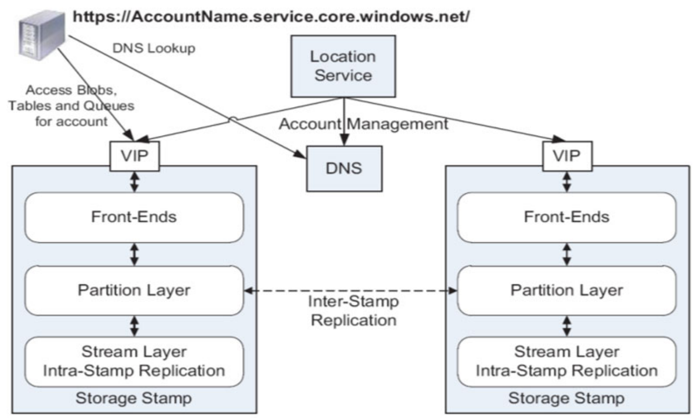
- Membership
  - Scuttlebutt: A p2p gossip protocol
    - Each node has a small subset of neighboring nodes, and will send update messages to neighbors
    - Eventually, the update message will be propagated to all nodes
    - The version number for each updated object is a vector clock
  - Update request
    - Make the update on the object
    - Increment its local clock by 1, use the vector clock as the version number for the updated object

WAS

- Namespace structure
  - URI for each object, with a naming hierarchy
  - Account
    - Like a tenant, can have many partitions
  - Partitions
    - Probably corresponds to a user or a group of users
    - Can have many objects
  - Objects
  - Storage design decisions
    - The objects of the same account are stored together
    - In GFS, the owners of the files are irrelevant, but in WAS, allocation considers owners (tenant based)

WAS Storage Stamp Architecture

- Storage Stamp
  - Typically, a storage stamp is a cluster with 10-20 racks, and a rack has 18 disk-heavy storage nodes
    - A rack is a separate fault domain (w. redundant network & power)



WAS Storage Stamp Architecture

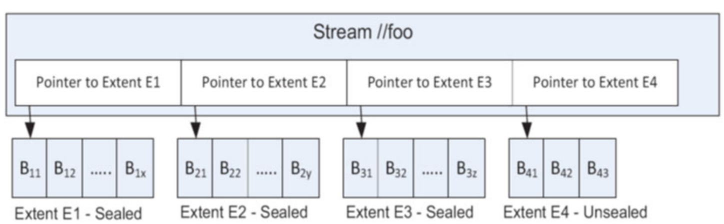
- Location and location service (LS)
  - A location = a data center has many storage stamps
    - Azure has 24 data centers in 2011, over US, Europe, Asia
  - LS manages all the locations and their storage stamps
    - Maintains a map of all nodes in a hierarchy
  - LS manages replication
    - Intra-stamp replication: Synchronous IO
    - Inter-stamp replication: Asynchronous backup
- Three layer management in each stamp

WAS Storage Stamp Architecture

- Three layer management
  - Stream layer
    - Manage the actual data storage
    - A stream may be one object or multiple objects
      - Small objects are appended together in an extent
    - A stream is divided into extents, it is a list of pointers to its extents
      - An extent is a unit for allocation
  - Partition layer
    - Stream layer is not aware of replications
    - Manage replica placements and consistency maintenance
  - Front-end
    - Use a virtual IP (VIP) to accept client requests
    - Cache a partition map to determine which partition server is in charge of the requested data objects

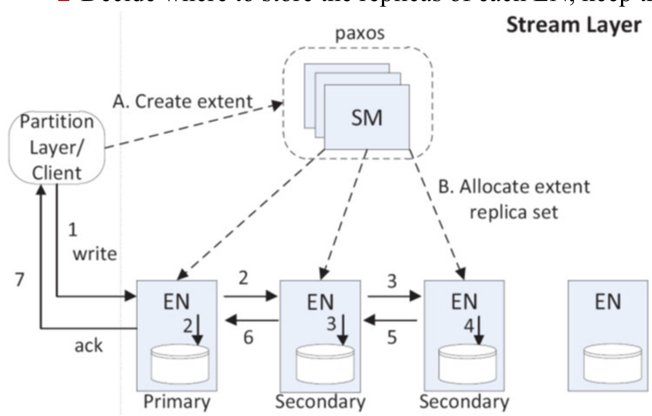
WAS Stream

- Stream structure
  - Extent: unit of placement/replication (like chunk)
  - Block: regular file blocks, each extent has a set of blocks
  - Append only (like the name indicates)
    - Only one extent is open for append, the rest are "sealed"
    - Replication for unsealed, erasure coding for "cold" sealed



WAS Stream

- Stream management
  - EN: extent node (data server)
  - SM: stream manager server (master)
    - Decide where to store the replicas of each EN, keep track of them



WAS Stream

- Stream append
    - Block is the unit for any IO in a stream
  - Append request sent to primary EN
    - Primary EN decides offsets and send the info to secondary
      - Will be consistent even if there are concurrent append requests
    - Other ENs forward the request in a pipeline
      - Write the new block to disk and forward the request in parallel
    - ACK after the new block is written
  - Primary-backup structure for each extent
    - Primary EN decides write (append) ordering and send to backups
    - Primary EN responds to client if received all ACKs
      - “Commit length” of an extent = offset of the last committed append
      - Commits have to be in the order of the offsets
    - When there is a failed EN in the group ⇒ seal the extent

WAS Stream

- Stream append
  - Journaling
    - Original WAS does not consider journaling ⇒ Incur congestions
    - Dedicated disks for journaling ⇒ Maximize the benefit of sequential writes
    - One journal disk per account
      - To support isolation among accounts (actually, among RangePartitions, which may correspond to an account or there may be multiple RangePartitions per account)
    - Each write is written to the proper extent as well as the journal
    - As long as one is successful ⇒ Respond to the client as "success"

WAS Stream

- Extent sealing
  - SM is in charge of sealing
  - SM ask all replica ENs their commit length
    ⇒ Choose the smallest commit length as the final value
    - Even the smallest one should contain all committed appends
    - May contain some uncommitted appends, but it is fine as long as it is consistent
  - For unreachable node
    - No problem, even if the EN comes back or becomes reachable, SM provides the commit length and forces this EN to synch to this commit length
  - Extent content is partitioned and erasure coded
    - Fragments are at block boundary, so may not be fully equal sized

WAS Stream

- Stream replication
  - Intra-stamp replication
    - One stamp can be the main storage for a stream
    - Synchronous replication within the stream
  - Inter-stamp replication
    - Can have additional replicas outside the main stamp to make the data more robust
    - Asynchronous replication
  - Stream write
    - Block is the unit for updates
    - Primary-backup approach

WAS Stream

- Stream read
  - Block is the unit for reading as well
  - Balance load in a replicated extent
    - Client chooses one replica to read the block, and gives a deadline
    - If the replica cannot satisfy the deadline, reply back
    - Client can go to another replica
  - Balance load in a sealed and erasure coded extent
    - If the replica cannot satisfy the read deadline, reply back
    - Client may rather perform reconstruction, rather than wait for the busy replica

Cloud Storage Systems So Far

	GFS/GBT	HDFS/HBase	Ceph	Dynamo Swift	Cassandra	Azure
Data place.	central	central	DHT-Rush	DHT ring	DHT ring	Central
Data R/W	Primary/… Read any	Central lock Read any	Primary/… for both r,w	Quorum-sloppy Timestamp	Multiple schemes	Primary/… Read any
W N objects		File level lock	Lock N obj.			
Metadata	Master	Name node	MDS	Container extern. Index.	None	
Metadata r/w	Central/log		Primary/…	Periodical scan		
Metadata lock	r/w lock		r/w lock			
Membership	Chubby		Monitor	Gossip	Gossip	
Load balancing	Central	Central		Virtual node		Central
Load balan. R	Client	client	By Primary	Proxy		

Summary: Data Consistency Protocols

- Mechanisms

  Assume group commit: specified in GBT, not in GFS

  - GFS: Primary backup update protocol, but read any
  - HDFS: Locking for updates, read any without lock
  - Dynamo: Sloppy quorum with timestamp
    - Read: to any node in the ring
    - Write: to one of the replica servers ⇒ Avoid long vector time
      - Consider preference list (all replicas) and top N nodes
      - But generally, there won't be that many replicas
  - Ceph: Primary backup for both read/write
    - Strongest among all
  - Cassandra: Multiple levels, but based on read R write W replicas
    - Similar to Dynamo
- Update to multiple objects
  - HDFS: lock at file level ⇒ consistent w. to multi-blocks of a file
  - Ceph: if updating multiple objects ⇒ lock all of them

Summary: Data Consistency Protocols

- Example of update effects
  - Originally: D_x = 5
  - Concurrent writes: W(D_x,7), W(D_x,9)
    - Ordering: as above, by HDFS lock, by primary of GFS and Ceph
- R(D_x); R(D_x) issued after W(D_x,7)
  - Ceph: Read a specific value depending on the primary order
  - HDFS: Read: 5, 7 or 7, 9 in any order
  - GFS: Read: 5, 7, 9 in any order   Depends on group commit boundary
- Dynamo: need to consider concurrent timestamp for W
  - Servers A, B, C: A handles W(D_x,7), B handles W(D_x,9)
  - A's W(D_x,7) has TS=(2,3,1), if reaches B
    - Before W(D_x,9) ⇒ TS for W(D_x,9) has to > (2,3,1), e.g., (2,4,1)
    - After W(D_x,9) ⇒ TS for W(D_x,9) may be, e.g., (1,4,0)
  - Read: 5, 7, 9 in any order

Summary: Metadata Maintenance

- What metadata to be maintained
  - Where data objects are
    - DHT will not need this
    - GFS and HDFS use central tables
  - Directory (Folder) structure for file systems
    - Ceph uses the MDS cluster for this
      - Follow the POSIX standard, same as Unix FSs
      - Taking care of file size and time modified while updates are going on
    - GFS and HDFS use master and namenode for this
      - GFS does not have the directory content (unlike Unix FS's)
      - Master maintains the file names in one data structure
      - So, HDFS supports ls, but not the detailed information
    - Cassandra does not maintain directory structure

Membership:
Is also metadata,
But in a different category

Summary: Metadata Maintenance

- What metadata to be maintained
  - Directory structure for file systems
    - Swift does not maintain directory structure
    - But it supports keyword search
    - It sends metadata to an external indexing service
      - E.g., ElasticSearch
      - To facilitate keyword based search in the storage
      - A background process periodically scan the database and send the updates to the external indexing service

Summary: Metadata Update

- Locking for metadata updates
  - Example 1
    - Create or delete a file /home/usr/foo
    - GFS
      - Read lock /, /home, /home/usr; write lock /home/usr/foo
      - Read lock can prevent the directory from being deleted
      - No need to write lock /home/usr, because there is no directory content
      - Multiple files under the same directory can be created concurrently, but no two files with the same name can be created concurrently
    - Ceph
      - Write lock /home/usr
      - Need to update /home/usr, add foo to it
      - The rest will be the same
    - You can consider other commands, like copy, create, etc.
      - With the same principle

Summary: Membership Maintenance

- GBT

  Chubby: A simple FS, can provide highly-available persistent lock service
  - Use Chubby for membership management
  - When a tablet server starts:
    - Create a uniquely-named file in a special directory in Chubby and acquire the lock to the file
      - This lock should be held permanently
    - Finds all tablets and their SSTable it serves from METADATA table
    - If a tablet server loses its lock
        - » If the lock fails or the file no longer exists, or a network partition or other error has broken the Chubby session
      - Tablet server tries to get back the lock
      - Table server terminates if it fails to get back the lock or the file no longer exists
      - A tablet server will not serve without the lock

72

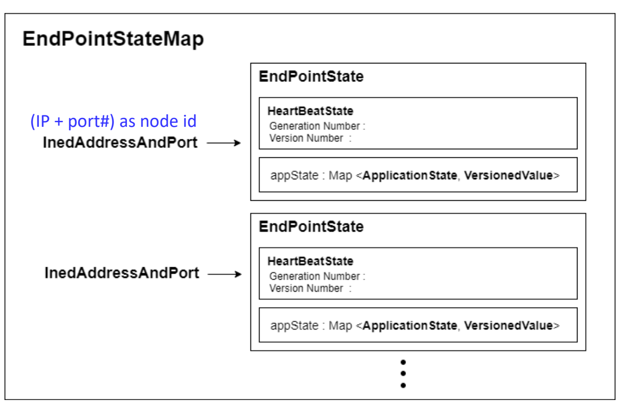
Summary: Membership Maintenance

- GBT
  - When a Master starts (by cluster manager)
    - Acquire a unique lock on Chubby
      - Prevent to have multiple Masters
    - Determine the list of running tablet servers by examining the special directory in Chubby
    - Communicate with each live tablet server to get their tablet information
      - In case there are unfinished updates (e.g., unfinished split or merge updates), complete the updates
    - Scan through the METADATA table to detect unassigned tablets and assign them
- Ceph
  - Centralized manager – monitor(s)

73

Summary: Membership Maintenance

- Cassandra
  - Use a gossip protocol for membership status updates
  - Node exchange the entire map periodically

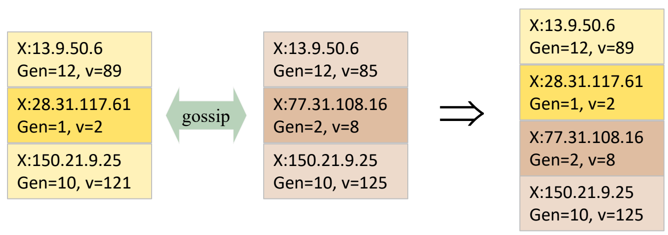


https://medium.com/@swarnimsinghal/implementing-cassandras-gossip-protocol-part-1-b9fd161e5f49

74

Summary: Membership Maintenance

- Cassandra
  - Exchanged information
    - Each reboot creates a new “generation number”
    - Version number recorded by node A regarding node B’s state = the latest timestamp of the message sent from B to A
      - A’s view: Up to that time, B has been alive
      - Note: it is B’s timestamp, not A’s ⇒ No time consistency issue
  - After exchange: set to the highest version (simplified)



75

Summary: Membership Maintenance

- Cassandra
  - Exchanged information between nodes X and Y
    - There are additional information to be exchanged
      - E.g., node status, node load, etc.
    - Multiple stage exchange
    - Sync: X sends (node id + generation + version) to Y
    - ACK: Y to X, for each node
      - For the nodes that Y has a more recent (generation, version) ⇒ Send the detailed information to X
      - For the nodes that X has a more recent (generation, version) ⇒ Ask X to send the detailed information of these nodes
    - ACK2: X sends back to Y
      - The detailed information Y needs

76

Summary: Membership Maintenance

- Cassandra
  - Who to exchange info with?
    - Each node keeps K neighboring nodes
      - Could be based on physical or logical structure or random selection
    - Every second, randomly pick 1 (up to 3) neighbor to gossip
  - Gossip protocol in general
    - How to minimize #messages for spreading each news
      - When to start and when to stop spreading news
        - E.g., periodical starting or event driven starting, count based or probability based stopping
      - Pull or push or pull&push
        - E.g., for a brand new news: push may be better, but if news had been spread for a while, pull is more likely to reach an informed node
      - Neighbor selection: will it let a message be spread to all nodes
      - …

77

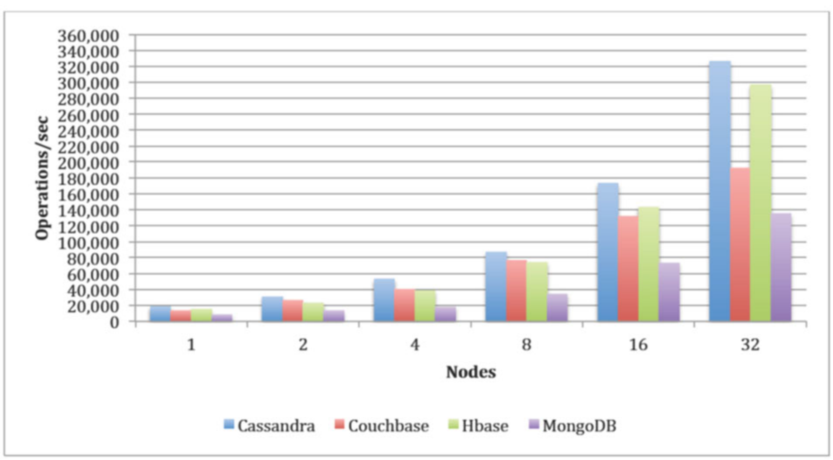
Summary: Load Balancing

- General schemes
  - Change data placement
    - GFS/GBT
      - Central mater decides where to place/move the data
    - Ceph: change weight offline
    - Dynamo: virtual node
    - Cassandra: change physical node place on the ring
- Read load balancing
  - Forward read request to a replica host with lighter load
    - GFS/GBT: master returns all replica hosts ⇒ client decides
    - Ceph: go through primary ⇒ primary forwards
    - Dynamo: go through proxy ⇒ proxy forwards
    - Cassandra: by any node on the ring that receives the request

Summary: WAS and GFS

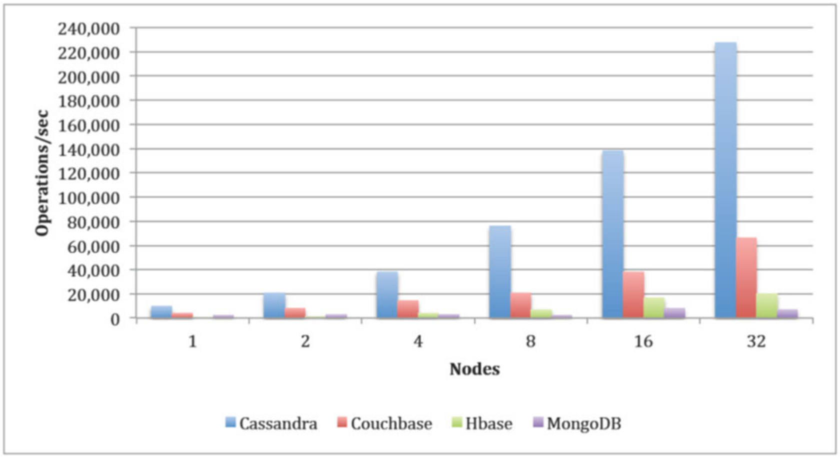
- WAS and GFS/GBT are very similar
  - Objects
    - Stream ≈ GBT table; Extent ≈ GBT tablet; Block ≈ GBT block
    - Stream is "Append only" ≈ GFS record append protocol
      - Append only ⇒ No need to have SSD like structure
    - Data consistency: same as GFS (primary-backup, r/w protocols, primary lease by PM)
    - Same split, merge, load balancing schemes
      - Discussed in more detail in the WAS paper
  - Metadata
    - Metadata stored in PM (partition manager) = GFS/GBT master
    - No discussion about locking for metadata update
  - Membership management: Chubby

Performance Comparisons



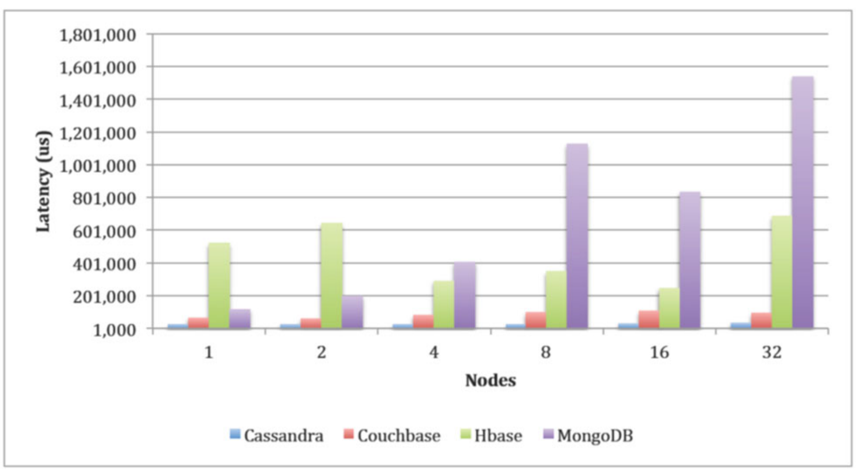
Bulk Insert (initial loading)

Performance Comparisons



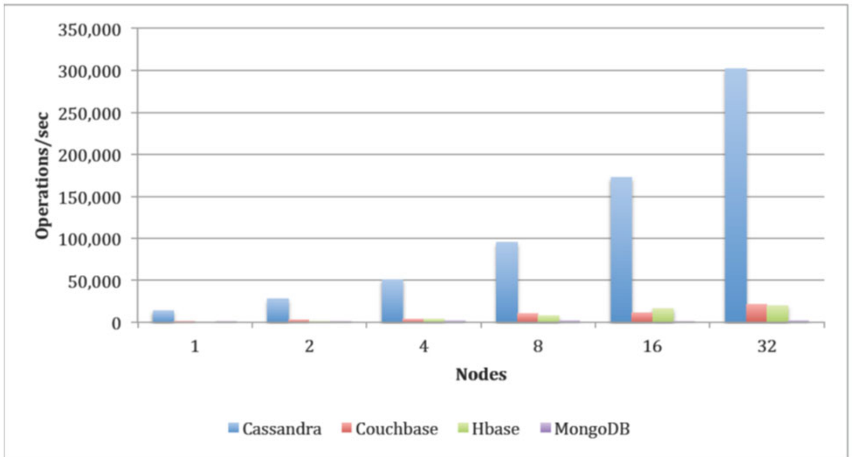
Mostly Read Workload

Performance Comparisons



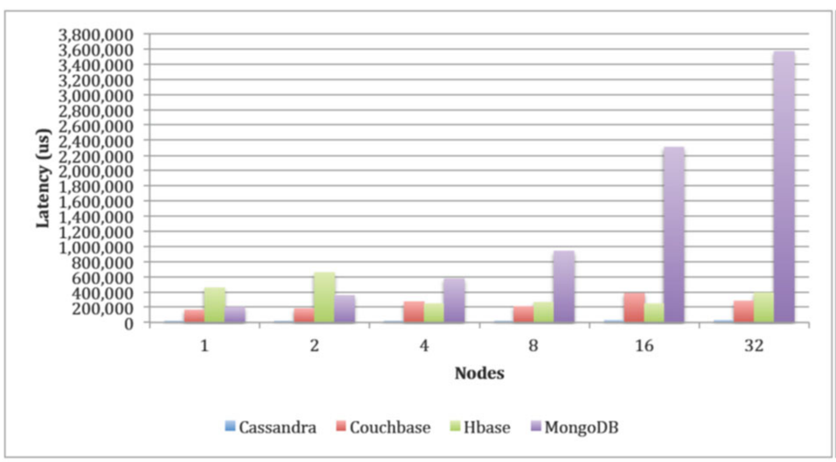
Mostly Read Workload

Performance Comparisons



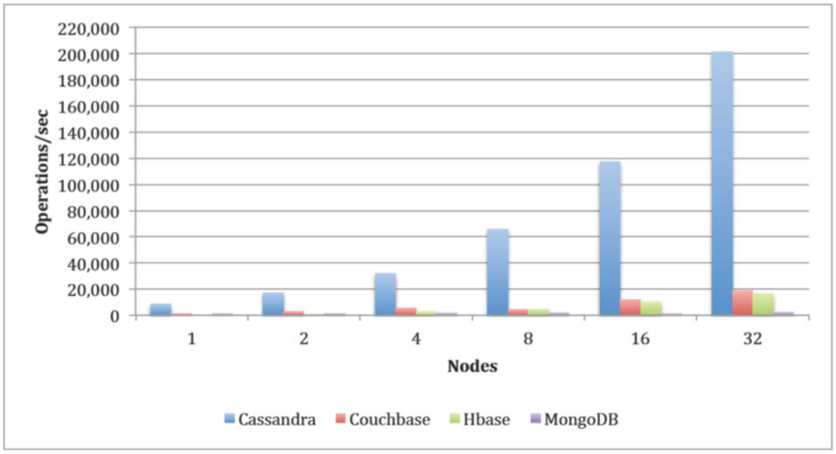
Balanced Read/Write Mix Workload

Performance Comparisons



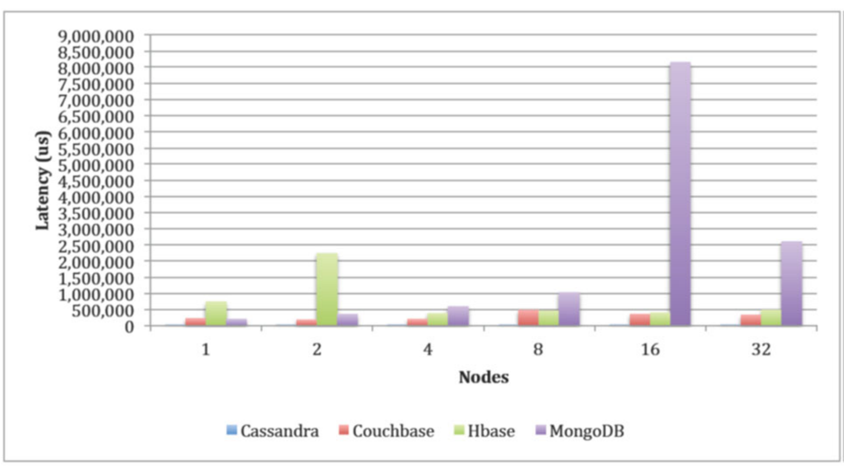
Balanced Read/Write Mix Workload

Performance Comparisons



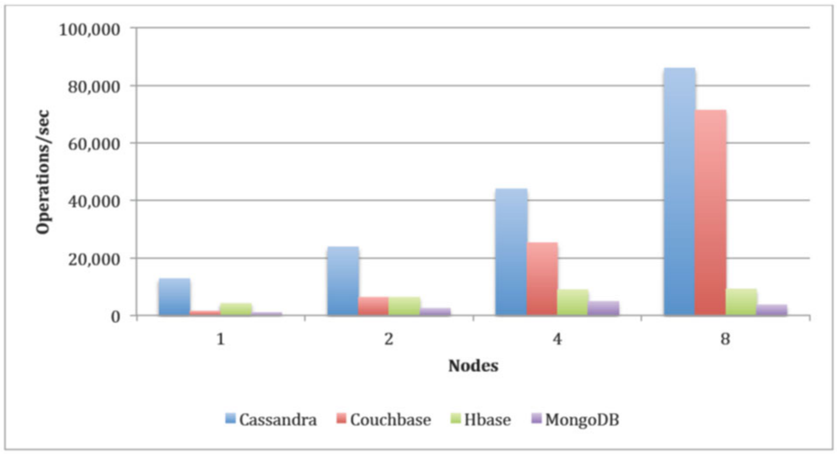
Read-Modify-Write Workload

Performance Comparisons



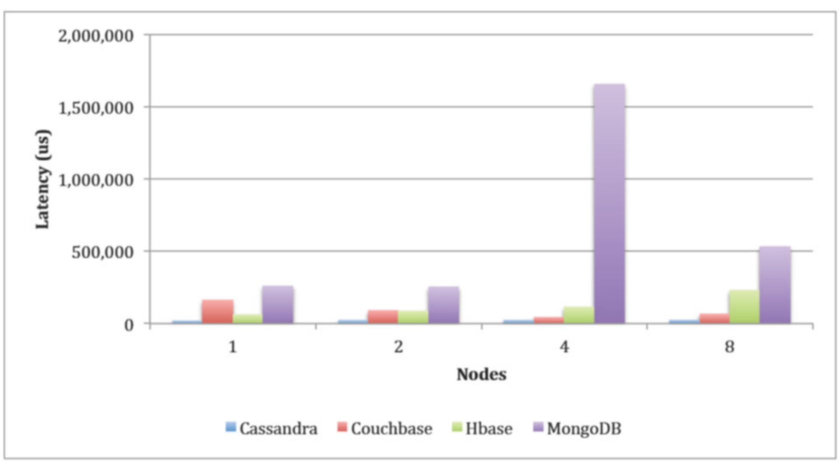
Read-Modify-Write Workload

Performance Comparisons



Mostly Insert Workload

Performance Comparisons



Mostly Insert Workload

References

- GBT
  - Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, “Bigtable: A distributed storage system for structured data,” ACM Trans. Comput. Syst., Vol. 26, No. 2. (June 2008), pp. 1-26.
- Cassandra
  - Avinash Lakshman, Prashant Malik. "Cassandra: a decentralized structured storage system." ACM SIGOPS Operating Systems Review, vol. 44, No. 2, 2010, pp. 35-40.
- WAS
  - B. Calder et al., “Windows Azure storage: A highly available cloud storage service with strong consistency,” ACM SOSP, 2011