Dynamo & Bigtable

CSCI 2270, Spring 2011
Irina Calciu
Zikai Wang

Dynamo

Amazon's highly available key-value store

Amazon's E-commerce Platform

- Hundreds of services (recommendations, order fulfillment, fraud detection, etc.)
- Millions of customers at peak time
- Tens of thousands of servers in geographically distributed data centers
- Reliability (always-on experience)
- Fault Tolerance
- Scalability, Elasticity

Why not RDBMS?

- Most □Amazon services only needs read/write by primary key
- RDBMS's complex querying and management functionalities are unnecessary and expensive
- Available replication technologies are limited and typically choose consistency over availability
- Not easy to scale out databases or use smart partitioning schemes for load balancing

System Assumptions & Requirements

- Query model: no need for relational schema, simple read/write operations based on primary key are enough
- ACID Properties: Weak consistency (in exchange for high availability), no isolation, only single key updates
- Efficiency: function on commodity hardware infrastructure, be able to meet stringent SLAs on latency and throughput
- Other assumptions: non-hostile operation environment, no security related requirements

Design considerations

- Optimistic replication & eventually consistency
- Always writable & resolve update conflicts during reads
- Applications are responsible for conflict resolution
- Incremental scalability
- Symmetry
- Decentralization
- Heterogeneity

Architecture Highlights

- Partitioning
- Replication
- Versioning
- Membership
- Failure Handling
- Scaling

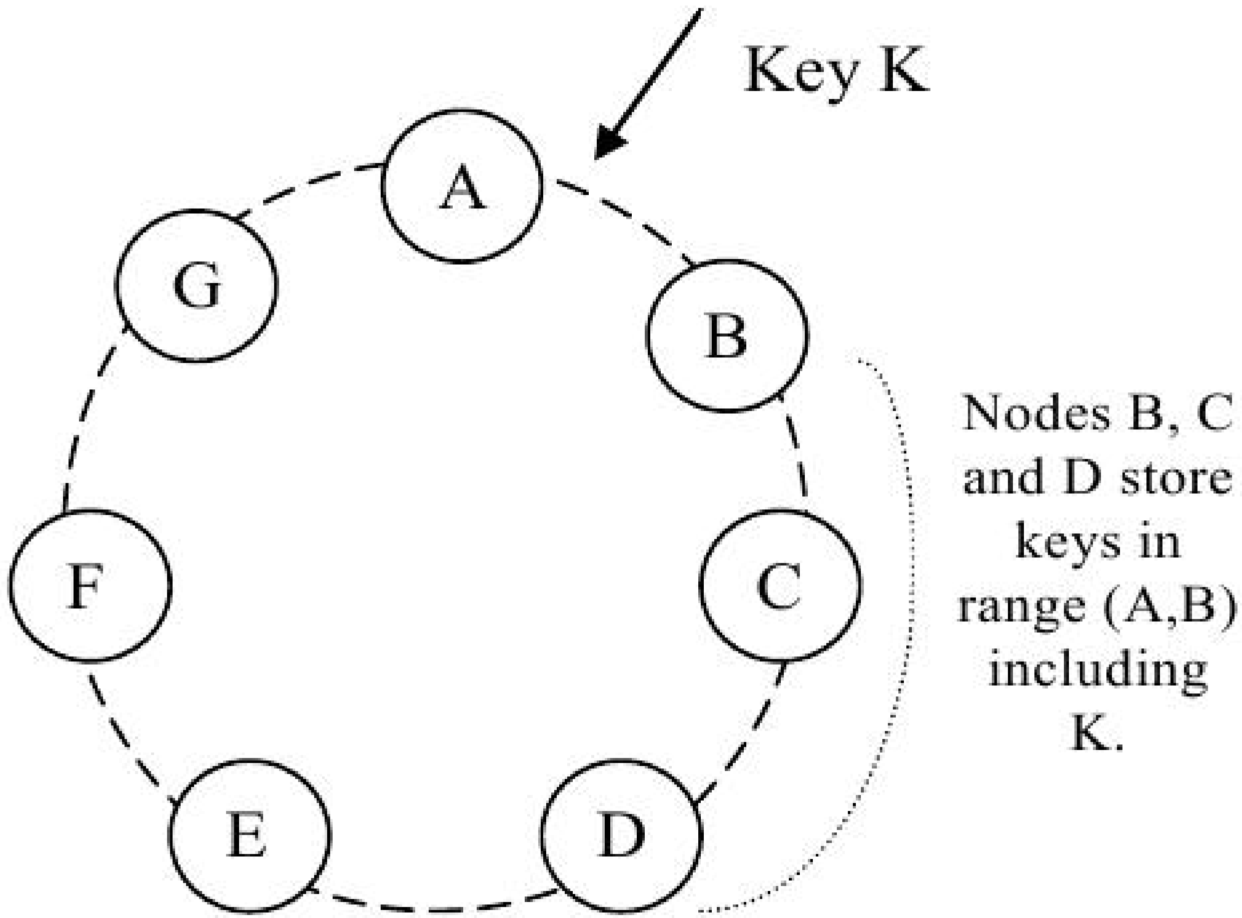
API / Operators

- get(key) returns:
  - one object or a list of objects with conflicting versions
  - a context
- put(key, context, object):
  - find correct locations
  - writes replicas to disk
  - context contains metadata about the object

Partitioning

- variant of consistent hashing similar to Chord
- each node gets keys between its predecessor and itself
- accounts for heterogeneity of nodes using virtual nodes
- the system scales incrementally
- load balancing

Replication



Versioning

- put operation can always be executed
- eventual consistency
- reconciled using vector clocks
- if automatic reconciliation not possible, the system returns a list of versions to the client

Versioning

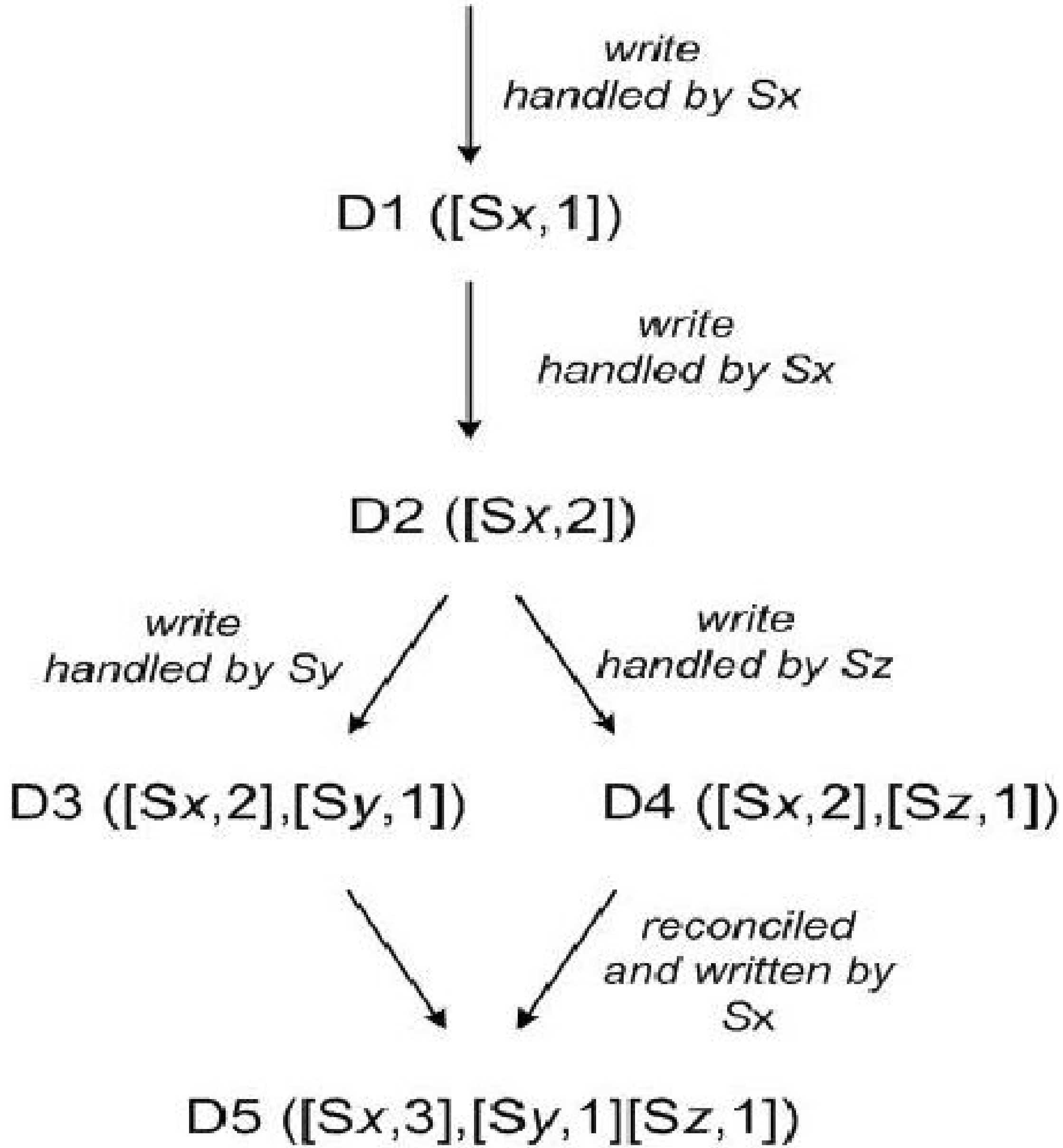


Figure 3: Version evolution of an object over time.

Executing a read / write

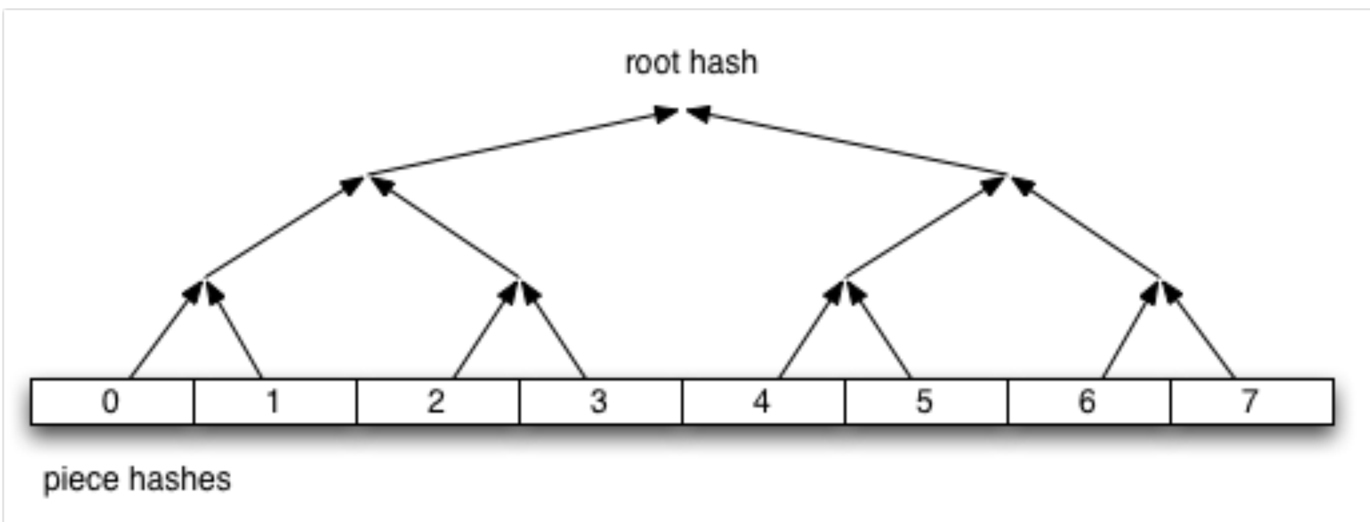
- coordinator node = first node to store the key
- put operation - written to W nodes (w/ the coord. vector clock)
- get operation - coordinator reconciles R versions or sends conflicting versions to the client
- if $R + W > N$ (preference list size) - quorum like system
- usually $R + W < N$ to decrease latency

Hinted Handoff

- the N nodes to which a request is sent are not always the first N nodes in the preference list, if there are failures
- instead a node can temporarily store a key for another node and give it back when that nodes comes back up

Replica Synchronization

- compute Merkle tree for each key range
- periodically check that key ranges are consistent between nodes



Membership

- Ring join / leave propagated via gossip protocol
- Logical partitions avoided using seed nodes
- When a node joins the keys it becomes responsible for are transferred to it by its peers

Summary

Table 1: 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.

Durability vs. Performance

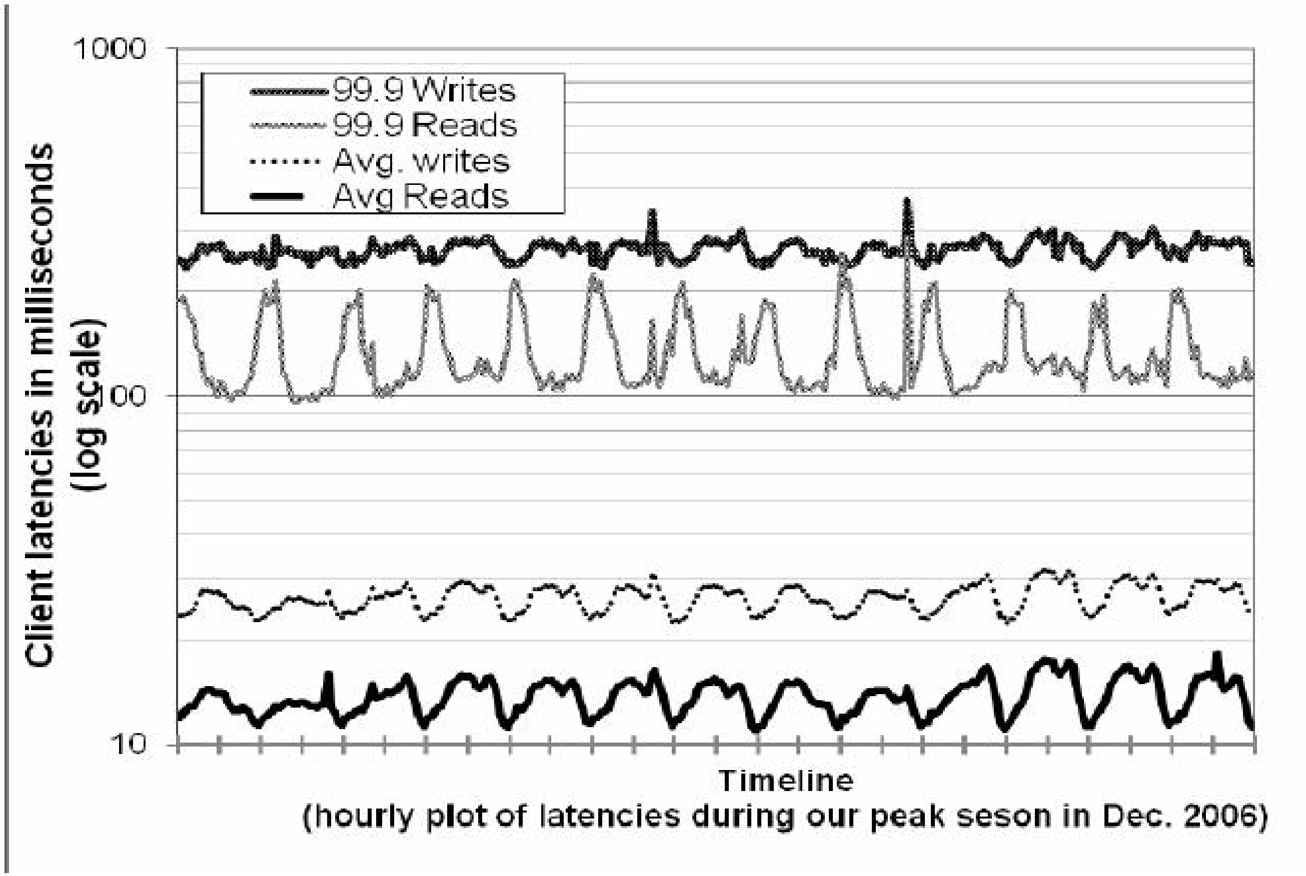


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

Durability vs. Performance

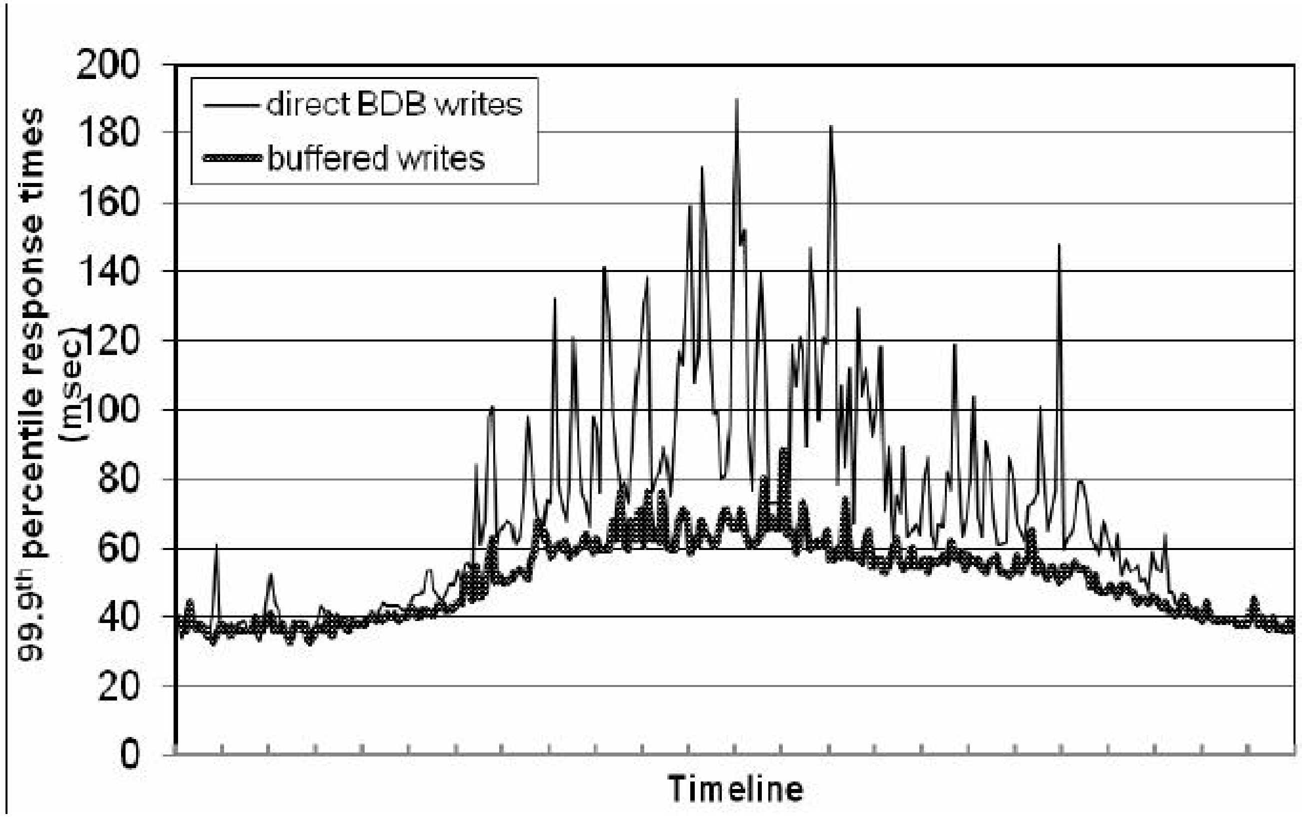


Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

Conclusion

- Combine different techniques to provide a single highly-available system
- An eventually-consistent system could be use in production with demanding applications
- Balancing performance, durability and consistency by tuning parameters N, R, W

Bigtable

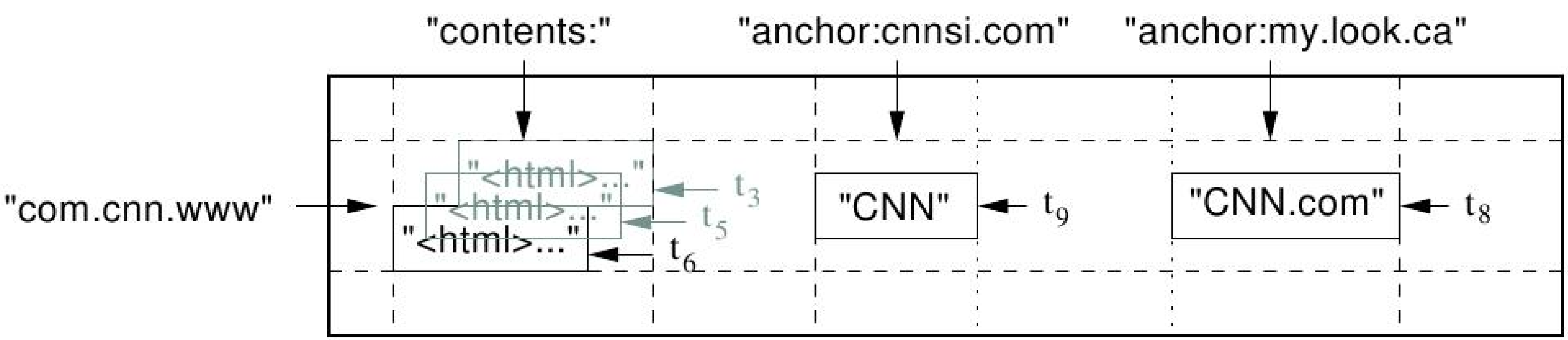
A distributed storage system for structured data

Applications and Requirements

- wide applicability for a variety of systems
- scalability
- high performance
- high availability

Data Model

- key / value pairs structure
- added support for sparse semi-structured data
- key: <row key, column key, timestamp>
- value: uninterpreted array of bytes
- example: Webtable



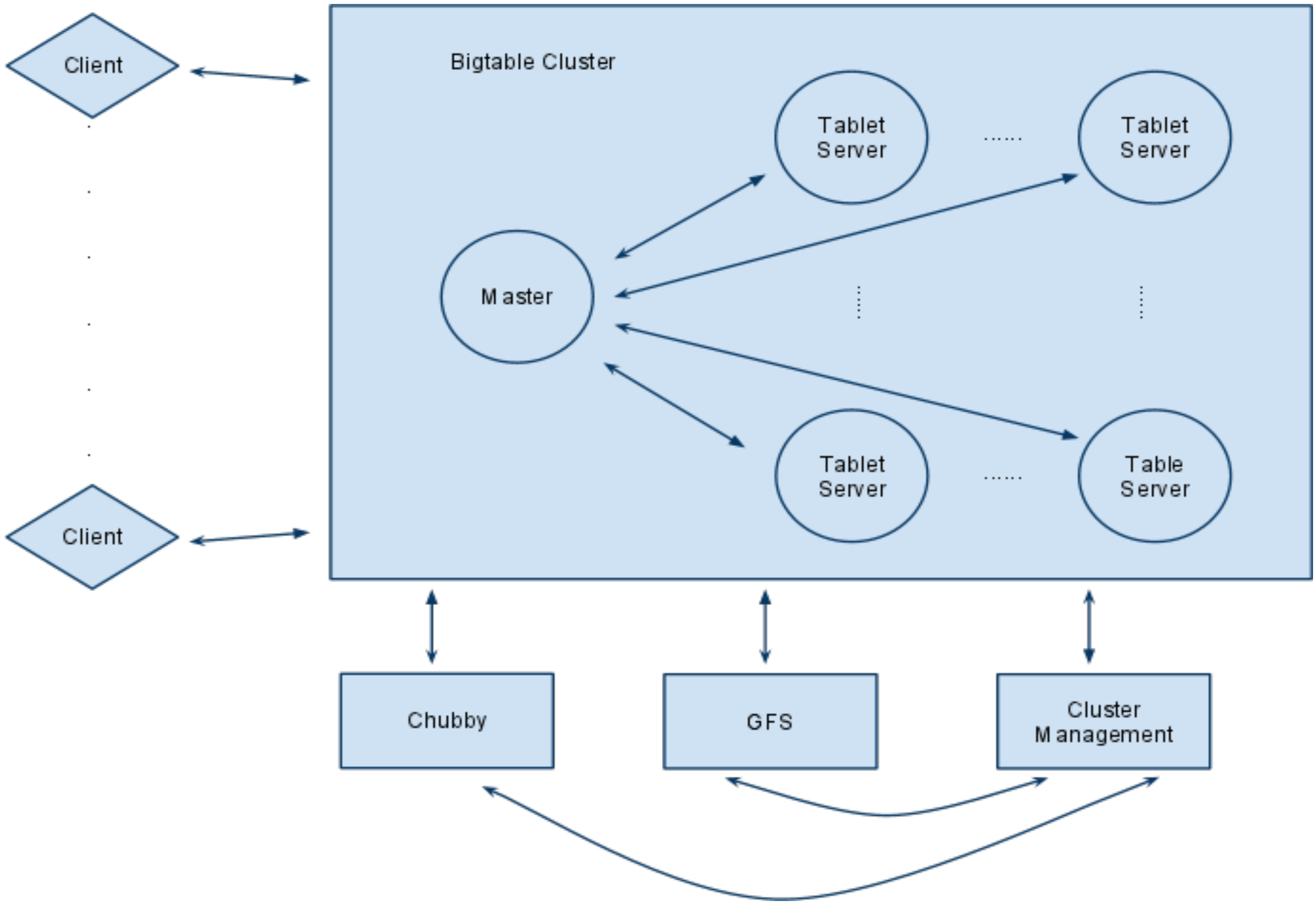
Data Model

- multidimensional map
- lexicographic order by row key
- row access is atomic
- row range dynamically partitioned (tablet)
- can achieve good locality of data
  - e.g. webpages stored by reversed domain
- static column families
- variable columns
- timestamps used to index different versions

API / Operators

- create / delete table
- create / delete column families
- change metadata (cluster / table / column family)
- single-row transactions
- use cells as integer counts
- execute client supplied scripts on the servers

Architecture at a Glance



GFS & Chubby

- GFS
  - Google's distributed file system
  - Scalable, fault-tolerant, with high aggregate performance
  - Store logs, tablets (SSTables)

- Chubby
  - Distributed coordination service
  - Highly available, persistent
  - Data model after directory tree structure of file systems
  - Membership maintenance (the master & tablet servers)
  - Location of root tablet of METADATA table (bootstrap)
  - Schema information, access control lists

The Master

- Detecting addition and expiration of tablet servers
- Assign tablets to tablet servers
- Balancing tablet-server load
- Garbage collection of GFS files
- Handling schema changes

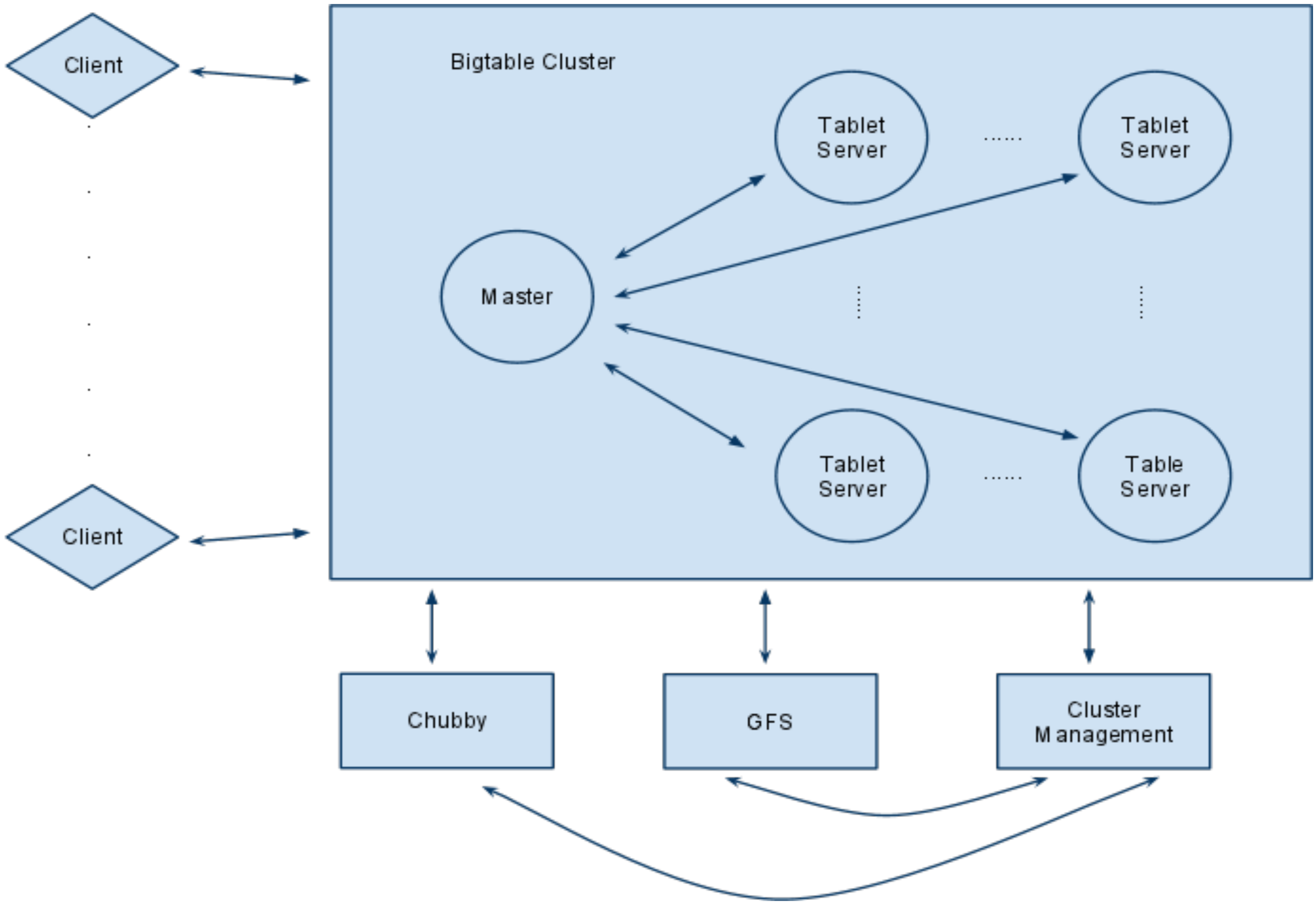
Performance bottleneck?

Tablet Servers

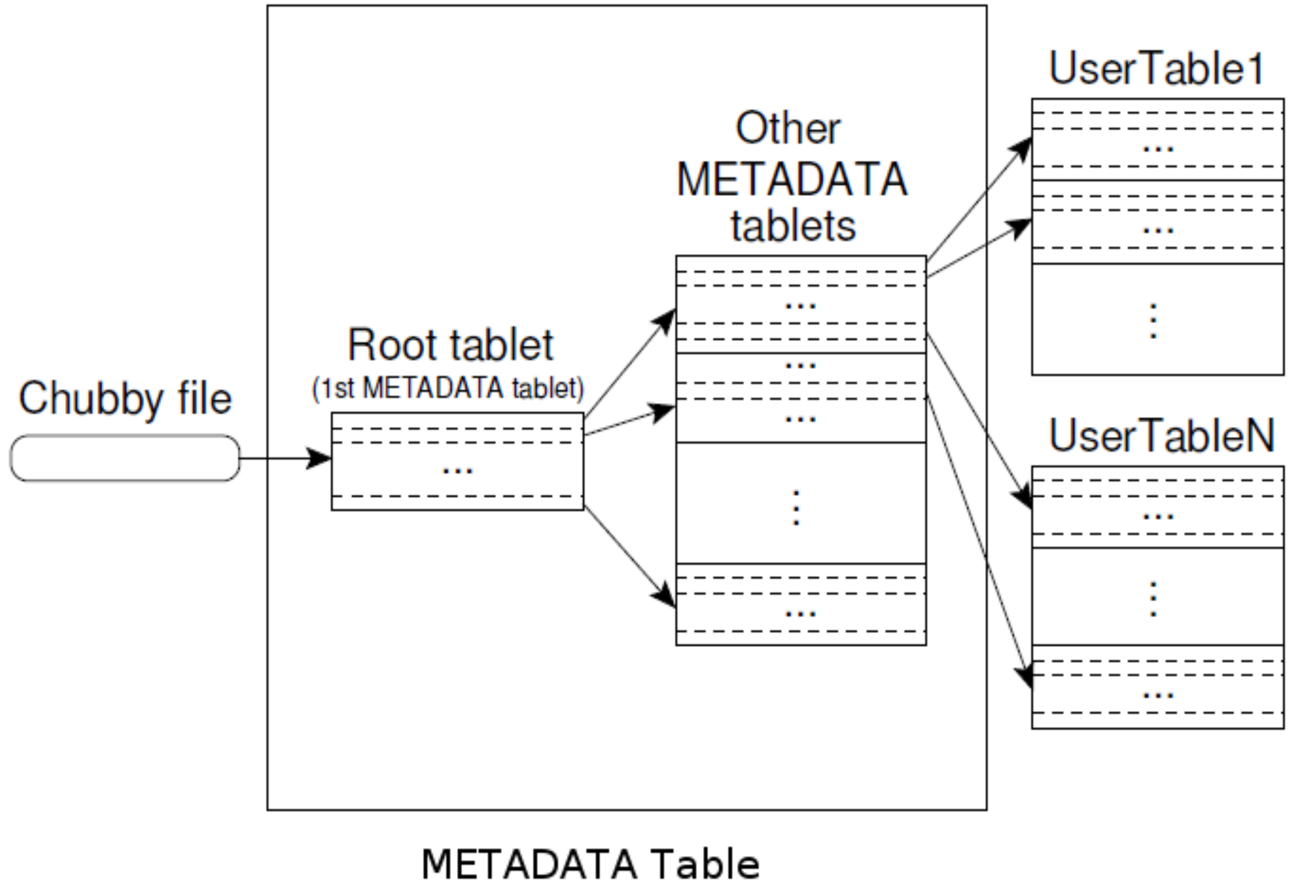
- Manage a set of tablets
  - Handle users' read/write requests for those tablets
  - Split tablets that have grown too large

- Tablet servers' in-memory structures
  - Two-level cache (scan & block)
  - Bloom filters
  - Memtables
  - SSTables (if requested)

Architecture at a Glance



Locate a Tablet: METADATA Table



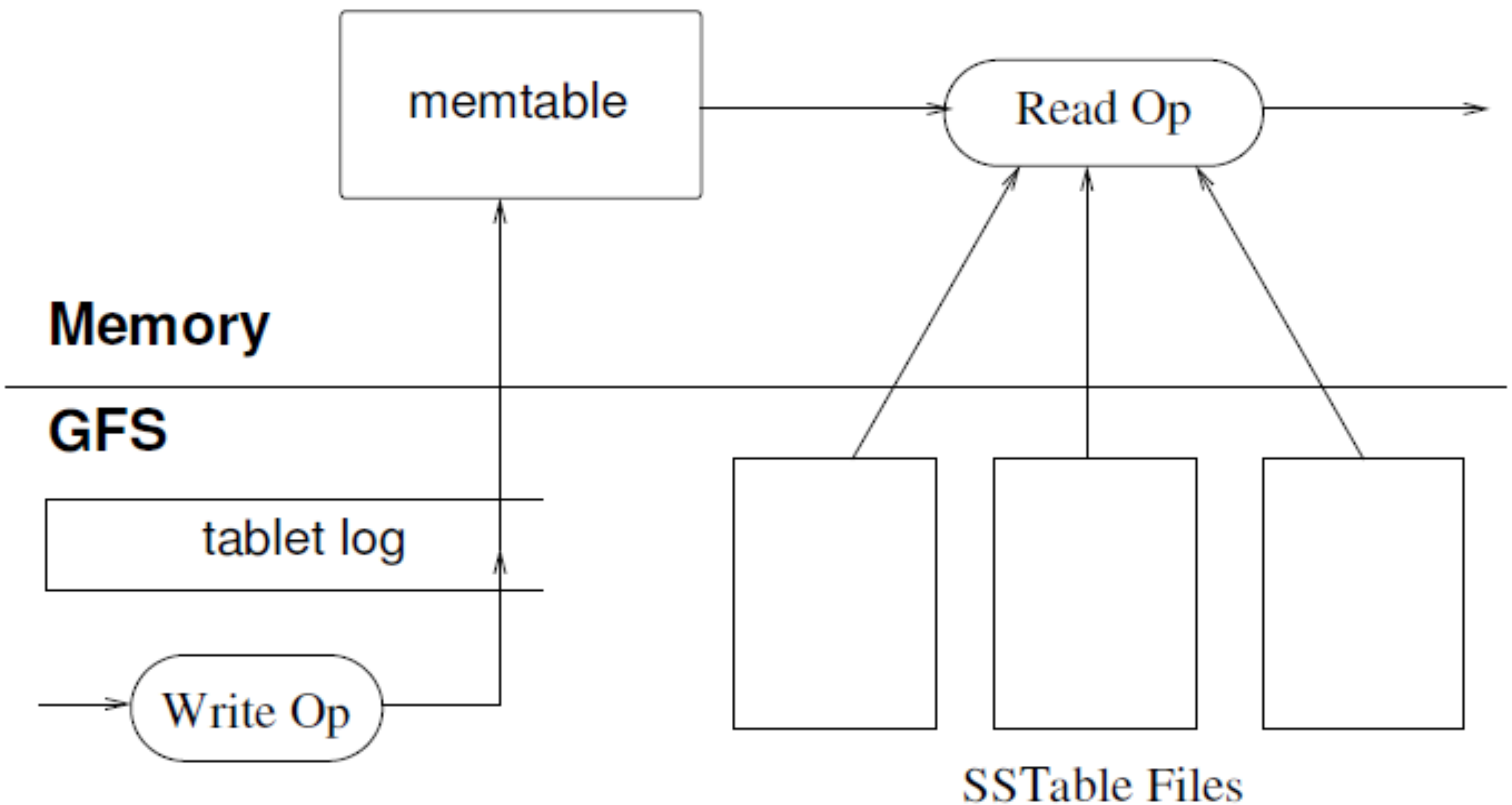
- METADATA table stores tablet locations of user tables
- Row key of METADATA table encodes table ID + end row
- Clients caches tablet locations

Assign a Tablet

- For tablet servers:
  - Each tablet is assigned to one tablet server
  - Each tablet server is managing several tablets

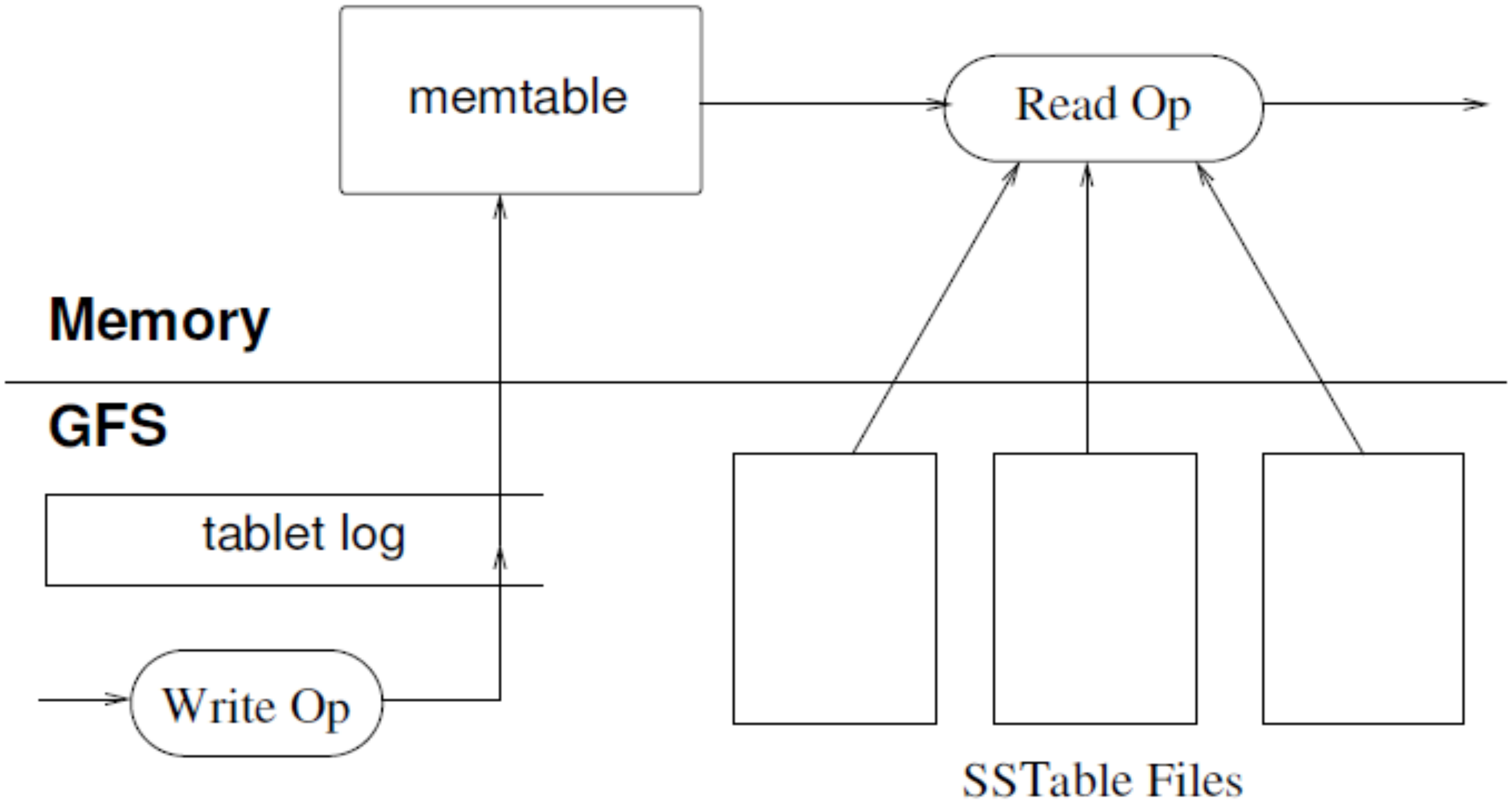
- For the master:
  - Keep track of live tablet servers with Chubby
  - Keep track of current assignment of tablets
  - Assign unassigned tablets to tablet servers considering load balancing issues

Read/Write a Tablet(1)



- Persistent state of a tablet includes a tablet log and SSTables
- Updates are committed to tablet log that stores redo records
- Memtable, a in-memory sorted buffer stores latest updates
- SSTables stores older updates

Read/Write a Tablet(2)



- Write operation
  - Write to commit log, commit it, write to memtable
  - Group commit
- Read operation
  - Read on a merged view of memtable and SSTables

Compactions

- Minor compaction
  - Write the current memtable into a new SSTable on GFS
  - Less memory usage, faster recovery
- Merging compaction
  - Periodically merge a few SSTables and memtable into a new SSTable
  - Simplify merged view for reads
- Major compaction
  - Rewrite all SSTables into exactly one SSTable
  - Reclaim resources used by deleted data
  - Deleted data disappears in a timely fashion

Optimizations(1)

- Locality groups
  - Group column families typically accessed together
  - Generate a separate SSTable for each locality group
  - Specify in-memory locality groups (METADATA:location)
  - More efficient reads
- Compression
  - Control if SSTables for a locality group are compressed
  - Speed VS space, network transmission cost
  - Locality has influences over compression rate

Optimizations(2)

- Two-level cache for read performance
  - Scan cache: caches accessed key-value pairs
  - Block cache: caches accessed SSTables blocks
- Bloom filters
  - Created for SSTables in certain locality groups
  - Identify whether SSTable might contain data queried
- Commit-log implementation
  - Single commit log per tablet servers
  - Co-mingle mutations for different tablets
  - Decrease number of log files
  - Complicate recovery process

Optimizations(3)

- Speeding up tablet recovery
  - Two minor compaction when moving tablet between tablet servers
  - Reduce uncompacted state in commit log

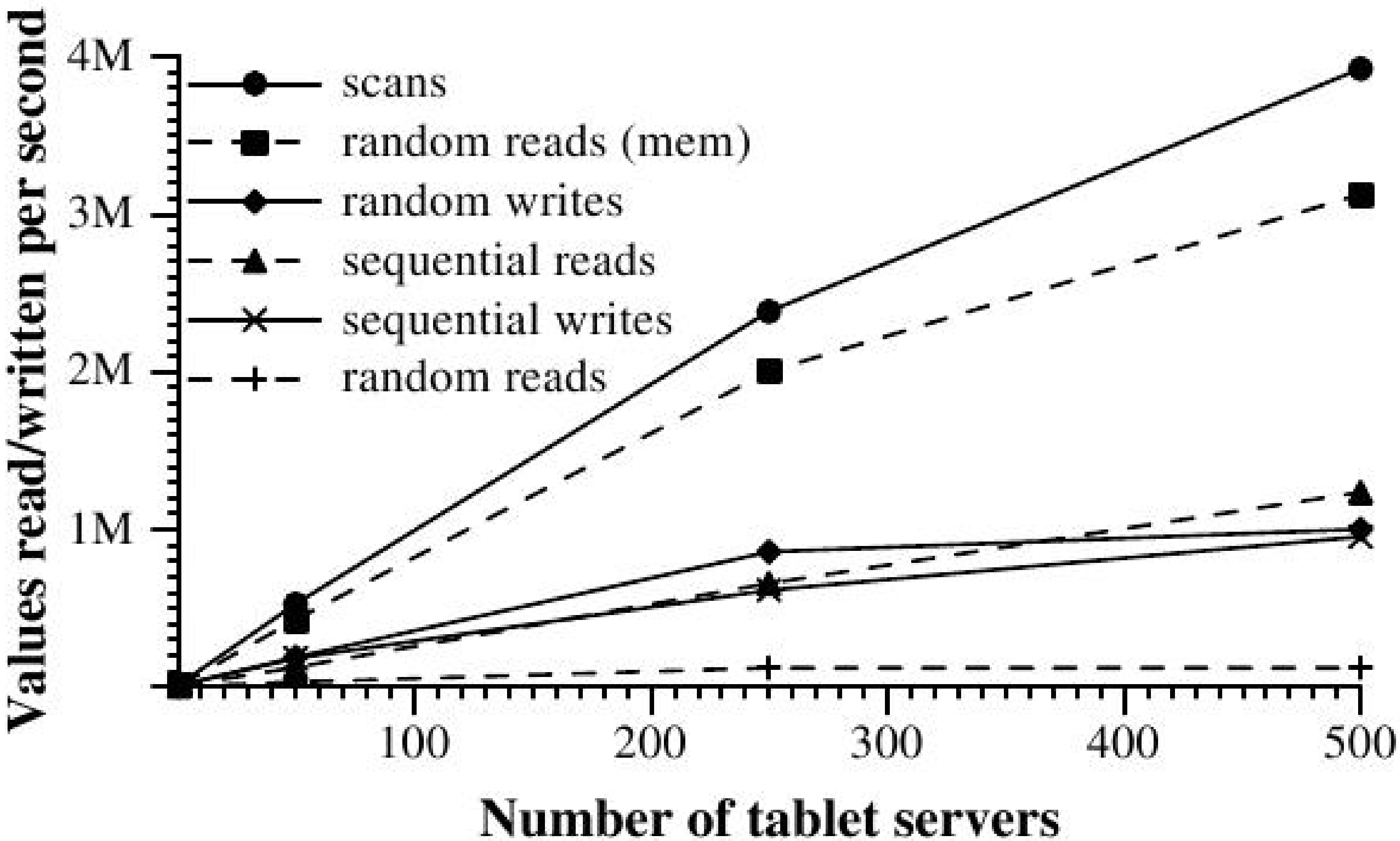
- Exploiting immutability
  - SSTables are immutable
  - No synchronization for reads
  - Writes generate new SSTables
  - Copy-on-write for memtables
  - Tablets are allowed to share SSTables

Evaluation

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Number of operations per second per tablet server

Evaluation



Aggregate number of operations per second

Applications

Click Table
Summary Table

One table storing raw imagery, served from disk

User data
Row: userid
Each group can add their own user column

Lessons Learned

1. many types of failures, not just network partitions
2. add new features only if needed
3. improve the system by careful monitoring
4. keep the design simple

Conclusion

- Bigtable is used in production code since April 2005
- used extensively by several Google projects
- "unusual interface"
  - compared to the traditional relational model
- It has empirically shown its performance, availability and elasticity

Dynamo vs. Bigtable