Big Table – A Distributed Storage System For Data

OSDI 2006

Fay Chang, Jeffrey Dean, Sanjay Ghemawat et.al.

Presented by

Rahul Malviya

Why BigTable ?

- Lots of (semi-)structured data at Google -
  - URLs: Contents, crawl metadata(when, response code), links, anchors
  - Per-user Data: User preferences settings, recent queries, search results
  - Geographical locations: Physical entities – shops, restaurants, roads
- Scale is large
  - Billions of URLs, many versions/page - 20KB/page
  - Hundreds of millions of users, thousands of q/sec – Latency requirement
  - 100+TB of satellite image data

Why Not Commercial Database ?

- Scale too large for most commercial databases
- Even if it weren't, cost would be too high
- Building internally means low incremental cost
  - System can be applied across many projects used as building blocks.
- Much harder to do low-level storage/network transfer optimizations to help performance significantly.
  - When running on top of a database layer.

Target System

- System for managing all the state in crawling for building indexes.
  - Lot of different asynchronous processes to be able to continuously update the data they are responsible for in this large state.
  - Many different asynchronous process reading some of their input from this state and writing updated values to their output to the state.
  - Want to access to the most current data for a url at any time.

Goals

- Need to support:
  - Very high read/write rates (millions of operations per second) – Google Talk
  - Efficient scans over all or interesting subset of data
    - Just the crawl metadata or all the contents and anchors together.
  - Efficient joins of large 1-1 and 1-* datasets
    - Joining contents with anchors is pretty big computation
- Often want to examine data changes over time
  - Contents of web page over multiple crawls
    - How often web page changes so you know how often to crawl ?

BigTable

- Distributed Multilevel Map
- Fault-tolerant, persistent
- Scalable
    - 1000s of servers
    - TB of in-memory data
    - Peta byte of disk based data
    - Millions of read/writes per second, efficient scans
- Self-managing
    - Servers can be added/removed dynamically
    - Servers adjust to the load imbalance

Background: Building Blocks

The building blocks for the BigTable are :

- Google File System – Raw Storage.
- Scheduler – Schedules jobs onto machines.
- Lock Service – Chubby distributed lock manager.
- Map Reduce – Simplified large scale data processing.

Background: Building Blocks Cont..

BigTable uses of building blocks -

- GFS – Stores persistent state.
- Scheduler – Schedules jobs involved in BigTable serving.
- Lock Service – Master election, location bootstrapping.
- Map Reduce – Used to read/write Big Table data.
  - BigTable can be and/or output for Map Reduce computations.

Google File System

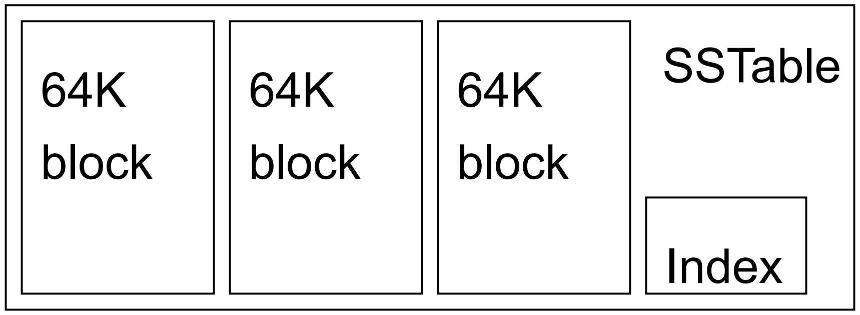
- Large-scale distributed “filesystem”
- Master: responsible for metadata
- Chunk servers: responsible for reading and writing large chunks of data
- Chunks replicated on 3 machines, master responsible for ensuring replicas exist.

Chubby Lock Service

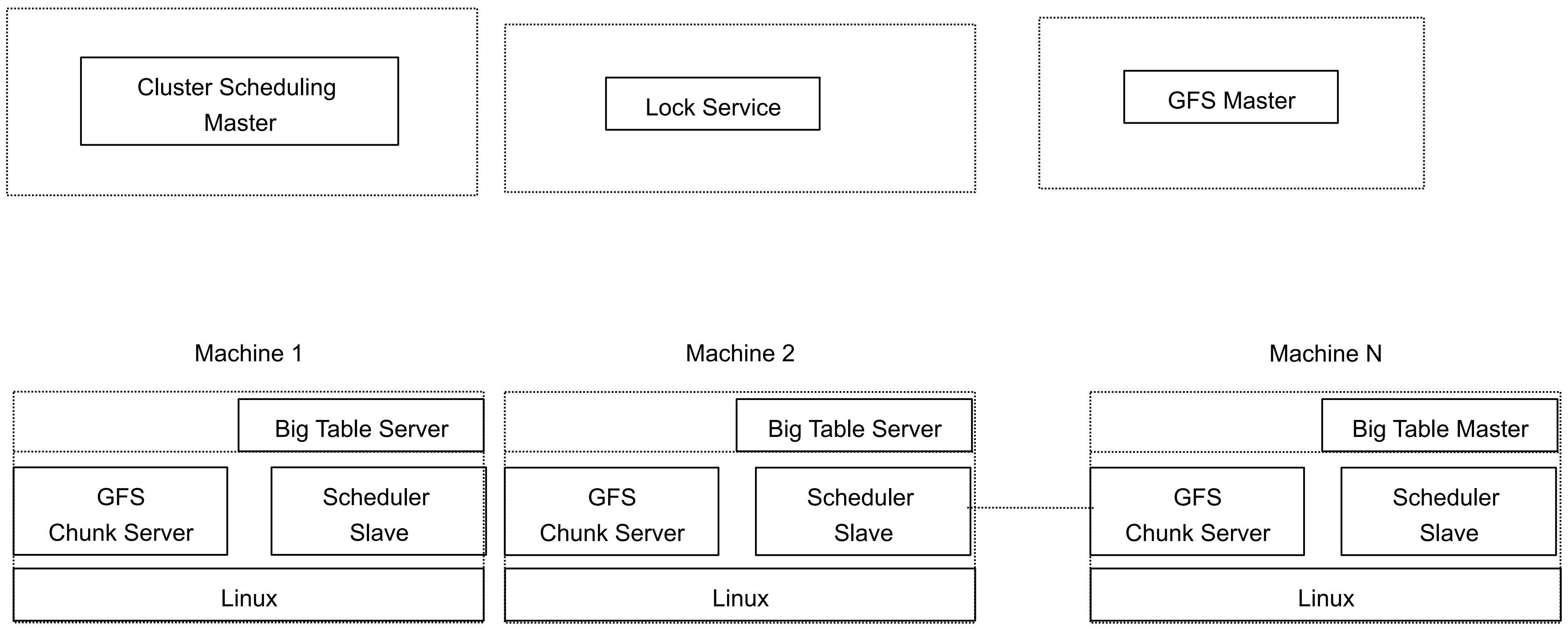
- Name space consists of directories and small files which are used as locks.
- Read/Write to a file are atomic.
- Consists of 5 active replicas – 1 is elected master and serves requests.
- Needs a majority of its replicas to be running for the service to be alive.
- Uses Paxos to keep its replicas consistent during failures.

SS Table

- Immutable, sorted file of key-value pairs
- Chunks of data plus an index
  - Index is of block ranges, not values

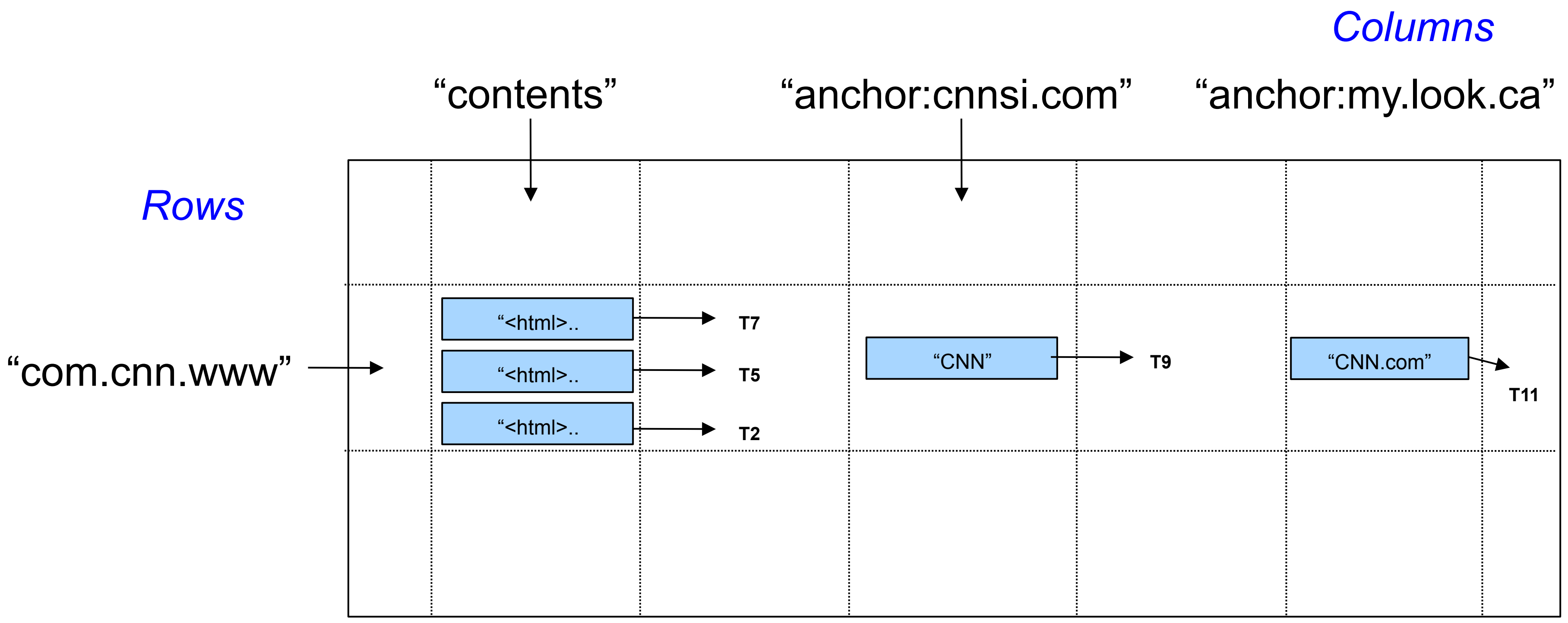


Typical Cluster



Basic Data Model

- Distributed multi-dimensional sparse map
  - (row, column, timestamp) -> cell contents



Rows

- Name is an arbitrary string.
  - Access to data in a row is atomic.
  - Row creation is implicit upon storing data.
  - Transactions within a row
- Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines.
- Does not support relational model
  - No table wide integrity constants
  - No multi row transactions

Columns

- Column oriented storage.
- Focus on reads from columns.
- Columns has two-level name structure:
  - family:optional_qualifier
- Column family
  - Unit of access control
  - Has associated type information
- Qualifier gives unbounded columns
  - Additional level of indexing, if desired

Timestamps

- Used to store different versions of data in a cell
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Look up options:
  - “Return most recent K values”
  - “Return all values in timestamp range(on all values)”
- Column families can be marked with attributes
  - “Only retain most recent K values in a cell”
  - “Keep values until they are older than K seconds”

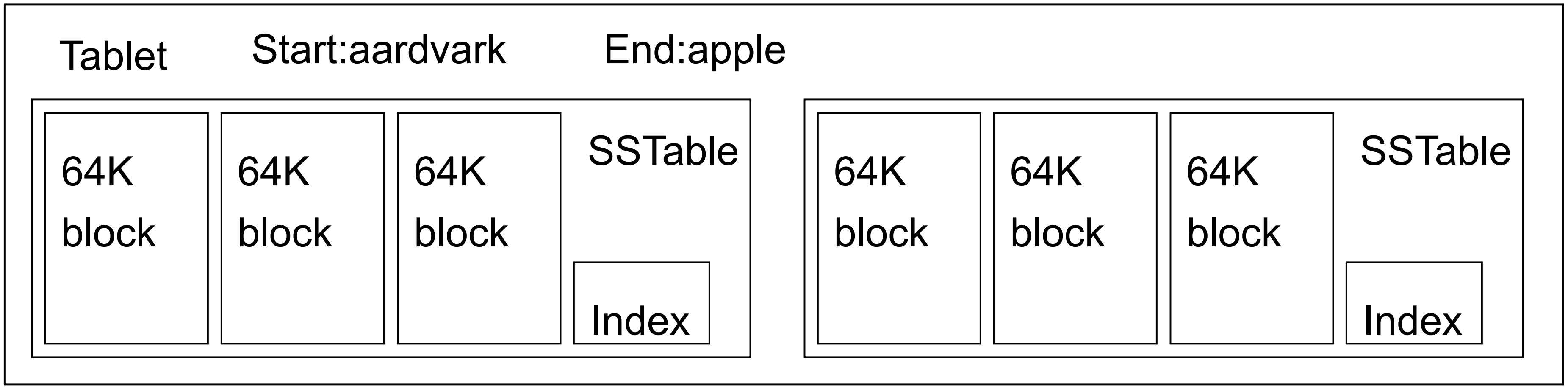
Tablets

The way to get data to be spread out in all machines in serving cluster

- Large tables broken into tablets at row boundaries.
  - Tablet holds contiguous range of rows
    - Clients can often chose row keys to achieve locality
  - Aim for 100MB or 200MB of data per tablet
- Serving cluster responsible for 100 tablets – Gives two nice properties -
  - Fast recovery:
    - 100 machines each pick up 1 tablet from failed machine
  - Fine-grained load balancing:
    - Migrate tablets away from overloaded machine
    - Master makes load balancing decisions

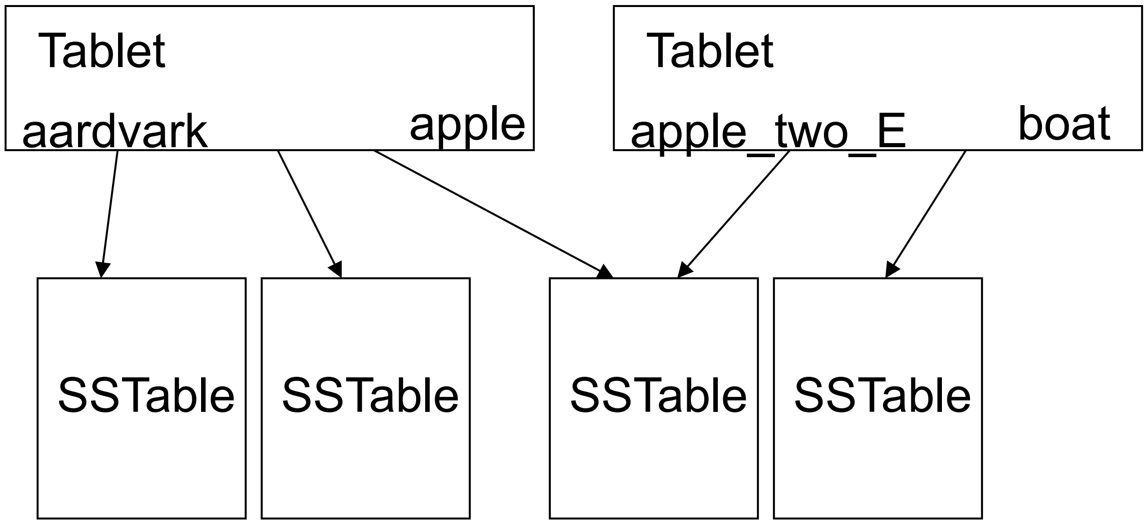
Tablets contd...

- Contains some range of rows of the table
- Built out of multiple SSTables

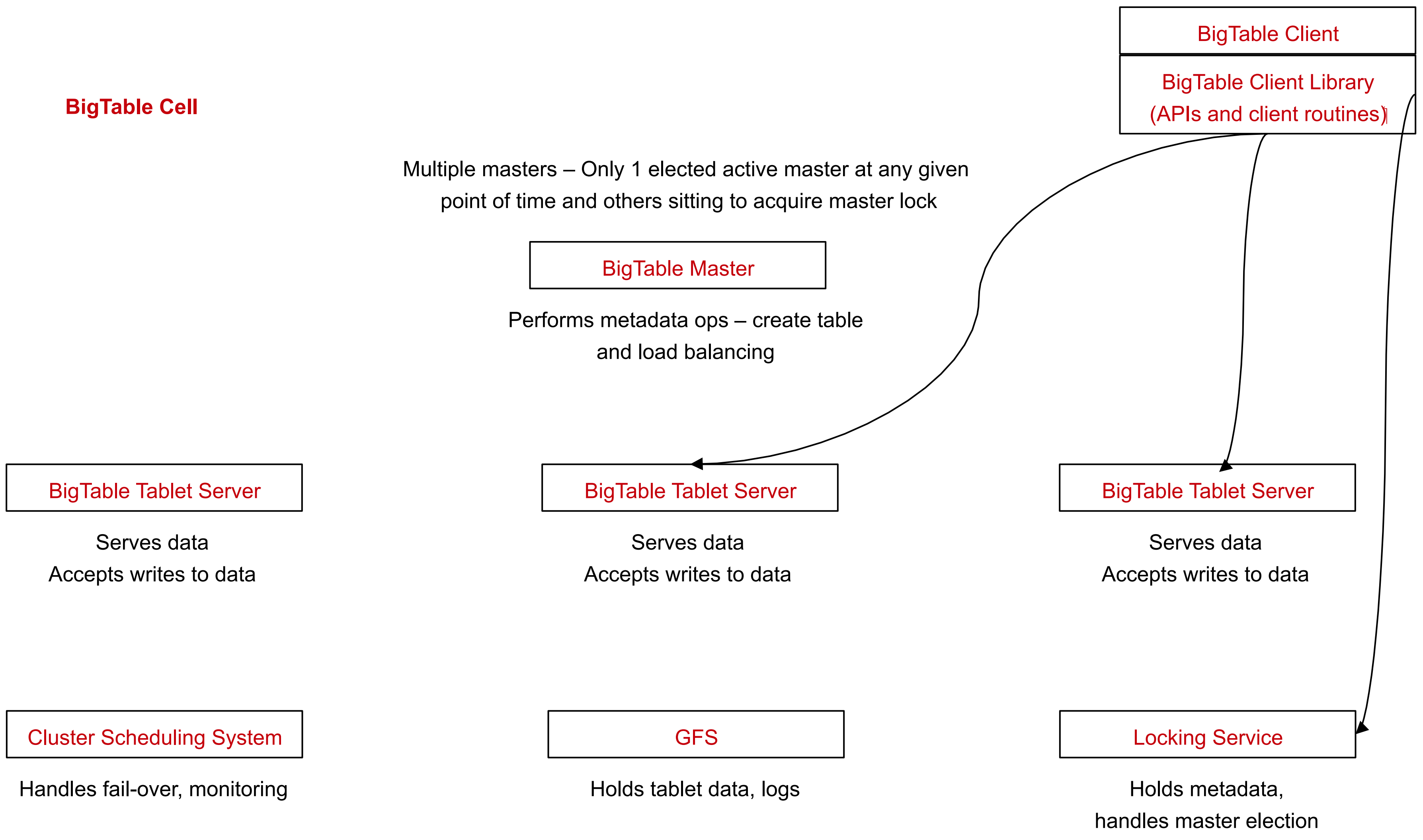


Table

- Multiple tablets make up the table
- SSTables can be shared
- Tablets do not overlap, SSTables can overlap



System Structure



BigTable Cell
BigTable Client
BigTable Client Library
(APIs and client routines)
Multiple masters – Only 1 elected active master at any given point of time and others sitting to acquire master lock
BigTable Master
Performs metadata ops – create table and load balancing
BigTable Tablet Server
Serves data
Accepts writes to data
BigTable Tablet Server
Serves data
Accepts writes to data
BigTable Tablet Server
Serves data
Accepts writes to data
Cluster Scheduling System
Handles fail-over, monitoring
GFS
Holds tablet data, logs
Locking Service
Holds metadata, handles master election

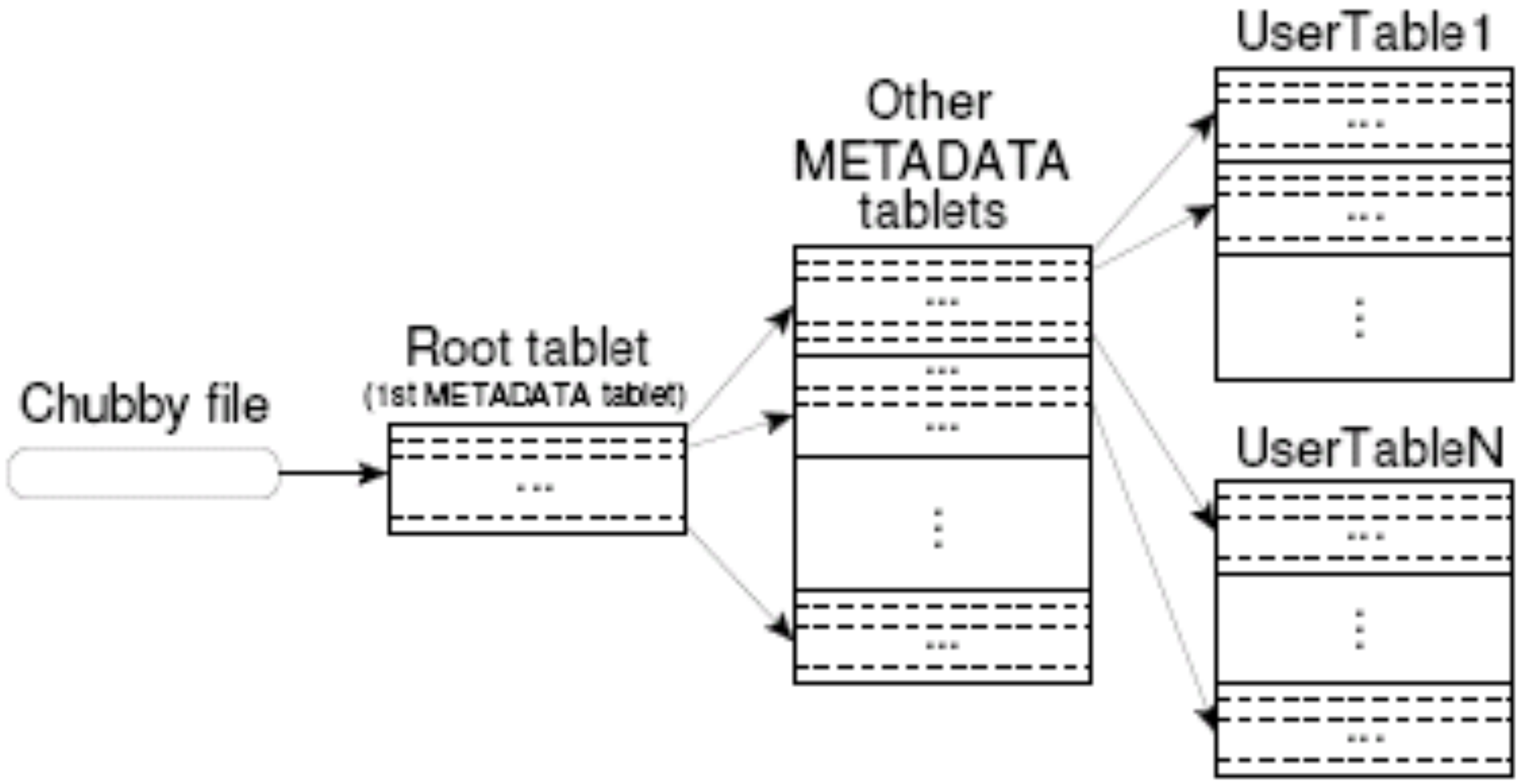
Locating Tablets

- Since tablets move around from server to server, given a row, how do clients find a right machine ?
  - Tablet property – startrowindex and endrowindex
  - Need to find tablet whose row range covers the target row
- One approach: Could use 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 (contd ..)

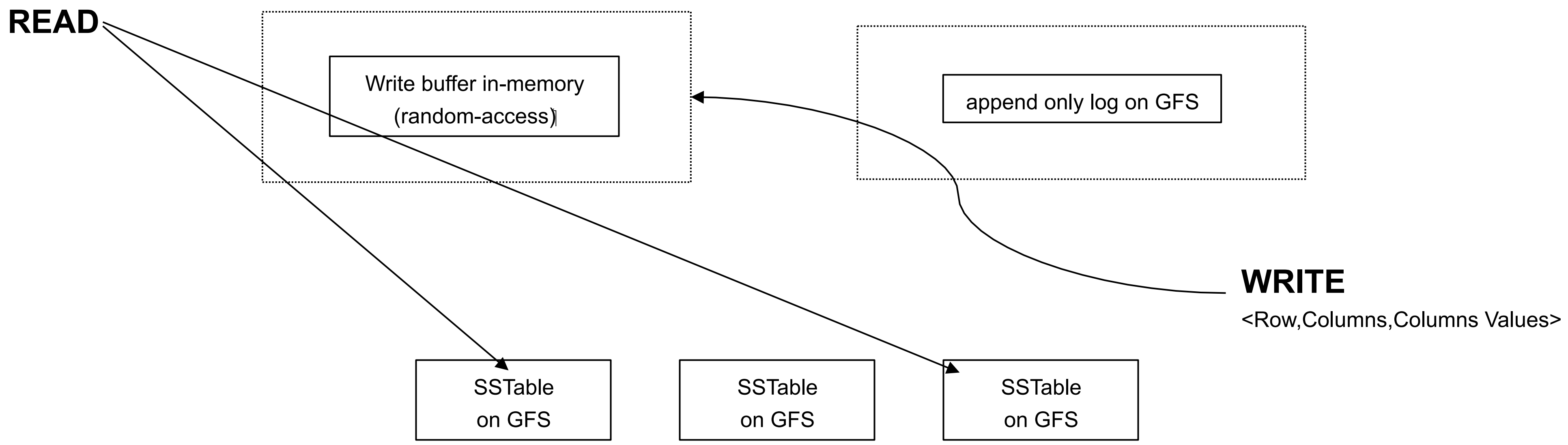
- Three level hierarchical lookup scheme for tablets
  - Location is ip port of relevant server.
  - 1st level: bootstrapped from lock service, points to the owner of META0
  - 2nd level: Uses META0 data to find owner of appropriate META1 tablet.
  - 3rd level: META1 table holds locations of tablets of all other tables
    - META1 itself can be split into multiple tablets

Locating tablets contd..



Tablet Representation

Given machine is typically servicing 100s of tablets



Assorted table to map string-string.

Tablet Assignment

- 1 Tablet => 1 Tablet server
- Master keeps tracks of set of live tablet serves and unassigned tablets.
- Master sends a tablet load request for unassigned tablet to the tablet server.
- BigTable uses Chubby to keep track of tablet servers.
- On startup a tablet server –
  - It creates, acquires an exclusive lock on uniquely named file on Chubby directory.
  - Master monitors the above directory to discover tablet servers.
- Tablet server stops serving -
  - Its tablets if its loses its exclusive lock.
  - Tries to reacquire the lock on its file as long as the file still exists.

Tablet Assignment Contd...

- If the file no longer exists -
  - Tablet server not able to serve again and kills itself.
- If tablet server machine is removed from cluster -
  - Causes tablet server termination
  - It releases it lock on file so that master will reassign its tablets quickly.
- Master is responsible for finding when tablet server is no longer serving its tablets and reassigning those tablets as soon as possible.
- Master detects by checking periodically the status of the lock of each tablet server.
  - If tablet server reports the loss of lock
  - Or if master could not reach tablet server after several attempts.

Tablet Assignment Contd...

- Master tries to acquire an exclusive lock on server's file.
  - If master is able to acquire lock, then chubby is alive and tablet server is either dead or having trouble reaching chubby.
  - If so master makes sure that tablet server never can server again by deleting its server file.
  - Master moves all the assigned tablets into set of unassigned tablets.
- If Chubby session expires
  - Master kills itself.
- When master is started -
  - It needs to discover the current tablet assignment.

Tablet Assignment Contd...

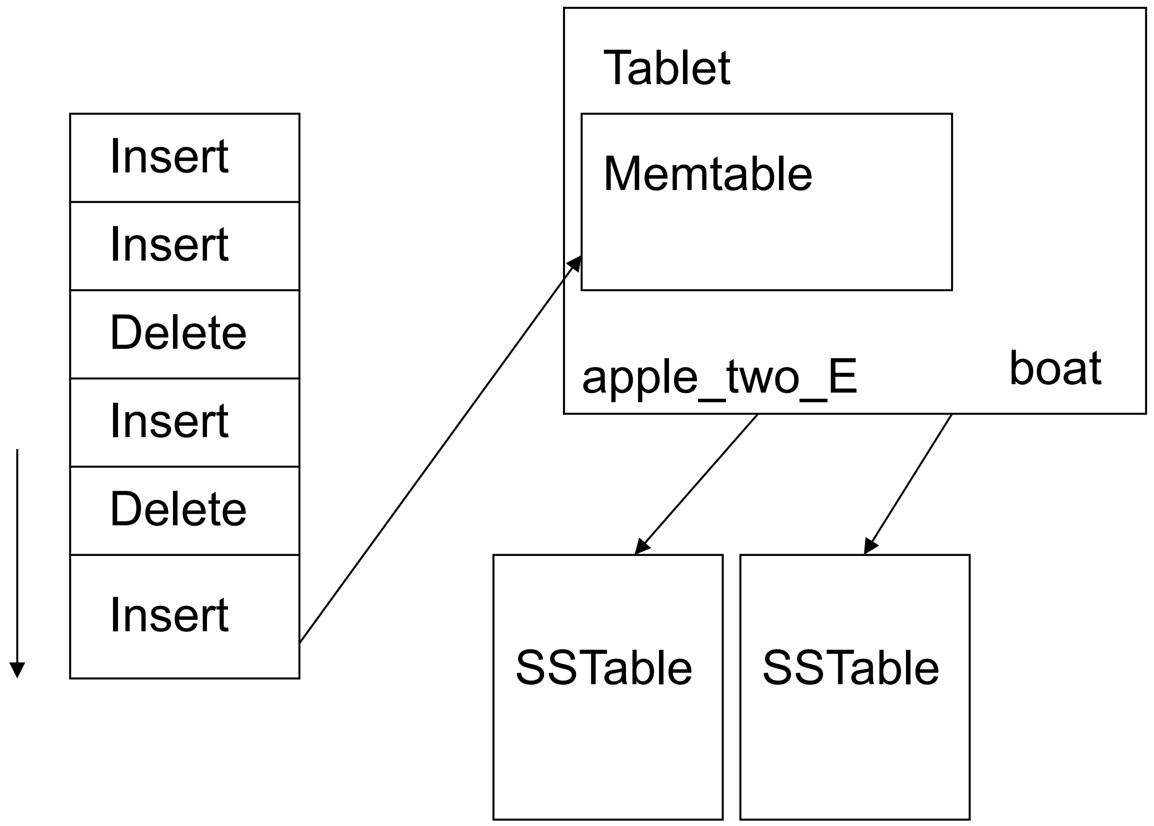
- Master startup steps -
  - Grabs unique master lock in Chubby.
  - Scans the server directory in Chubby.
  - Communicates with every live tablet server
  - Scans METADATA table to learn set of tablets.

Tablet Serving

- Updates committed to a commit log
- Recently committed updates are stored in memory – memtable
- Older updates are stored in a sequence of SSTables.
- Recovering tablet -
  - Tablet server reads data from METADATA table.
  - Metadata contains list of SSTables and pointers into any commit log that may contain data for the tablet.
  - Server reads the indices of the SSTables in memory
  - Reconstructs the memtable by applying all of the updates since redo points.

Editing a table

- Mutations are logged, then applied to an in-memory version
- Logfile stored in GFS



Compactions

When in-memory is full

- Minor compaction -
  - When in-memory state fills up, pick tablet with most data and write contents to SSTables stored in GFS
    - Separate file for each locality group for each tablet.
- Merging Compaction -
  - Periodically compact all SSTables for tablet into new base SSTables on GFS
    - Storage reclaimed from deletions at this point.
- Major Compaction -
  - Merging compaction that results in only one SS table.
  - No deleted records, only sensitive live data.

Refinements

The implementation described previously require a number of refinements to achieve the high performance, availability, and reliability.

They are :

- Locality Groups
- Compression
- Caching for read performances
- Bloom Filters
- Commit-log implementation
- Speeding up tablet recovery
- Exploiting immutability

Refinements

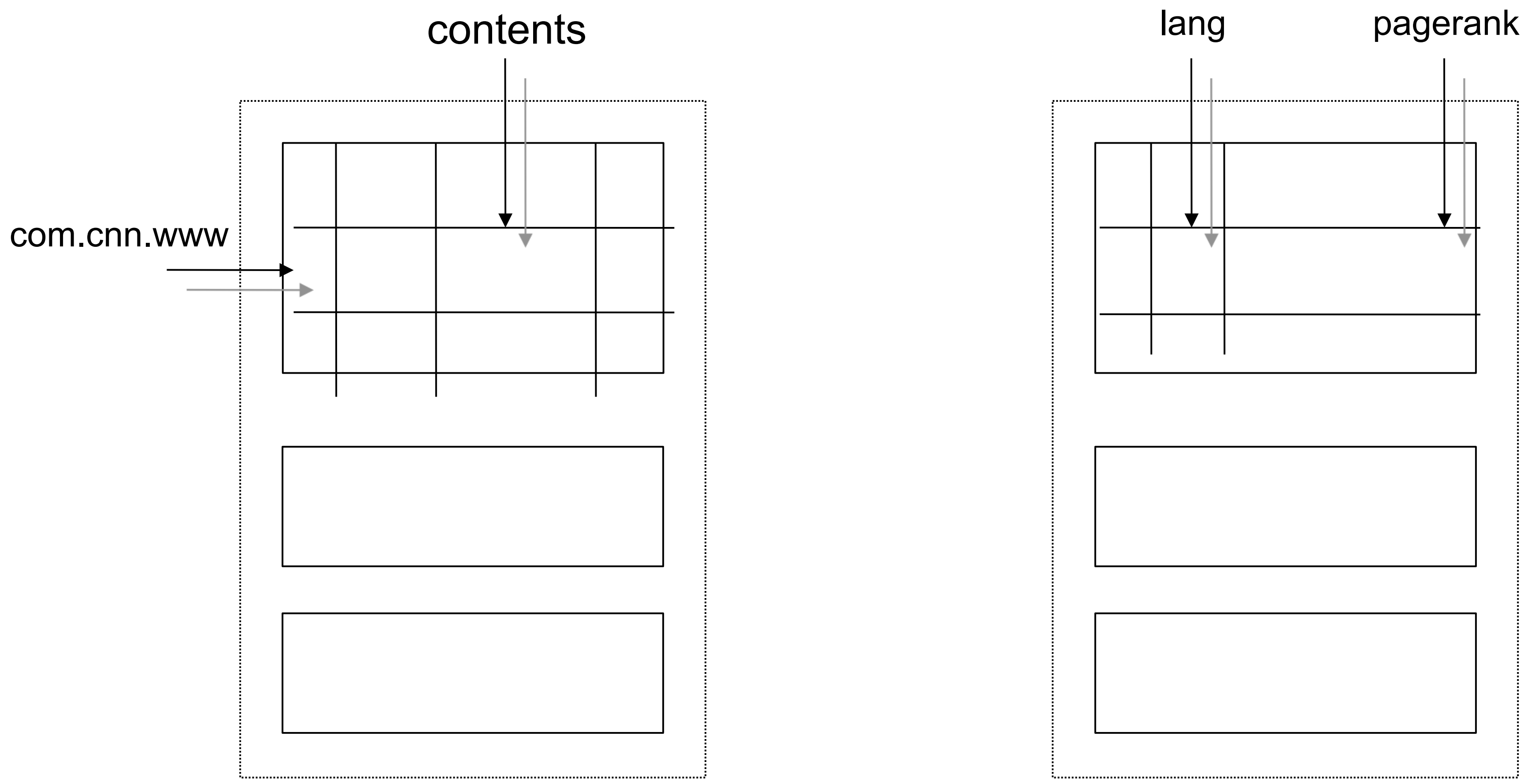
Locality Groups

Storage optimization to be able to access subset of the data

- We partition certain kind of data from other kind of data in the underlying storage so that when you process the data if you want to scan over a subset of data you would be able to do that.
- Columns families can be assigned to a locality group
  - Used to organize underlying storage representation for performance
    - Scans over one locality group are O(bytes_in_locality_group), not O (bytes_in_table)
- Data in locality group can be explicitly memory mapped

Refinements

Locality Groups Cont..



Refinements

Compression

- Clients can control whether or not the SSTable for a locality group are compressed, and if so, which compression format is used.
- The user specified compression format is then applied to the SSTable lock whose size is controllable via a locality group specific tuning parameter.
- Although we lose some space by compressing each block separately, we benefit in that small portions of an SSTable can be read without decompressing the entire file.
- In our Webtable example we use compression scheme to store the Web page contents.

Refinements

Caching for read performance

- To improve read performance, tablet servers use two levels of caching.
- The Scan Cache is a higher-level cache that caches the key-value pairs returned by the SSTable interface to the tablet server code.
- The Block Cache is a lower-level cache that caches the SSTable blocks that were read from the GFS.
- The Scan cache is most useful for the applications that tends to read the same data repeatedly and The Block Cache is useful for the application that tend to read data that is close to the data they recently read for example sequential reads, or random reads.

Refinements

Bloom Filters

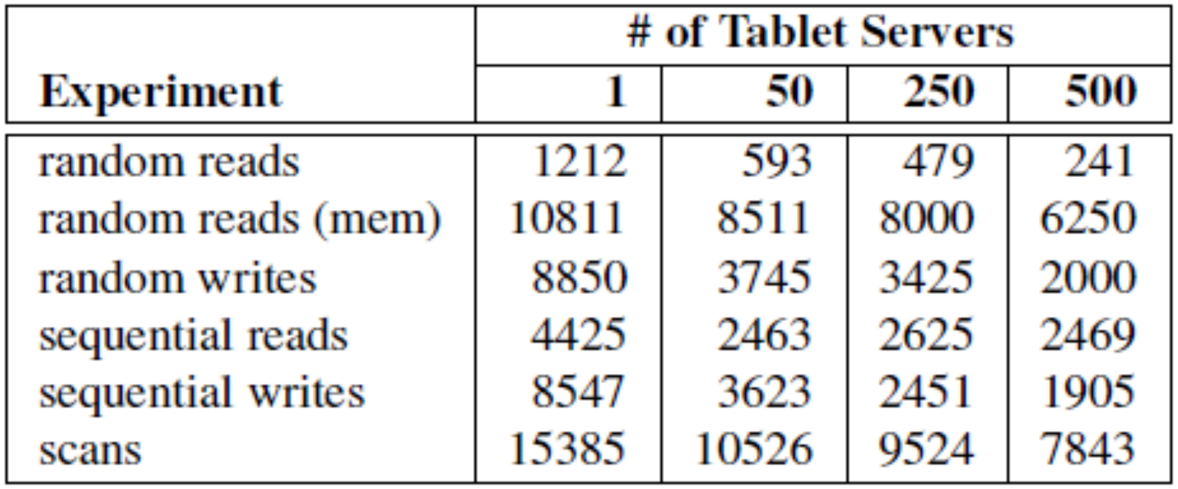
- A read operation has to read from all SSTable that make the state of a tablet. If these SSTables are not in the memory that we may end up with many disk accesses.
- We reduce the disk accesses by allowing client to specify that a Bloom filter should be created for SSTables in particular locality group.
- A Bloom filter allows us to ask whether an SSTable might contain any data for the specified row/column pair. This drastically reduces the number of disk seeks required for read operations.
- Use of Bloom filters also implies that most lookups for non-existent rows or columns do not need to touch the disk.

Performance Evaluation

- We set up a Bigtable cluster with N tablet servers to measure the performance and scalability of Bigtable as N is varied. The tablet servers were configured to use 1 GB of memory and to write to a GFS cell consisting of 1786 machines with two 400 GB IDE hard drives each. N client machines generated the Bigtable load used for these tests.
- Each machine had two dual-core Opteron 2 GHz chips, enough physical memory to old the working set of all running processes, and a single gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.
- The tablet servers and master, test clients, and GFS servers all ran on the same set of machines. Every machine ran a GFS server. Some of the machines also ran either a tablet server, or a client process, or processes from other jobs that were using the pool at the same time as these experiments.

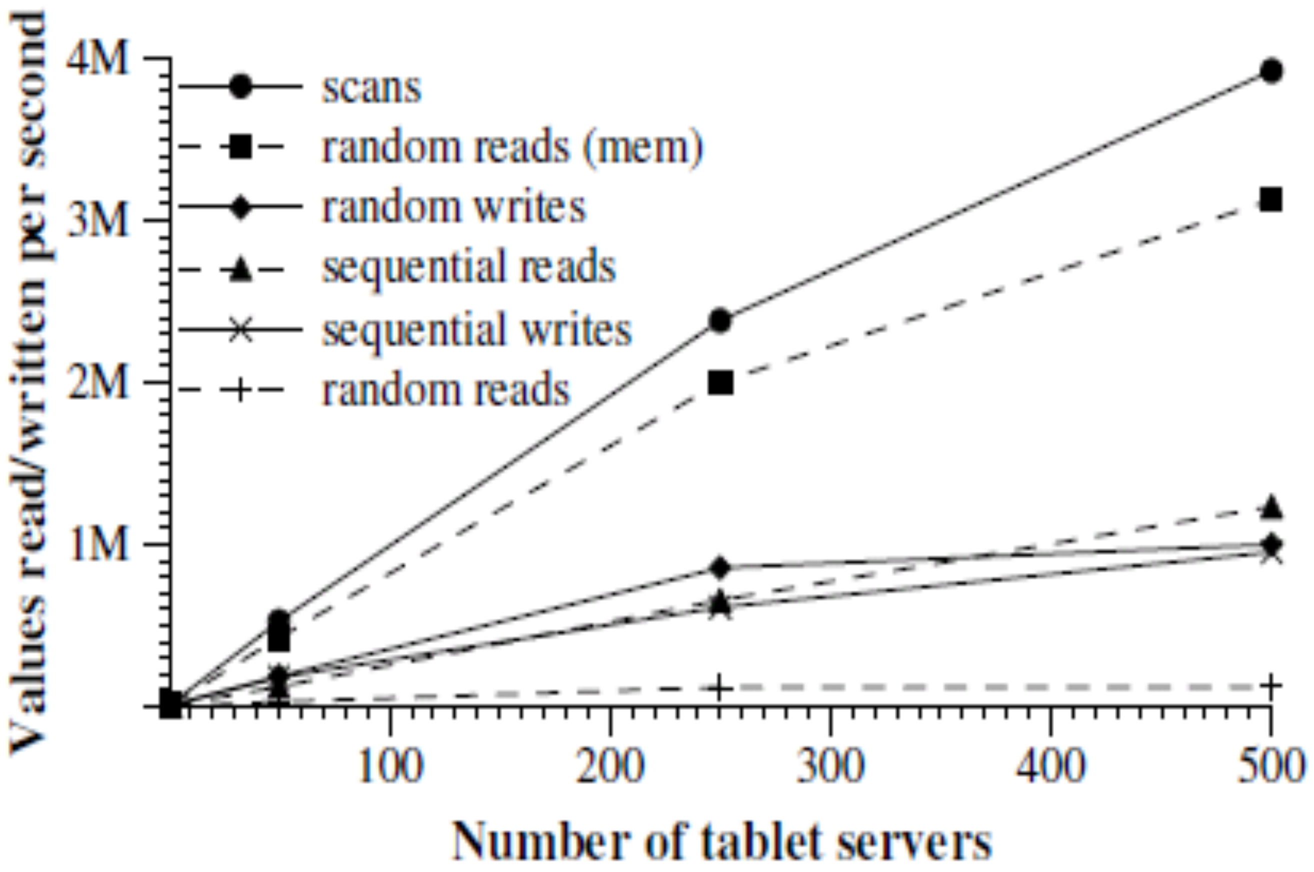
Performance Evaluation Cont..

- Used various benchmarks to evaluate the performance like *sequential write, random write, sequential read, random read, scan, and random reads (mem).*
- Two views on the performance of our benchmarks when reading and writing 1000-byte values to Bigtable. The table shows the number of operations per second per tablet server; the graph shows the aggregate number of operations per second.



	# of Tablet Servers			
Experiment	**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

Performance Evaluation Cont..



THANKS