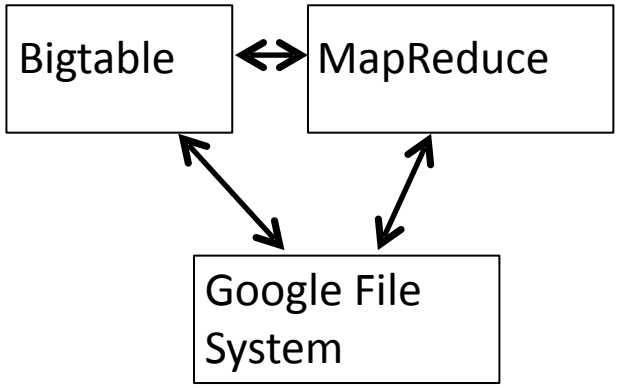
Cloud & Cluster Data Management

BIGTABLE

Bigtable – Motivation & Uses

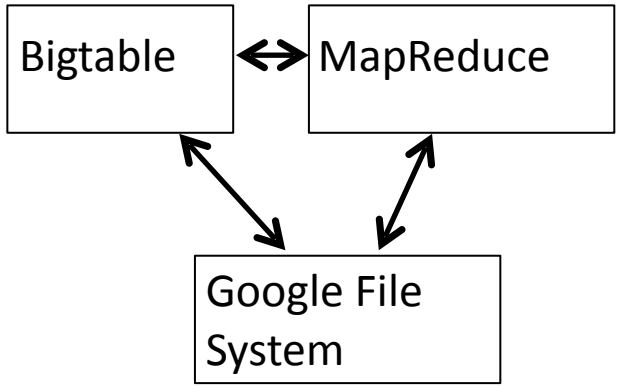
- Store structured data – scale to large size
  - Petabytes on commodity servers
- Goals:
  - Wide applicability
  - Scalablity
  - High performance
  - High availability



- Uses: Web indexing, Google Earth, Google Finance
  - Data type and size: URLs vs. Satellite imagery
  - **Throughput-oriented batch jobs vs. latency-sensitive jobs**
    - bulk processing, real-time data serving

Bigtable – Key Concepts

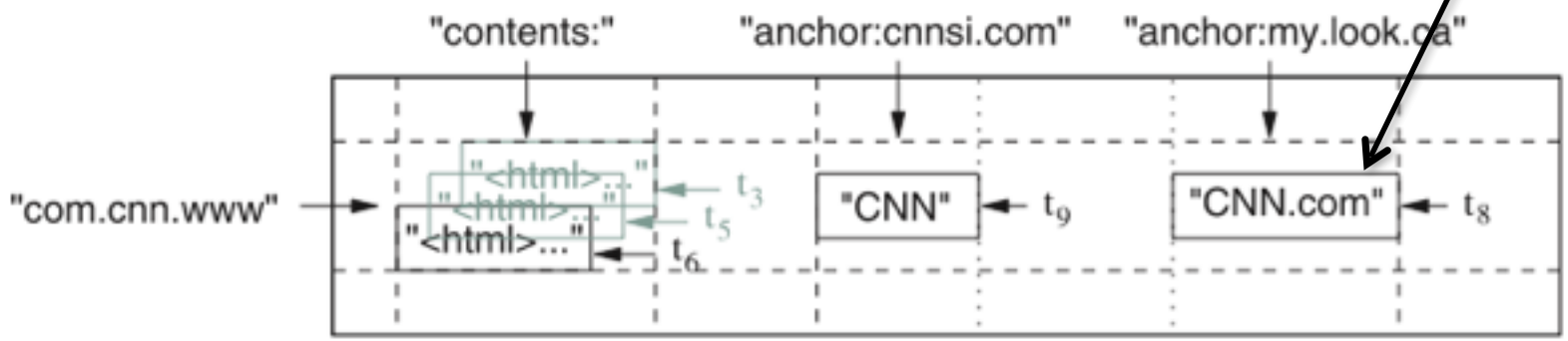
- Column family data model
- Dynamic control over data layout and format
- Control whether to serve data out of memory vs. disk
- Data is uninterpreted strings (no typing)
- Partitioning:
- Transactions: single-row only
  - But rows are heavyweight



Bigtable - Tables

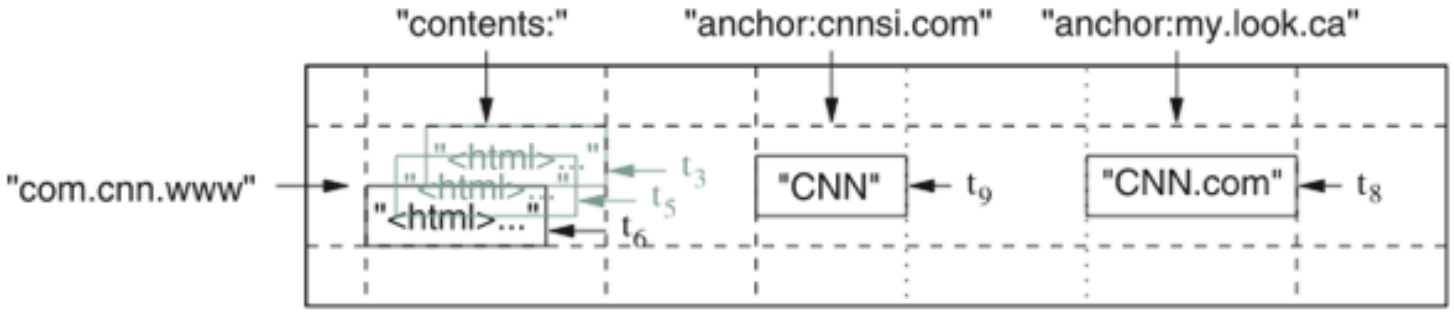
- Bigtable *cluster* – set of processes that runs Bigtable software
  - Each process serves a set of tables
- Tables:
  - **Sparse, Distributed, Persistent, Multi-dimensional Sorted map**
- Three Dimensions:
  - row: string
  - column: string
  - time: int64
- (row:string, column:string, time:int64) -> (uninterpreted) string

(row key, column key, timestamp) -> cell



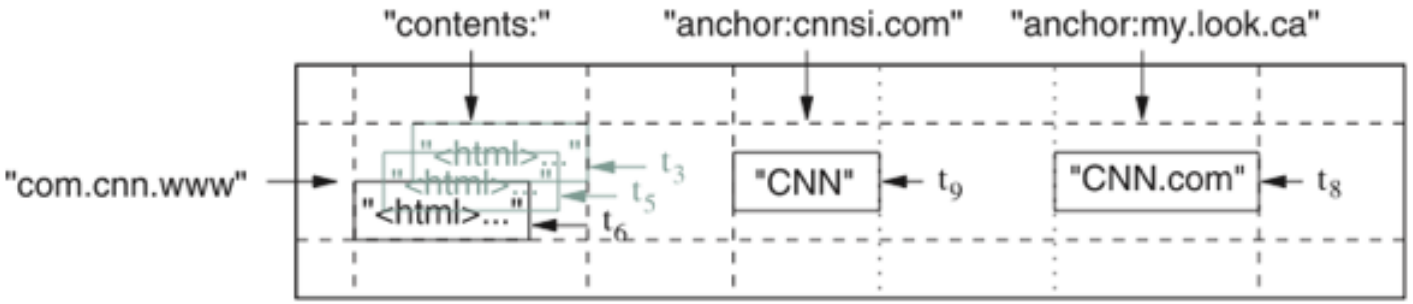
Bigtable - Rows

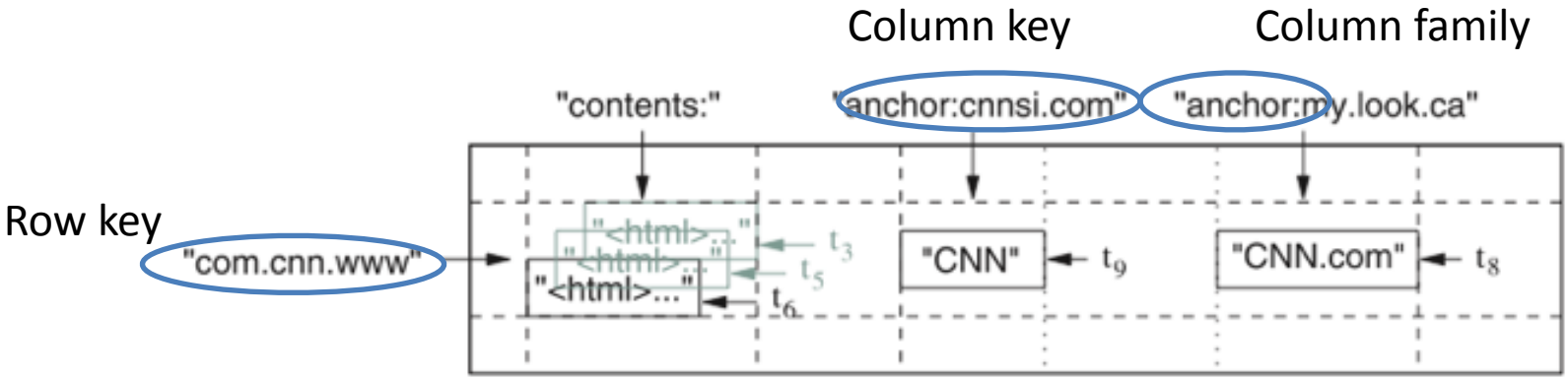
- Rows – kept ordered by row key
  - Choice of row key important
    - Select row key to get good locality of data access (i.e. reverse hostname)
    - Small row ranges => small # machines
  - Rows with consecutive keys grouped into tablets
  - Partitioning is fixed (in contrast with RDBMS)
  - # rows in a table is unbounded
- **Tablets are unit of distribution and load balancing**
- **Row is unit of transactional consistency**
  - Row read/writes are serializable
  - No transactions across rows



Bigtable - Columns

- Columns grouped into Column Families
  - Data stored in column families is usually of the same type (compressed together)
  - Number of **column families intended to be small (unlimited rows, cols)**
    - Keeps shared meta-data small
  - **Column families must be created explicitly**
  - Column key is family:qualifier
- Example column families:
  - language:_____ cell contains: language id
  - anchor: referring site cell contains: text associated with link
- **Column family is unit of access control**




Column key
Column family
"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
Row key
"com.cnn.www"
"<html>..."
"<html>..."
"<html>..."
t3
t5
t6
"CNN"
t9
"CNN.com"
t8

7

Bigtable - Timestamps

- Multiple version of the data in a cell - indexed by timestamp
- Timestamps assigned implicitly by Bigtable or explicitly by clients

Why might you chose implicit or explicit timestamps?

- Bigtable stores in decreasing timestamp order (most recent version read first)

What implicit assumption are they making?

- Garbage collection settings:
  - Keep last N versions
  - Keep last 7 days of data
- Webtable ex: timestamps are times pages were crawled

Bigtable API

- Create delete tables and column families
- Change cluster, table and column family meta-data
- Single-row transactions
  - Atomic read-modify-write sequences on a single row
- Does not support transactions across row keys

Bigtable API – Row Mutation Example

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

"Irrelevant details were elided to keep the example short."

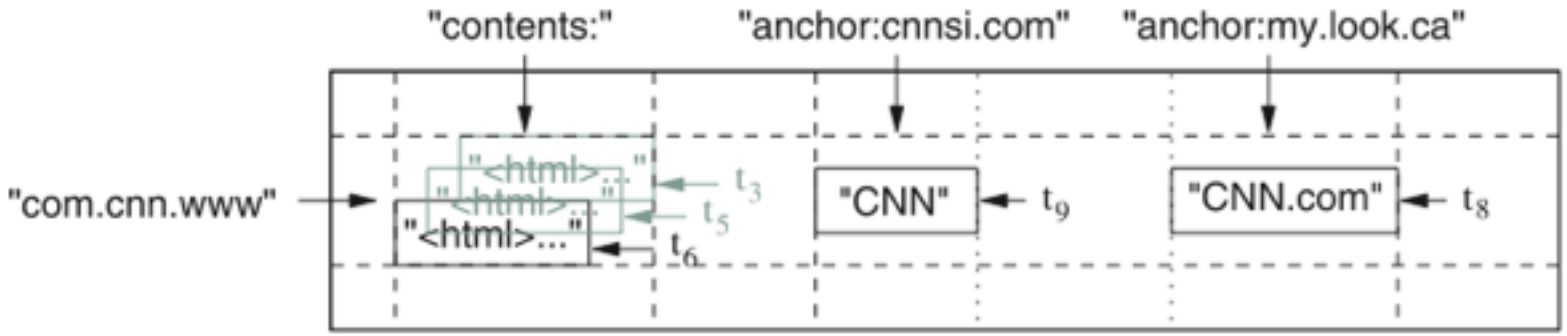
Bigtable API – Scanner Example

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Could restrict the scan

-produce only anchors whose columns match *anchor:*.cnn.com*,

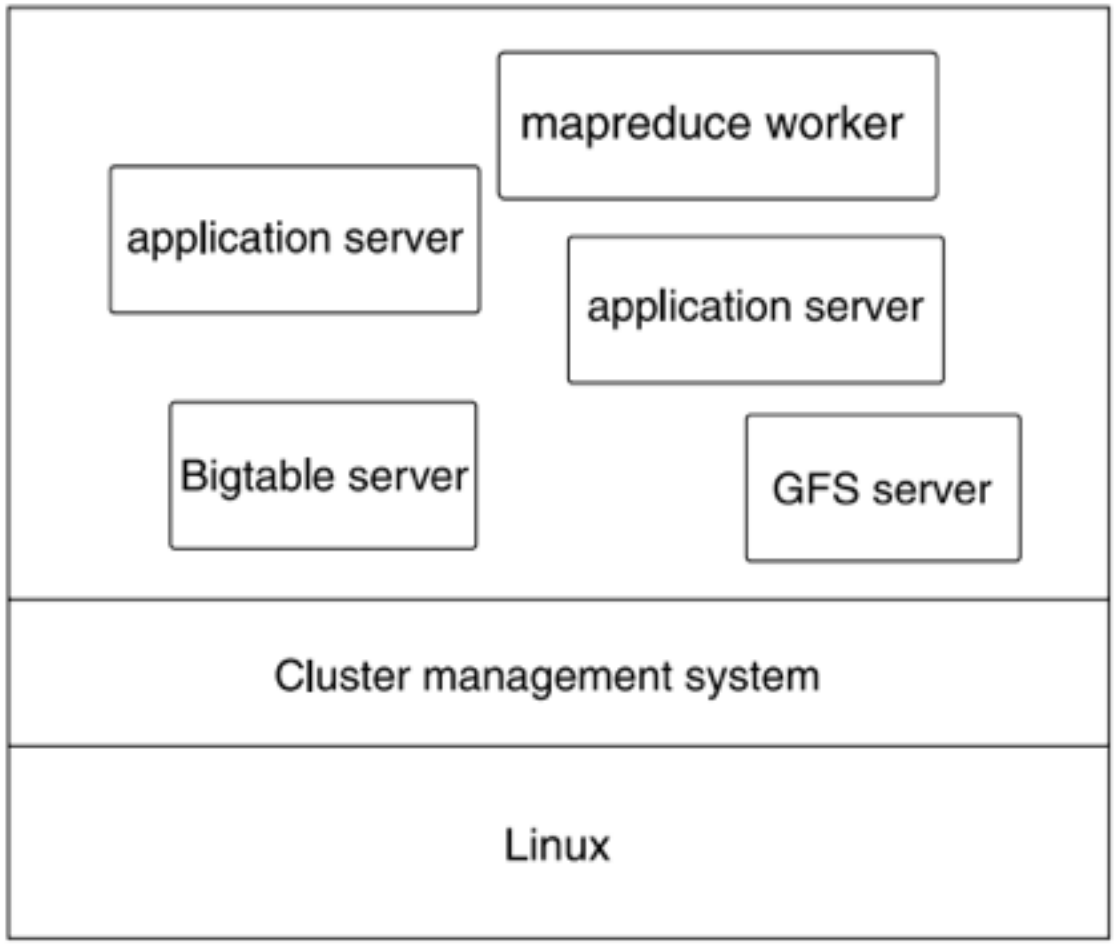
- produce only anchors whose timestamps fall within ten days of the current time



Bigtable API – More details

- Create delete tables and column families
- Change cluster, table and column family meta-data
- Single-row transactions
  - Atomic read-modify-write sequences on a single row
- Does not support transactions across row keys
- Interface for batching writes across row keys at the client
- Execution of client-supplied scripts in server address space (Sawzall)
  - Filtering, summarization, but no writes into Bigtable
- Wrappers written so Bigtable can be an input source and output source for MapReduce jobs

Bigtable – Building Blocks



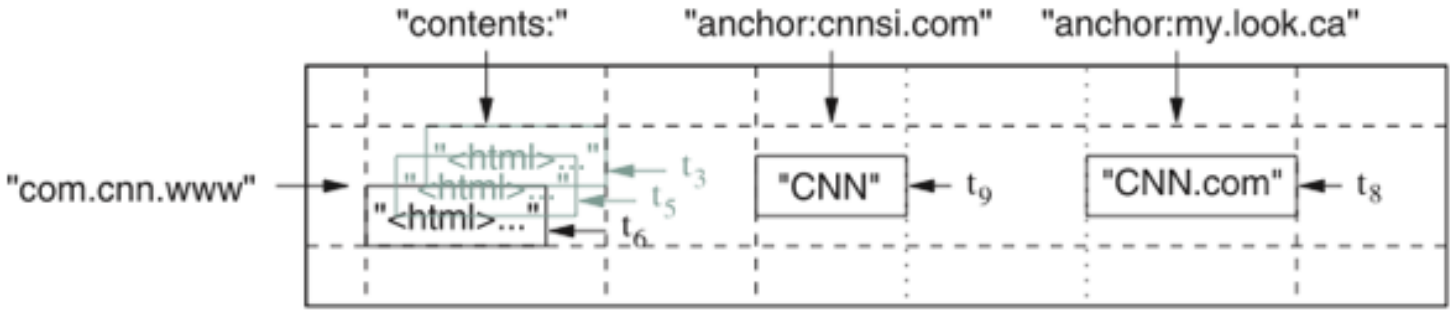
A typical set of processes that run on a Google machine. A machine typically runs many jobs from many different users.

Bigtable – Building Blocks

- Shared pool of machines – many applications
- Uses Google cluster management system for scheduling jobs, managing resources, monitoring machine status, dealing with machine failures
- GFS used to store log files and data files
  - Files are replicated with GFS
- SSTable immutable-file format
  - **Persistent, ordered immutable map** from keys->values
  - Keys and values are arbitrary byte strings
- Operations: lookup key and iterate over key/values in a range
- SSTable contains blocks (64KB in size), block index at end of file
  - Block index always stored in memory – lookup is one disk seek
- SSTable can be memory mapped

Recall: Bigtable - Rows

- Rows – kept ordered by row key
  - Choice of row key important
    - Select row key to get good locality of data access (i.e. reverse hostname)
    - Small row ranges => small # machines
  - Rows with consecutive keys grouped into tablets
  - Partitioning is fixed (in contrast with RDBMS)
  - # rows in a table is unbounded
- **Tablets are unit of distribution and load balancing**
- **Row is unit of transactional consistency**
  - Row read/writes are serializable
  - No transactions across rows



Discussion Question

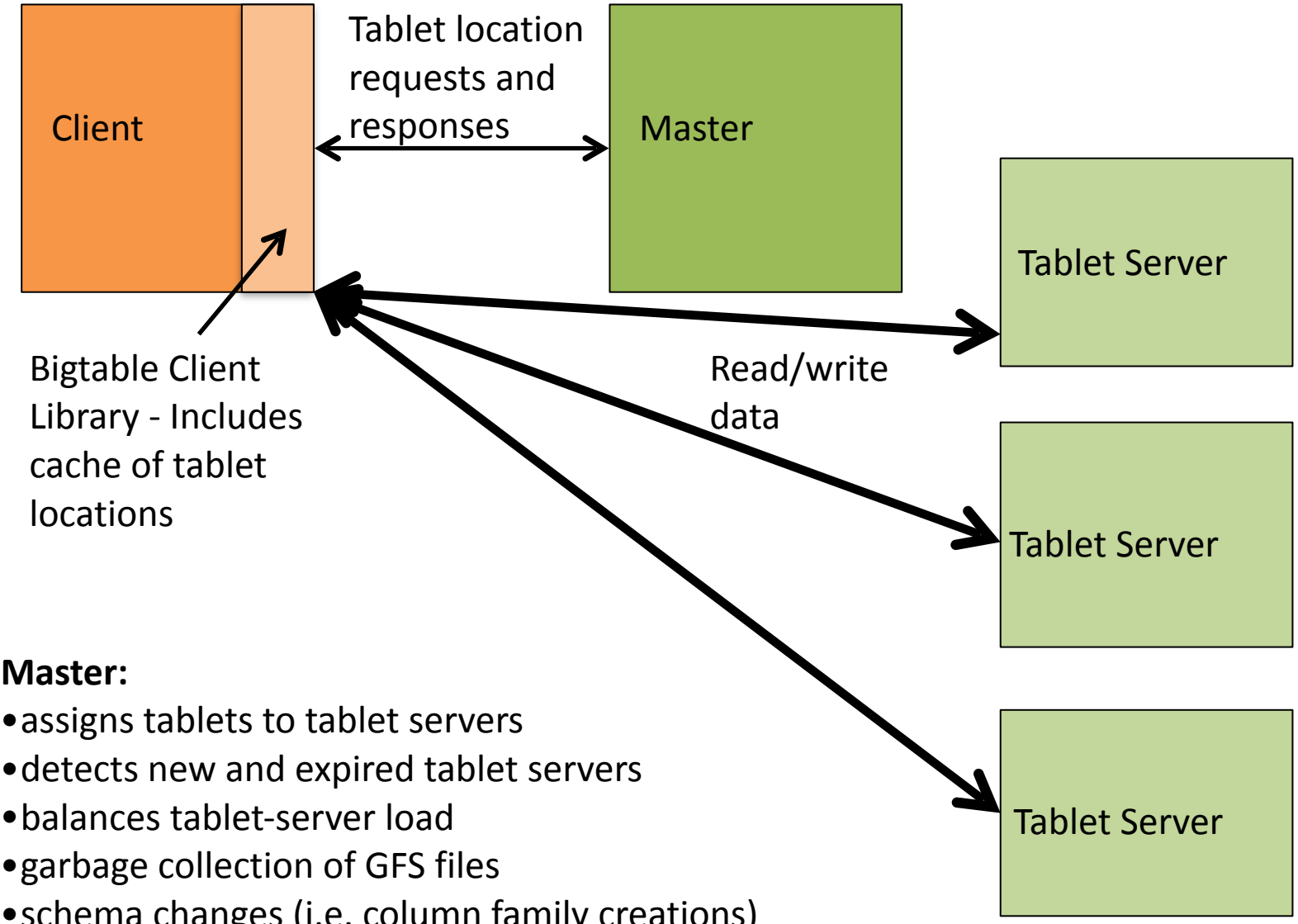
- Pick your favorite application. How does the single row transaction work for your application?
  - Areas in the application where this transaction model works well?
  - Areas in the application where this transaction model does not work well?

Bigtable - Implementation

- Three components
  - Library linked into clients
  - One master server
  - Tablet servers (dynamically removed or added)
- Master Duties:
  - Assigns tablets to Tablet Servers
  - Detects addition & expiration of tablet servers
  - Load Balancing
  - Garbage collection of GFS files
  - Schema changes (table & column family additions and deletions)
- Client
  - **Communicates with Tablet Servers for data**
  - Clients cache Tablet Server location information
  - **Most clients don't communicate with the master**

What was a tablet again?

Bigtable - Implementation

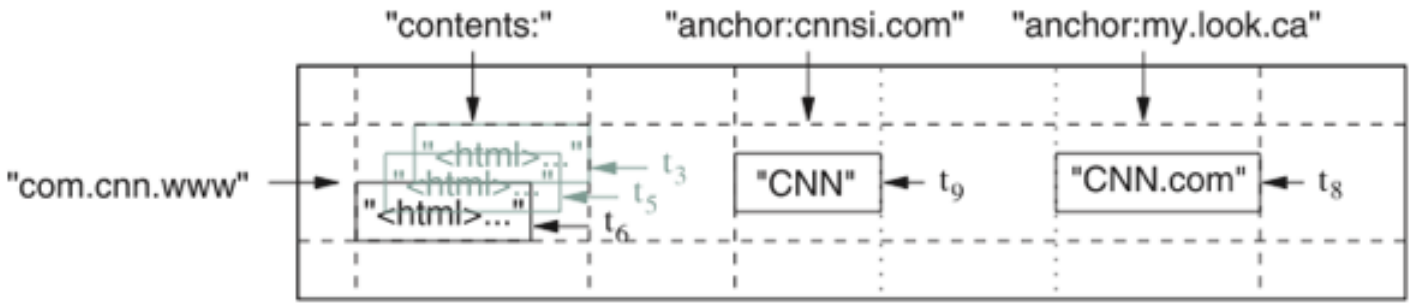


Master:

- assigns tablets to tablet servers
- detects new and expired tablet servers
- balances tablet-server load
- garbage collection of GFS files
- schema changes (i.e. column family creations)

Bigtable – Tablets

- Table consists of a set of tablets
- Each tablet contains all of the data associated with a row range (range partitioning on row key)
- Table automatically split into multiple tablets – each tablet about 1GB in size
- Tablet cannot be split in the middle of the row – row should be < few hundred GB



Tablet Location

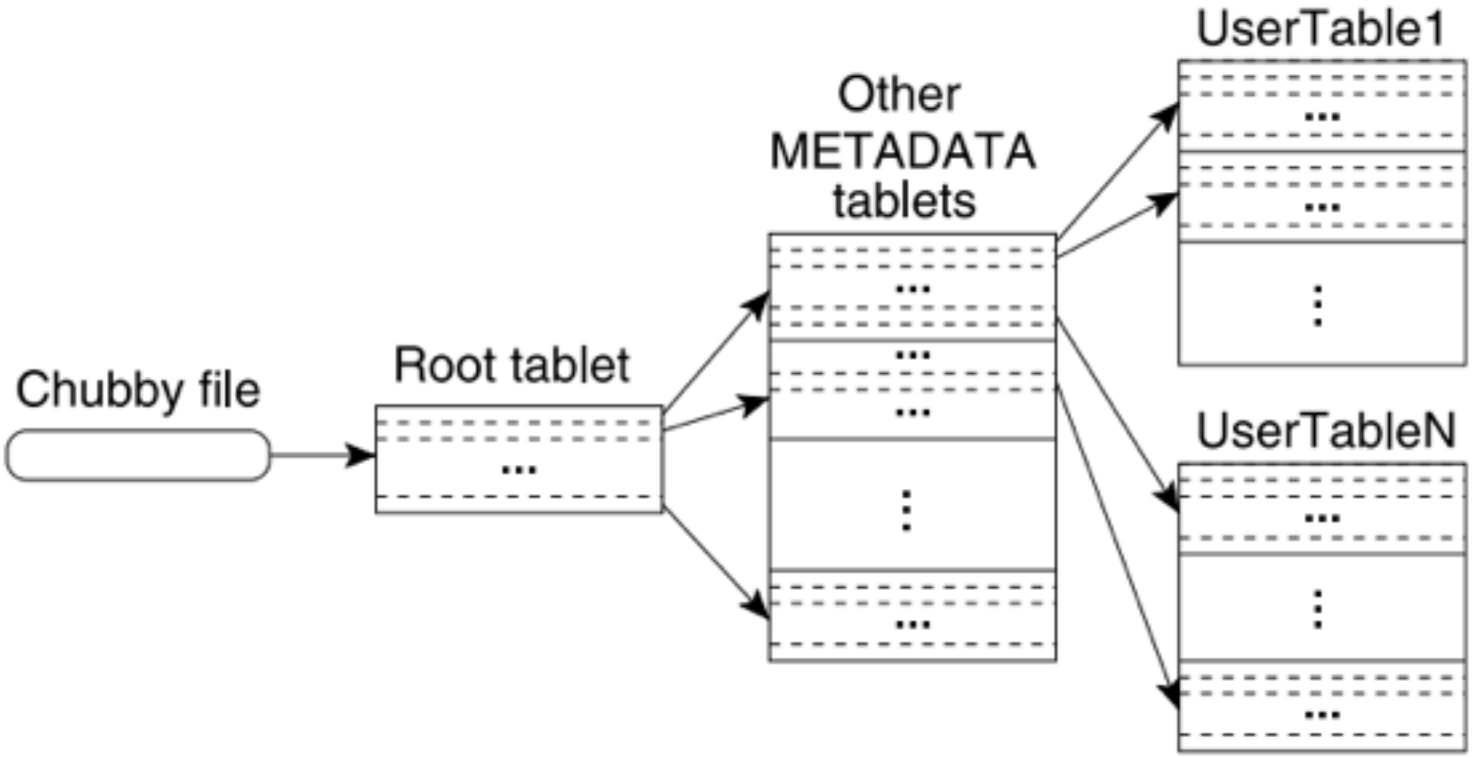


Fig. 5. Tablet location hierarchy.

What is Chubby?

- A highly-available and persistent distributed lock service
- Uses 5 active replicas – one is the master
- Replicas are consistent in the face of failure
- Provides a namespace that consists of directories and small files
- Each directory or file can be used as a lock
  - Reads and writes to a file are atomic
- Chubby client provides consistent caching of chubby files

Tablet Location

- Location tree is like B+ tree
- Chubby stores the location of the root tablet
  - Root tablet – stores locations of tablets of a special METADATA table
- METADATA tablets contains locations of user tablets
- Root is never split
  - Hierarchy is always 3 levels
  - Up to 2^{61} bytes with 128MB Metadata Tablets (2^{34} tablets)
- METADATA table/tablets store location and a rowkey – tableid and end row
- Each METADATA row ~1k

Tablet Location

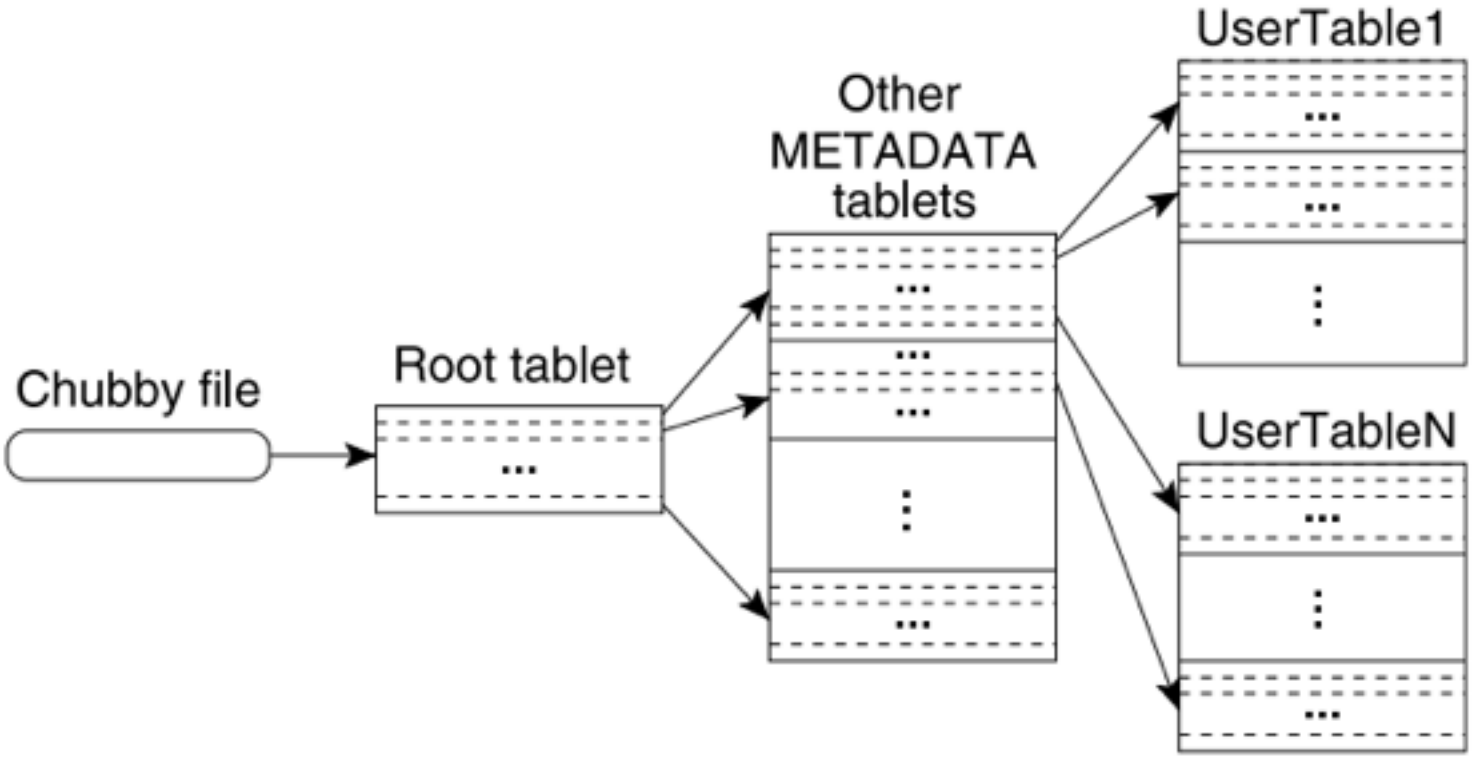


Fig. 5. Tablet location hierarchy.

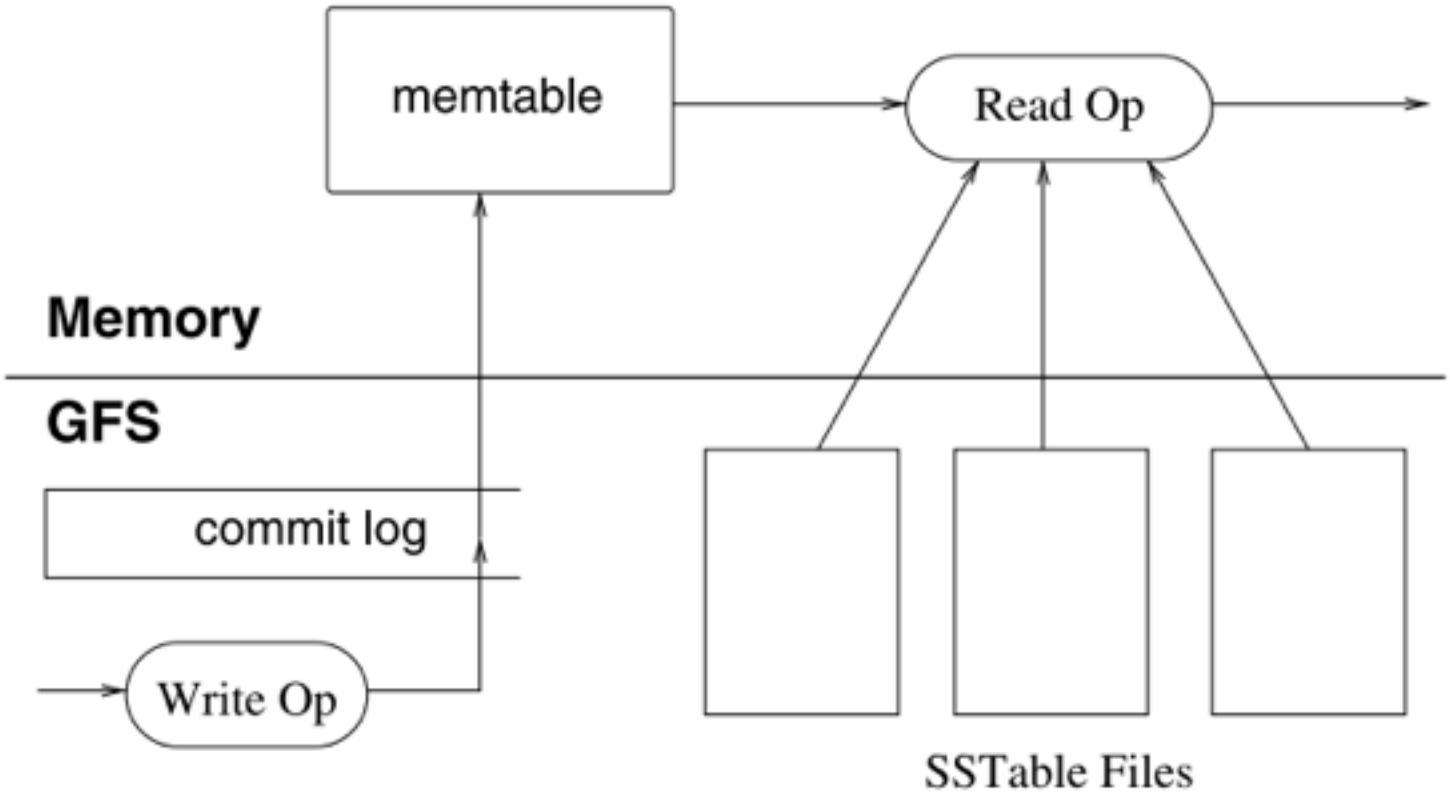
Client - Tablet Location

- Client traverses hierarchy to locate tablets, caches locations
- Empty / stale cache
  - Moves up in hierarchy if information is incorrect or not known (like B+ tree)
  - Empty client cache can cause 3 round trips to master + 1 read from Chubby
  - Stale client cache – 6 round trips + 1 read from Chubby
    - Stale cache entries discovered on misses
- Client also uses prefetching to limit round trips to master
- Metadata tables also store logging info (for debugging and performance analysis)

Tablet Assignment

- Tablet assigned to at most one tablet server
- Master keeps track of live tablet servers and assignment of tablets to tablet servers
- Chubby used to keep track of tablet servers
  - TS starts – creates and acquires exclusive lock on file in specific Chubby directory (servers directory)
  - Master monitors this directory
  - Tablet stops serving if it loses its exclusive lock (i.e. network partition causes loss of chubby session)
- Master periodically pings tablet servers to make sure they still have their locks

Tablet Representation



Tablet Serving – General Features

- Persistent state of tablet stored in GFS
- Updates written to a commit log that stores redo records
- Recently-committed updates stored in memtable
- Copy on write for updates
- Redo after crash:
  - Read metadata from METADATA table
    - SSTables & set of redo points
  - Read indices of SSTables into memory & reconstruct memtable by redoing updates since last redo point (checkpoint)
- Tablet servers handle tablet splits, other changes handled by master (table created, tablets merged)
  - TS commits split – recording new info in metadata table, notifies master
  - If notification fails, split discovered when a TS goes to load the tablet (file will contain only a portion of the tablet)

Tablet Serving – Writes & Reads

- Write operation
  - Check for well-formed
  - Check for authorization (chubby file – usually hit on chubby client cache) (remember perms are at column family level)
  - Valid mutation written to commit log (*group commit*)
  - After commit, contents inserted into memtable
- Reads
  - Check for well-formed
  - Check for authorization
  - Read operation executed on a merged view of SSTables and the memtable (both sorted)

Compactions

- Memtable increases in size with write operations
  - At threshold – minor compaction
    - memtable frozen
    - new memtable created
    - frozen memtable converted to an SSTable and written to GFS
  - Goals:
    - Shrinks memory usage of memtable
    - Reduces amount of data that has to be read from the commit log if server dies
- Merging compaction
  - Reads SSTables and memtable and outputs a SSTable
  - Run in background
  - Discard memtable and SSTable when done
- Major compaction – leaves only one SSTable
  - Non-major compactions can leave deletion entries and deleted data
  - Major compaction leaves no deleted data
  - Major compactions done periodically on all tables

Bigtable Schemas - Chubby

- Schemas stored in Chubby
- Recall: Chubby provides
  - Atomic whole-file writes
  - Consistent caching
- Chubby client sends update to Chubby master, ACL is checked
- Master installs new schema by (atomically) writing new schema file
- Tablet servers get schema by reading appropriate file from Chubby
  - Usually a hit on the Chubby cache
  - File is up-to-date due to consistent caching
- Comment: note impact of having only column families at schema level

Bigtable - Refinements

- Goal of refinements: performance, availability, reliability
- Locality groups
  - Column families assigned to client-defined locality group
  - SSTable generated for each locality group in each tablet (vertical partitioning)
  - Segregate column families that are not typically accessed together
- Ex: locality groups for Webtable
  - page meta-data (language, checksums)
  - page contents
- User wanting meta-data does not need to read page contents
- Locality groups can be declared to be in-memory
  - Good for small pieces of data that are accessed frequently
  - Note: SSTables immutable
- Clients control if SSTables for a locality group are compressed
  - Compress page contents in Webtable example

Bigtable-Caching / Commit Log

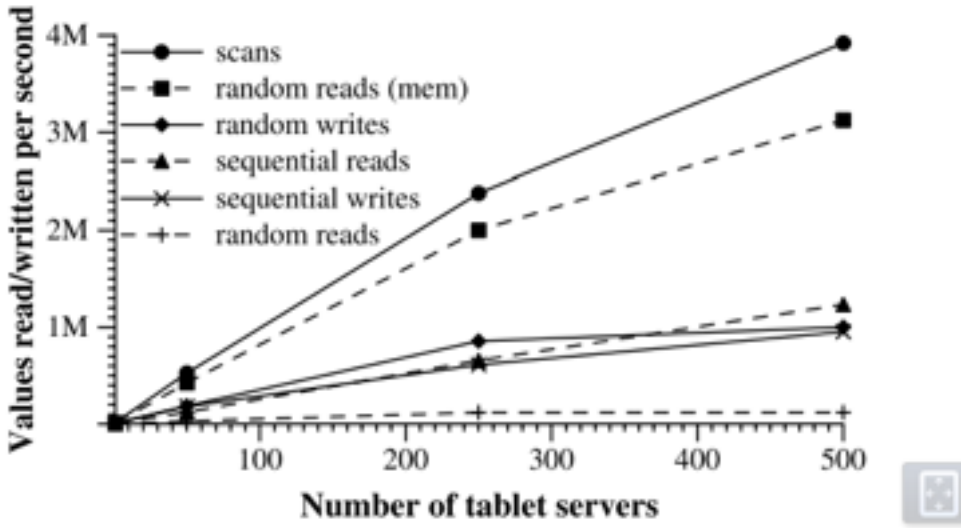
- Scan Cache
  - High-level cache that caches key-value pairs
  - Useful for applications that read the same data repeatedly
- Block Cache
  - Lower-level cache, caches SSTable blocks
  - Useful for applications that read data that is close to the data they recently read (sequential read, random reads)
- Commit Log
  - If commit logs were separate files, lots of disk seeks for writes
  - One commit log per tablet server – good performance during normal operation, but complicates recovery
    - When TS crashes, its tablets are split among many other tablet servers – all of which now need the commit log…

Performance

- N tablet servers – scale as N varies
- Same number of client servers as tablet servers
  - Clients are not a bottleneck
- R – rows in test – chosen to read/write approx 1GB of data per tablet server
- Sequential write
  - partitioned into 10N equal-sized ranges, assigned to N clients
  - Wrote a single string under each row key (uncompressible)
- Random write
  - Same as sequential except row key is hashed modulo R to spread load uniformly
- Sequential /random reads – similar to writes
- Scan – Bigtable API for scanning values in a row range – reduces RPCS executed

Table I. Number of 1000-byte values read/written per second. The values are the rate per tablet server.

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



Performance Observations

- Table shows number of ops/sec/tablet server
- Graph shows aggregate # ops/second
- Single Tablet Server performance
  - Random reads slower by order of magnitude
  - Transfer 64KB block, use one 1000-byte value
  - Could reduce block size to 8K for this use case
  - Random reads from memory faster - 1000-byte reads satisfied from Tablet Server's local memory
  - Scans faster – returns large number of values in response to a single RPC, amoritizes RPC overhead
  - Writes: TS appends writes to a single commit log – use group commit (Reads – one disk seek for each access)
  - Random & sequential writes have similar performance

Performance - Scaling

- Throughput increases as tablet servers increased
- Bottleneck on performance is the tablet server CPU
- Drop from 1-50 TSs, caused by an imbalance in load
  - Rebalancing throttled
  - Load shifted around during benchmark
- Random read – poor scaling
  - Transfer one 64KB block for each 1000-byte read, saturates network

Table II. Distribution of Number of Tablet Servers in Bigtable Clusters.

# of tablet servers			# of clusters
0	..	19	259
20	..	49	47
50	..	99	20
100	..	499	50
> 500			12

Table III. Characteristics of a Few Tables in Production Use.

Project name	Size (TB)	Comp. ratio	# (B) Cells	# Families	# Groups	% MMap	Frontend
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Analytics	20	29%	10	1	1	0%	Yes
Analytics	200	14%	80	1	1	0%	Yes
Base	2	31%	10	29	3	15%	Yes
Earth	0.5	64%	8	7	2	33%	Yes
Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Pers. Search	4	47%	6	93	11	5%	Yes

Size (measured before compression) and # *Cells* indicate approximate sizes. *Comp. ratio* (compression ratio) is not given for tables that have compression disabled. *Frontend* indicates that the application's performance is latency-sensitive.

Real Applications

- Google Analytics
  - Raw click table (200TB) – row for each end-user session – name is web site name and session creation time
    - Sessions for the same web site are contiguous and sorted chronologically
  - Summary table (20TB) – predefined summaries for each web site
    - Generated from Raw Click table by Map Reduce jobs
- Google Earth
  - Preprocessing pipeline uses table to store raw imagery (70TB) – served from disk
  - Preprocessing – Map Reduce over Bigtable to transform data
  - Serving system – one table to index data stored in GFS (500GB) – in-memory column families used