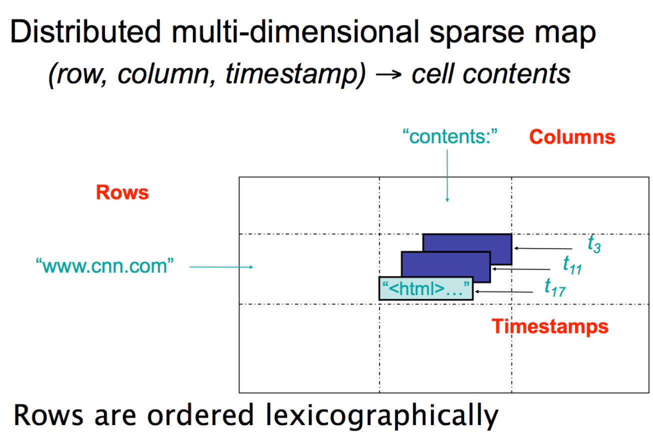
Bigtable

- Google's first answer to the question: "how do you store semi-structured data at scale?"
  - scale in capacity
    - e.g., webtable
      - 100,000,000,000 pages * 10 versions per page * 20 KB / version
      - 20 PB of data (200 million gigabytes)
    - e.g., google maps
      - 100TB of satellite image data
  - scale in throughput
    - hundreds of millions of users
    - tens of thousands to millions of queries per second
  - low latency
    - a few dozen milliseconds of total budget inside Google
    - probably have to involve several dozen internal services per request
    - can afford only a few milliseconds / lookup on average

Data model

- Data model is richer than a filesystem but poorer than full-fledged database
- Table indexed by
  - row key . column key . timestamp
  - lets you do fast lookup on a key
  - lets you do multiversion storage of items
- Rows ordered lexicographically, so scans are in order
  - lets you use a bigtable to do a sort



- Simple access model: column family is unit of access control

Programming interface

Imperative, language-specific APIs vs. declarative SQL-like language

Bottom-up description of Bigtable's design

Tablet

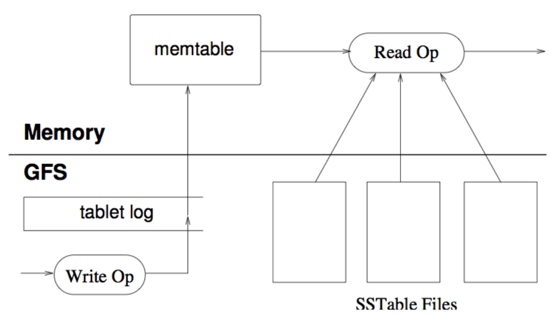
- a row range
- is the unit of distribution and load balancing
- reads of short row ranges are efficient, as stay within a single tablet usually

SSTable

- a file format used to store bigtable data durably
- persistent, ordered, immutable key:value map
  - lookup, iterate operations
- stored as a series of 64KB blocks, with a block index at the end
  - index is loaded into memory when SSTable is opened
  - lookup in a single seek; find block in memory index, seek to block on disk

Tablet server

- manages 10-1000 tablets
  - handles read/write requests to tablets that it is assigned
  - splits tablets when they get too big
- durable state is stored in GFS (a scalable file system)
  - GFS gives you atomic append and fast sequential reads/writes
- writes:
  - updates committed to a commit log that stores REDO records (i.e., WAL)
  - recently committed writes are cached in memory in a memtable
  - older writes are stored in a series of SStables
- reads:
  - executed on a merged view of the SSTables and the memtable
    - both are lexicographically stored, so merge is efficient

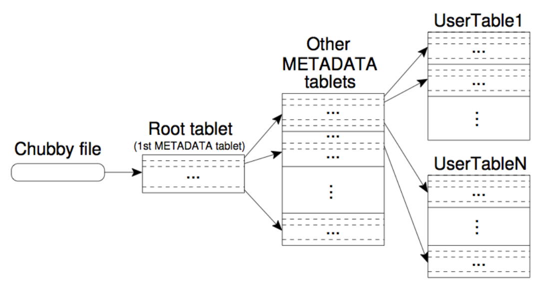


- Compactions
  - When memtable reaches a threshold size, a minor compaction happens
    - memtable is frozen and written to GFS as an SStable
    - new Memtable is created, and tablet log "redo point" is updated – i.e., tablet log is pruned
    - goals: reduce memory footprint, reduce amount of tablet log read during recovery
  - Periodically, do a merging compaction
    - read multiple SStables and memtable, write out a single new SStable
  - Once in a while, do a major compaction
    - merging compaction that produces a single SStable
    - lets you suppress deleted data, that previously lived in old SStables (tombstone is needed)

High-level structure

- three major components to bigtable
    - a "client library" that is linked into each client
        - soft-state: caches (key range) -> (table server location) mappings
    - a single "master" server
        - assigns tablets to tablet servers
        - detects addition/deletion of tablet servers
        - balances load across tablet servers
        - garbage collects GFS files
        - handles schema changes (e.g., addition of column families, tables)
    - many tablet servers
        - handles read/write requests to its tablets
        - splits tablets when they get too big (> ~200MB)
    - a chubby cell
        - ensures there is a single master
        - stores bootstrap location of bigtable data
        - discovery and death finalization of tablet servers
        - store bigtable schema and access control information

- (key) to (tablet) to (tablet server) mapping
    - chubby file stores the location of the root tablet
    - root tablet stores the location of all METADATA tablets
    - METADATA tablet stores the location of a set of user tablets
    - tablet locations served out of memory



    - client library caches tablet locations
        - moves up the hierarchy if it misses in cache or cache entry is stale
        - empty cache: three round-trips
            - one to chubby, one to root tablet, one to other METADATA table
        - prefetching done here – why?

Tablet assignment

- Chubby is used to track tablet servers
    - when a tablet server is started, it creates and acquires a lock on a uniquely named file in Chubby's "servers" directory (membership management!)
        - master monitors this directory to discover new tablet servers
        - if a tablet server loses its exclusive lock, tablet server stops serving, and attempts to regain the lock
        - master grabs lock and removes file to cause tablet server to kill itself permanently; master reassigns tablets in this case
- Chubby is used to track the master
    - when a new master is started, it:

        - grabs the unique master lock in chubby (leader election!)
        - scans the servers directory to find live servers
        - communicates with the live servers to learn what tablets they are serving
        - scans METADATA table to learn set of tablets that exist
        - assigns unassigned tablets to tablet servers
    - Q: why not store tablet → tablet server assignments durably in chubby?
        - no need; tablet servers already store the truth of what tablets they serve, and small enough number of them that a full scan is cheap, given how infrequently the master restarts

Optimizations

- tablet-server-side write-through cache
    - scan cache: high-level key/value cache   blockcache: GFS block cache
    - why no data cache in client library?

- SSTable per "locality group" – a set of column families
    - excludes from reads unrelated columns

- SSTables can be compressed
    - 10-1 reduction in space
        - and thus improvement in logical disk bandwidth
    - decompression presumably done on server-side? no network bandwidth benefits!

- bloom filter
    - explain what a bloom filter is
    - in-memory bloom filter filters out most lookups for non-existent rows/columns

Performance – 1000 byte read/write benchmark

Go over ordering of lines

Scans: batch multiple reads into a single RPC (read-sequential reads one-per-RPC)

Random writes and sequential writes are roughly the same, since both result in appends to a log

Random reads is the worst, since each request involves a 64KB SStable block read from GFS to a tablet server