CS262 Lecture Notes
Eric Brewer

BigTable

Goal: a general-purpose data-center storage system

- big or little objects
- ordered keys with scans
- notion of locality
- very large scale
- durable and highly available
- hugely successful within Google -- very broadly used

Data model: a big sparse table

- rows are sort order
    - atomic operations on single rows
    - scan rows in order
    - locality by rows first
- columns: properties of the row
    - variable schema: easily create new columns
    - column families: groups of columns
        - for access control (e.g. private data)
        - for locality (read these columns together, with nothing else)
        - harder to create new families
- multiple entries per cell using timestamps
    - enables multi-version concurrency control across rows

Basic implementation:

- writes go to log then to in-memory table “memtable” (key, value)
- periodically: move in memory table to disk => SSTable(s)
    - “minor compaction”
        - frees up memory
        - reduces recovery time (less log to scan)
    - SSTable = immutable ordered subset of table: range of keys and subset of their columns
        - one locality group per SSTable (for columns)
    - tablet = all of the SSTables for one key range + the memtable
        - tablets get split when they get too big
        - SSTables can be shared after the split (immutable)
    - some values may be stale (due to new writes to those keys)
- reads: maintain in-memory map of keys to {SSTables, memtable}
    - current version is in exactly one SSTable or memtable
    - reading based on timestamp requires multiple reads
    - may also have to read many SSTables to get all of the columns
- scan = merge-sort like merge of SSTables in order

        - easy since they are in sorted order
    - Compaction
        - SSTables similar to segments in LFS
        - need to “clean” old SSTables to reclaim space
            - also to *actually* delete private data
        - Clean by merging multiple SSTables into one new one
            - “major compaction” => merge all tables

Bloom Filters
- goal: efficient test for set membership: member(key) -> true/false
- false => definitely not in the set, no need for lookup
- true => probably is in the set
    - so do lookup to make sure and get the value
- generally supports adding elements, but not removing them
    - but some tricks to fix this (counting)
    - or just create a new set once in a while
- basic version:
    - m bit positions
    - k hash functions
    - for insert: compute k bit locations, set them to 1
    - for lookup: compute k bit locations
        - all = 1 => return true (may be wrong)
        - any = 0 => return false

    - 1% error rate ~ 10 bits/element
        - good to have some a priori idea of the target set size
- use in BigTable
    - avoid reading all SSTables for elements that are not present (at least mostly avoid it)
    - saves many seeks

Three pieces to the implementation:
- client library with the API (like DDS)
- tablet servers that serve parts of several tables
- master that tracks tables and tablet servers
    - assigns tablets to tablet servers
    - merges tablets
    - tracks active servers and learns about splits
    - clients only deal with master to create/delete tables and column family changes
    - clients get data directly from servers

All tables part of one big system
- root table points to metadata tables
    - never splits => always three levels of tablets
- these point to user tables

Tricky bits:

- SSTables work in 64k blocks
  - pro: caching a block avoid seeks for reads with locality
  - con: small random reads have high overhead and waste memory
    - solutions?
- Compression: compress 64k blocks
  - big enough for some gain
  - encoding based on many blocks => better than gzip
  - second compression within a block
- Each server handles many tablets
  - merges logs into one giant log
    - pro: fast and sequential
    - con: complex recovery
      - recover tablets independently, but their logs are mixed...
        - solution in paper: sort the log first, then recover...
      - long time source of bugs
  - Could we keep the logs separate?
- Strong need for monitoring tools
  - detailed RPC trace of specific requests
  - active monitoring of all servers