Cassandra: Structured Storage over P2P network (CS349C)

Presented by:
Debangsu Sengupta
Michael Chan
2010/01/12

Motivation

- Inbox Search problem at Facebook
  - Term search
  - Firstname, Lastname of contacts

Overview

- Structured storage system
- Combines ideas from:
  - Dynamo:
    - Low-level messaging/routing
    - Consistent hashing, one-hop DHT, vector clocks, hinted handoff.
  - BigTable:
    - Storage layer/data model
    - Column, ColumnFamily etc.
    - Not SQL

Design goals

- Cost-effective
  - Commodity hardware
- High availability
  - “always-up” service in the middle of failures
- Incremental scaling, Minimal administration
  - Drop in nodes. System manages data around
- Efficient data layout
  - No random reads/writes.

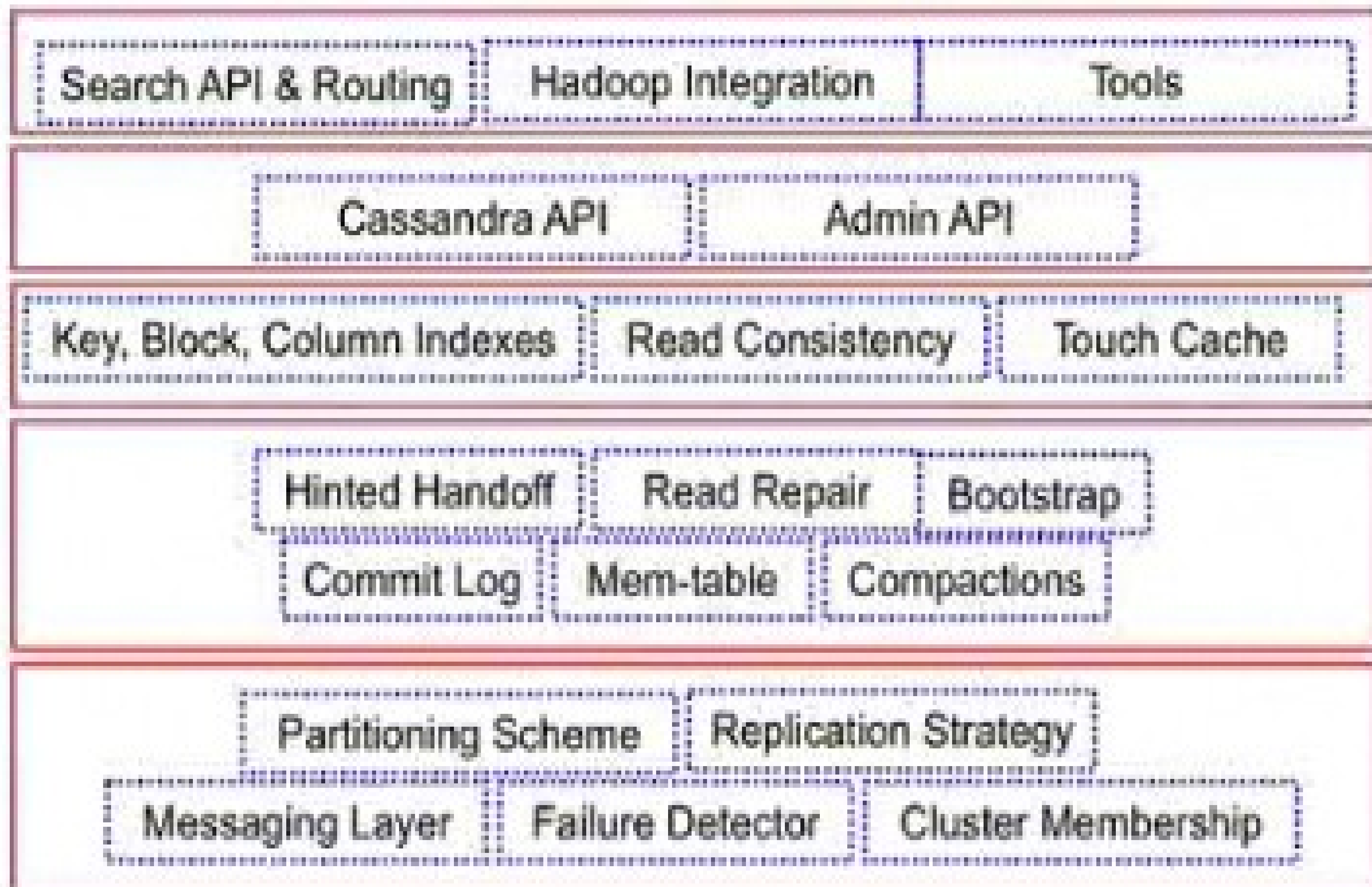
Existing solutions comparison

- MySQL, File-based solutions
  - Random reads and writes: Slows throughput
  - Read-modify-write semantics.
  - Locking on files/tables etc.

Solution: Cassandra

- No single point of failure: P2P storage
- Storage layer:
  - Datamodel is Bigtable like. Structured – column/column family. Dynamo is key-value only.
- Replication
  - Prevent data loss
- Optimized writes:
  - Sequential writes. Disk content is immutable.
- Caching
  - Improve read performance.

Architecture



Architecture: Top Layer

- Efficient reads/write: Key, Block, Column indexes
- Efficient reads
  - Read consistency, touch cache (load into memory as person types)
- App API's
  - Cassandra API, Admin API
- Tools API:
  - Search API, Hadoop integration, Tools API (system loading)

Cassandra API

Get API

- ColumnFamily get(key, columnFamily)
- ColumnFamily get(key, columnFamily, column)
- ColumnFamily get(key, columnFamily, superColumn)
- ColumnFamily get(key, columnFamily, superColumn, column)

Put API

- put(Key, columnFamily)
- put(Key, columnFamily, column)
- put(Key, columnFamily, superColumn)
- put(Key, columnFamily, superColumn, column)

Middle Layer

- Commit Log:
  - Journaling / WAL. Authoritative copy of data.
- Memtable:
  - In-memory replica of data. Cache
- Compaction:
  - Compact files when data structures are full

Middle Layer (2)

- Application concepts
- Hinted handoff:
  - For high availability. Node down => successor becomes temporary coordinator with a hint about downed node. Resolved when node re-joins.
  - Q: How does quorum write work in this case?
    - Say n=3, w=2. 1 node up. Pick a random node to make w=2 and write succeeds.
- Bootstrap:
  - Drop in node. Take position. Info about some nodes. Gossip to disseminate state. Get data for which it is coordinator.

Data Model: ColumnFamily

- CF: ColumnFamily
  - Defined upfront in config file, can only be changed by admin.
  - Name: Arbitrary String
  - Type: Simple/Super
    - SuperCF contains ColumnFamilies (Multi-Column Index)
    - Simple CF contains Columns
  - Sort: Name/Time
  - Analogous to BigTable's Locality Group

Data Model: Column

- Column
  - Name: arbitrary string
  - Value: Binary blob. Non-indexed. Client interprets.
  - Timestamp: client provided.

Data Model: Keys

- Keys:
  - Arbitrary string.
  - Associated with # of columns. 1 or many.
  - e.g. key: “dude”. 2 columns.

Data Model: Example

- Inbox Search Schema
- MaiList: personId → mailID mapping.
  - Sort: Name
- ThreadList: word → threadID mapping.
  - Sort: Time
- UserList → personId1 ↔ personId2 interaction mapping.
  - Sort: Time
- User1 sends mail to User2 with word “dude”:
  - Columns added in UserList.
  - Column added for “dude” in ThreadList
  - Column added in MailList for User1.

Inbox Search Example

Writes

- 2 types
  - Asynchronous
  - Quorum writes
- Inbox uses async. for performance
  - Client sends request to any node.
  - Based on replica state, pushes write to a few nodes.
  - Returns immediately.
  - Perf over guarantee.
  - Reduce loss probability with replication.
  - Order by timestamp

Writes ctd

- Sequential disk writes:
  - Write to commit log before updating memtables (hash maps).
  - Dedicates log disks => sequential writes using offsets. Immutable writes.
  - Commit log rollover every 127 MB.
- Writes 2 ops:
  - Sequential disk write to log.
  - In mem op.

Memtable issues

- Memtable crosses thresholds:
  - Data size, # of keys (128 default), lifetime expiry (client provided).
  - => Flush operation.
    - Memtable put in a queue. Flushed to disk.
    - Update headers.
      - Checkpoint: Commit Log checks whether all its column families in log have been flushed. If so, deletes old records.
- On flush:
  - For every CF, two files: Data File, Index File.
  - Index file: Key, offset pair in data file
    - bloom filter (all keys in data file)
  - Data file: data, block index (key → block + offset).

Compaction

- Minor and major compactions. Similar to BigTable.
- Run merge sort on data files. Update indexes.
- Update bloom filter.

Write properties

- Lockless.
- Not doing read before write in common case.
- Like write-through cache:
  - Read from same node: see own writes.
  - Latest change in failures using version clock/timestamp?
- Sequential writes
- Atomicity for single key:
  - All or none CF's will be written.
- Always writeable: in the event of failures
- Failure recovery from commit log.

Read

- Read request comes to random node
- Each node knows about all other nodes. Routes to closest node.
- Node services with diff levels of consistency
  - Inbox: Low. Financial data: High
  - Consistency mechanisms: quorum, version lock and read repair on digest mismatch.
  - Send digest req to all replicas. Get digest resp w/ delta. Share responses with outdated replicas.
- Failure modes:
  - Nodes down, corrupt commit log.
  - Tombstones: TBD.

Read on a node

- Memory
  - Look in memtable hashmap. Return immediately.
  - Look in buffer cache for disk.
  - Disk.
- Touch cache:
  - Have a hint about key.
  - Pre-load data for upcoming search to buffer cache.
  - Service request from memory.

Read from list of files

- Check bloom filter for key
- If key exists, look at file.
  - 99% time the filter is correct.
  - Go to index file. Jump to block index, column offset.
  - Idea: Minimize amount of disk read. Exact file, ad block.

Bootstrap

- Load new node in cluster. Assign random value in circle.
- Admin command – loadme command.
- Gossip propagation.
- Receives data from existing nodes.
- Existing nodes: Anti-compaction:
  - Split ranges into:
    - FiletoSend: Send the file to new nodes
    - Remaining

Architecture: Core Services

- Lowest layer
- Modules:
  - Messaging layer: Async msg based communication
- User modules:
  - Failure detector: Estimates probability of node down.
  - Cluster membership: Gossip.
- Partitioning Scheme: Consistent hashing (DHT)
- Replication strategy: Which nodes have copies of data.

Cassandra Core

- Partition using consistent hashing.
  - Servers and keys hashed over ring.
- Nodes take positions on the circle.
- A, B, D. exists. B responsible for AB range. D resp for BD range. A resp for DA range.
- C joins. B, D split ranges. C gets BC from D.

Messaging Service Features

- API for intra cluster control/data communication
  - send/recvOneWay / UDPOneWay
- Control Msgs: UDP
- Data / Replication: TCP
- Non-blocking I/O mostly.
- SEDA pipeline stages for processing msgs.
- TCP connection pooling and established.
- Multicast possible but disabled.

Replication Strategy

- Replicate node data in up to n-1 neighbors over ring.
- How to pick:
  - Rack Aware
  - Rack Unaware
  - Datacenter Aware
  - Example N = 3
    - 2 copies on different racks
    - 3rd copy on different data center.

Cluster membership

- Dissemination using gossip.
- N nodes. Takes O (log N) rounds to disseminate states.

Accrual Failure Detector

- Probabilistic estimation of a node's uptime based on inter-arrival time of past msgs.
- Past predicts future.

References

- Cassandra - A Decentralized Structured Storage System, Lakshman et al. (Link)
- Cassandra presentation, NOSQL. (Link)