Design Patterns for Distributed Non-Relational Databases

aka
Just Enough Distributed Systems To Be Dangerous
(in 40 minutes)

Todd Lipcon
(@tlipcon)

Cloudera

June 11, 2009

Introduction

Common Underlying Assumptions

Design Patterns
- Consistent Hashing
- Consistency Models
- Data Models
- Storage Layouts
- Log-Structured Merge Trees

Cluster Management
- Omniscient Master
- Gossip

Questions to Ask Presenters

Why We're All Here

- Scaling up doesn't work
- Scaling out with traditional RDBMSs isn't so hot either
  - Sharding scales, but you lose all the features that make RDBMSs useful!
  - Sharding is operationally obnoxious.
- If we don't need relational features, we want a distributed NRDBMS.

Closed-source NRDBMSs

"The Inspiration"

- Google BigTable
  - Applications: webtable, Reader, Maps, Blogger, etc.
- Amazon Dynamo
  - Shopping Cart, ?
- Yahoo! PNUTS
  - Applications: ?

Data Interfaces

"This is the NOSQL meetup, right?"

- Every row has a key (PK)
- Key/value get/put
- multiget/multiput
- Range scan? With predicate pushdown?
- MapReduce?
- SQL?

Underlying Assumptions

Assumptions - Data Size

- The data does not fit on one node.
- The data may not fit on one rack.
- SANs are too expensive.

Conclusion:
The system must partition its data across many nodes.

Assumptions - Reliability

- The system must be highly available to serve web (and other) applications.
- Since the system runs on many nodes, nodes *will* crash during normal operation.
- Data must be safe even though disks and nodes *will* fail.

Conclusion:
The system must replicate each row to multiple nodes and remain available despite certain node and disk failure.

Assumptions - Performance

...and price thereof

- All systems we're talking about today are meant for real-time use.
- 95th or 99th percentile is more important than average latency
- Commodity hardware and slow disks.

Conclusion:
The system needs to perform well on commodity hardware, and maintain low latency even during recovery operations.

Design Patterns

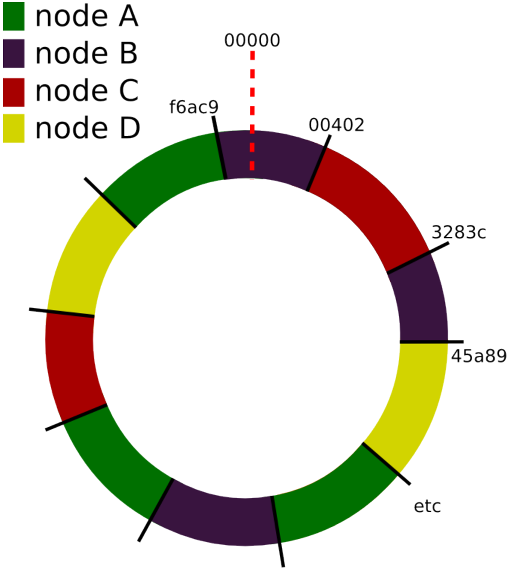
Partitioning Schemes

"Where does a key live?"

- Given a key, we need to determine which node(s) it belongs on.
- If that node is down, we need to find another copy elsewhere.
- Difficulties:
  - Unbounded number of keys.
  - Dynamic cluster membership.
  - Node failures.

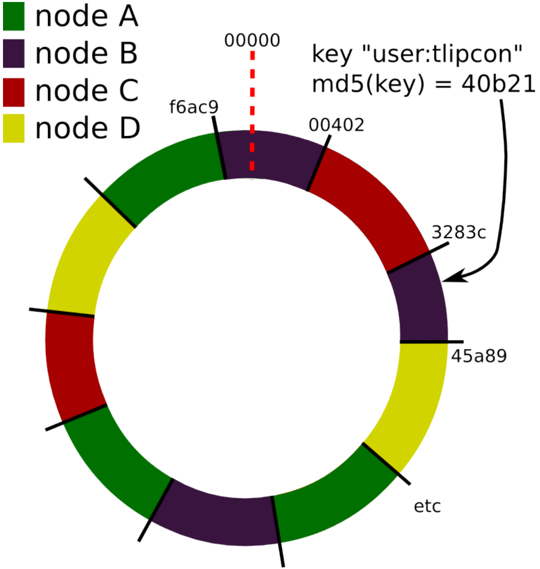
Consistent Hashing

Maintaining hashing in a dynamic cluster



Consistent Hashing

Key Placement



Consistency Models

- A consistency model determines rules for **visibility** and **apparent order** of updates.
- Example:
  - Row X is replicated on nodes M and N
  - Client A writes row X to node N
  - Some period of time t elapses.
  - Client B reads row X from node M
  - Does client B see the write from client A?
- Consistency is a continuum with tradeoffs

Strict Consistency

- All read operations must return the data from the latest completed write operation, regardless of which replica the operations went to
- Implies either:
  - All operations for a given row go to the same node (replication for availability)
  - **or** nodes employ some kind of distributed transaction protocol (eg 2 Phase Commit or Paxos)
- CAP Theorem: Strict Consistency can't be achieved at the same time as availability and partition-tolerance.

cloudera

Eventual Consistency

- As $t \longrightarrow \infty$, readers will see writes.
- In a steady state, the system is guaranteed to eventually return the last written value
- For example: DNS, or MySQL Slave Replication (log shipping)
- Special cases of eventual consistency:
  - Read-your-own-writes consistency ("sent mail" box)
  - Causal consistency (if you write Y after reading X, anyone who reads Y sees X)
  - gmail has RYOW but not causal!

Timestamps and Vector Clocks

Determining a history of a row

- Eventual consistency relies on deciding what value a row will eventually converge to
- In the case of two writers writing at "the same" time, this is difficult
- Timestamps are one solution, but rely on synchronized clocks and don't capture causality
- *Vector clocks* are an alternative method of capturing order in a distributed system

Vector Clocks

- Definition:
  - A vector clock is a tuple $\{t_1, t_2, ..., t_n\}$ of clock values from each node
  - $v_1 < v_2$ if:
    - For all i, $v_{1i} \leq v_{2i}$
    - For at least one i, $v_{1i} < v_{2i}$
  - $v_1 < v_2$ implies global time ordering of events
- When data is written from node i, it sets t_i to its clock value.
- This allows eventual consistency to resolve consistency between writes on multiple replicas.

Data Models

What's in a row?

- Primary Key $\longrightarrow$ Value
- Value could be:
  - Blob
  - Structured (set of columns)
  - Semi-structured (set of column families with arbitrary columns, eg `linkto:<url>` in webtable)
  - Each has advantages and disadvantages
- Secondary Indexes? Tables/namespaces?

Multi-Version Storage

Using Timestamps for a 3rd dimension

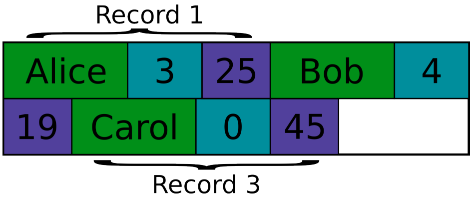
- Each table cell has a timestamp
- Timestamps don't necessarily need to correspond to real life
- Multiple versions (and tombstones) can exist concurrently for a given row
- Reads may return "most recent", "most recent before T", etc. (free snapshots)
- System may provide optimistic concurrency control with compare-and-swap on timestamps

Storage Layouts

How do we lay out rows and columns on disk?

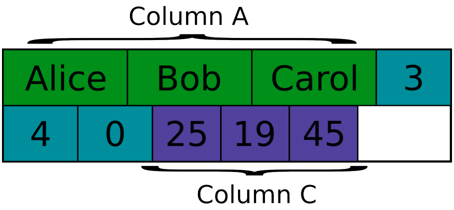
- Determines performance of different access patterns
- Storage layout maps directly to disk access patterns
- Fast writes? Fast reads? Fast scans?
- Whole-row access or subsets of columns?

Row-based Storage



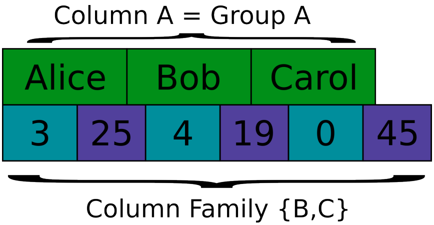
- **Pros:**
  - Good locality of access (on disk and in cache) of different columns
  - Read/write of a single row is a single IO operation.
- **Cons:**
  - But if you want to scan only one column, you still read all.

Columnar Storage



- **Pros:**
  - Data for a given column is stored sequentially
  - Scanning a single column (eg aggregate queries) is fast
- **Cons:**
  - Reading a single row may seek once per column.

Columnar Storage with Locality Groups



- Columns are organized into families ("locality groups")
- Benefits of row-based layout within a group.
- Benefits of column-based - don't have to read groups you don't care about.

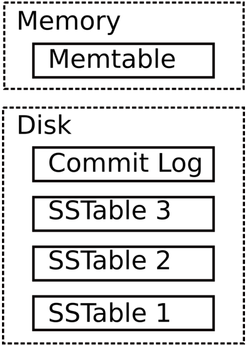
Log Structured Merge Trees

aka "The BigTable model"

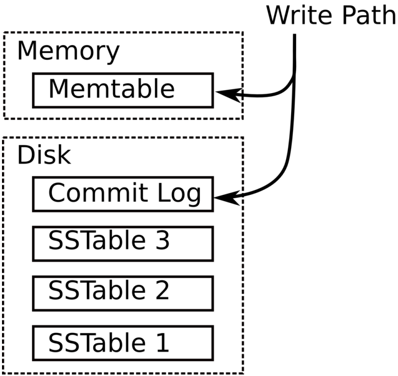
- Random IO for writes is bad (and impossible in some DFSs)
- LSM Trees convert random writes to sequential writes
- Writes go to a commit log and in-memory storage (Memtable)
- The Memtable is occasionally flushed to disk (SSTable)
- The disk stores are periodically compacted

P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). Acta Informatica. 1996.

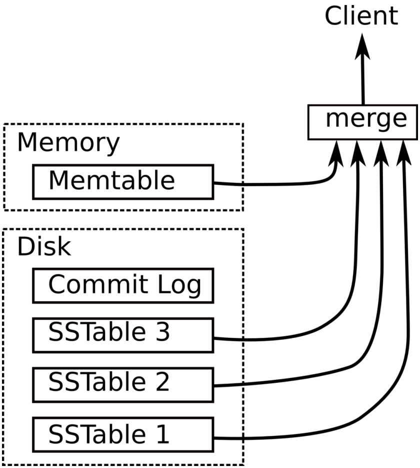
LSM Data Layout



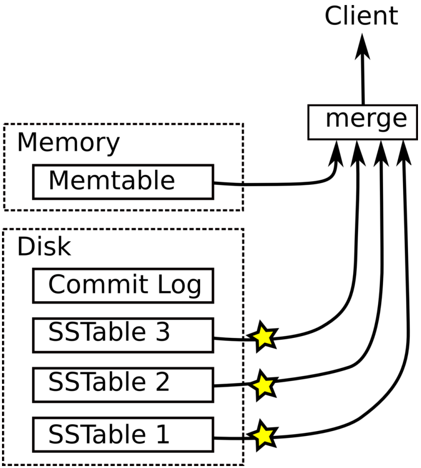
LSM Write Path



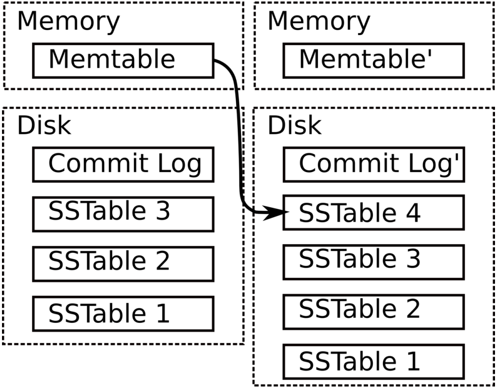
LSM Read Path



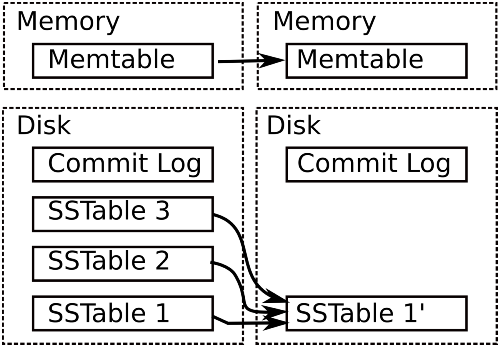
LSM Read Path + Bloom Filters



LSM Memtable Flush



LSM Compaction



Cluster Management

- Clients need to know where to find data (consistent hashing tokens, etc)
- Internal nodes may need to find each other as well
- Since nodes may fail and recover, a configuration file doesn't really suffice
- We need a way of keeping some kind of consistent view of the cluster state

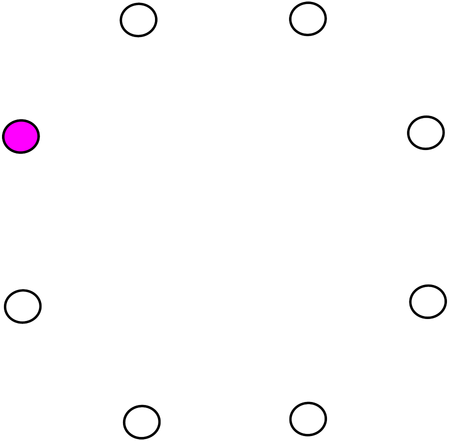
Omniscient Master

- When nodes join/leave or change state, they talk to a master
- That master holds the authoritative view of the world
- **Pros:** simplicity, single consistent view of the cluster
- **Cons:** potential SPOF unless master is made highly available. Not partition-tolerant.

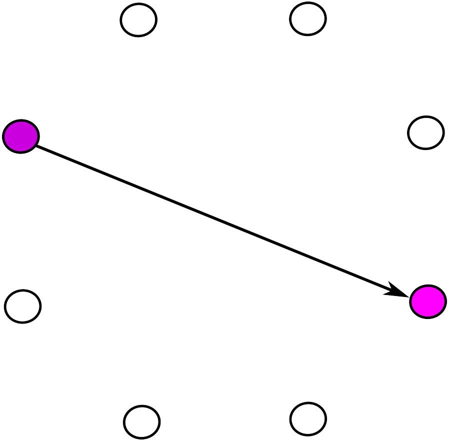
Gossip

- Gossip is one method to propagate a view of cluster status.
- Every t seconds, on each node:
  - The node selects some other node to chat with.
  - The node reconciles its view of the cluster with its gossip buddy.
  - Each node maintains a "timestamp" for itself and for the most recent information it has from every other node.
- Information about cluster state spreads in $O(lgn)$ rounds (eventual consistency)
- Scalable and no SPOF, but state is only *eventually* consistent

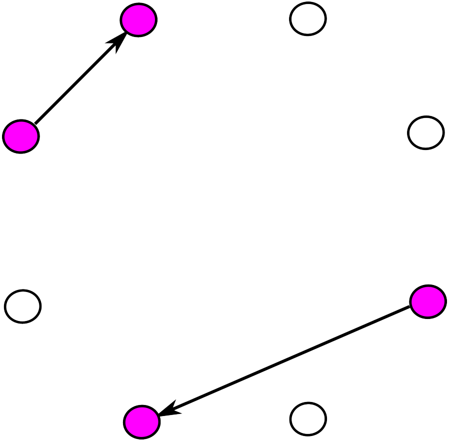
Gossip - Initial State



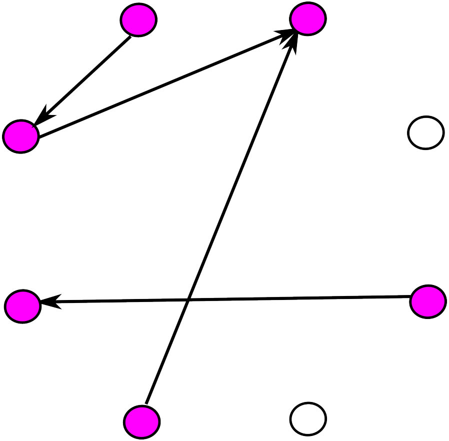
Gossip - Round 1



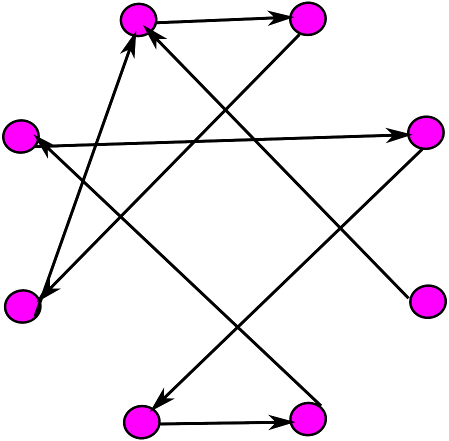
Gossip - Round 2



Gossip - Round 3



Gossip - Round 4



Questions to Ask Presenters

Scalability and Reliability

- What are the scaling bottlenecks? How does it react when overloaded?
- Are there any single points of failure?
- When nodes fail, does the system maintain availability of all data?
- Does the system automatically re-replicate when replicas are lost?
- When new nodes are added, does the system automatically rebalance data?

Performance

- What's the goal? Batch throughput or request latency?
- How many seeks for reads? For writes? How many net RTTs?
- What 99th percentile latencies have been measured in practice?
- How do failures impact serving latencies?
- What throughput has been measured in practice for bulk loads?

Consistency

- What consistency model does the system provide?
- What situations would cause a lapse of consistency, if any?
- Can consistency semantics be tweaked by configuration settings?
- Is there a way to do compare-and-swap on row contents for optimistic locking? Multirow?

Cluster Management and Topology

- Does the system have a single master? Does it use gossip to spread cluster management data?
- Can it withstand network partitions and still provide some level of service?
- Can it be deployed across multiple datacenters for disaster recovery?
- Can nodes be commissioned/decomissioned automatically without downtime?
- Operational hooks for monitoring and metrics?

Data Model and Storage

- What data model and storage system does the system provide?
- Is it pluggable?
- What IO patterns does the system cause under different workloads?
- Is the system best at random or sequential access? For read-mostly or write-mostly?
- Are there practical limits on key, value, or row sizes?
- Is compression available?

Data Access Methods

- What methods exist for accessing data? Can I access it from language X?
- Is there a way to perform filtering or selection at the server side?
- Are there bulk load tools to get data in/out efficiently?
- Is there a provision for data backup/restore?

Real Life Considerations

(I was talking about fake life in the first 45 slides)

- Who uses this system? How big are the clusters it's deployed on, and what kind of load do they handle?
- Who develops this system? Is this a community project or run by a single organization? Are outside contributions regularly accepted?
- Who supports this system? Is there an active community who will help me deploy it and debug issues? Docs?
- What is the open source license?
- What is the development roadmap?

Questions?

http://cloudera-todd.s3.amazonaws.com/nosql.pdf