

Distributed Storage Systems

Theory and practice

Pietro Michiardi

Eurecom

Introduction

Overview

- The CAP Theorem
- Amazon Dynamo
- Apache HBase
- Apache Cassandra

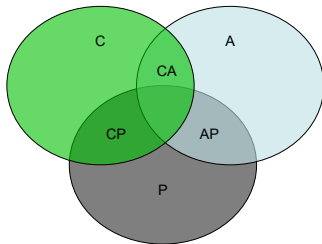
The CAP Theorem

The CAP Theorem

- **Frequently cited distributed systems theorem**
- **Relates the following three properties**
 - ▶ **C: Consistency**
 - ★ One-copy semantics, linearizability, atomicity, total-order
 - ★ Every operation must appear to take effect in a single indivisible point in time between its invocation and response
 - ▶ **A: Availability**
 - ★ Every client's request is served (receives a response) unless a client fails (despite a strict subset of server nodes failing)
 - ▶ **P: Partition-tolerance**
 - ★ A system functions properly even if the network is allowed to lose arbitrarily many messages sent from one node to another

The CAP Theorem

- In the folklore interpretation, the theorem says:
C, A, P: pick two



Precautions: be careful with CA

- **Sacrificing P (partition tolerance)**
- **Negating:** A system functions properly even if the network is allowed to lose arbitrarily many messages sent from one node to another
- **Yields:** A system **does not** function properly even if the network is allowed to lose arbitrarily many messages sent from one node to another
 - ▶ This implies sacrificing C or A, *i.e.*, the system does not work

Precautions: be careful with CA

- **Negating P:** A system function properly if the network is **not allowed** to lose arbitrarily many messages
- **However, in practice:** It is not possible to choose whether the network will lose messages! This either happens or not
- **One can argue that not “arbitrarily” many messages will be lost**
 - ▶ But “a lot” of them might be (before the network repairs)
 - ▶ In the meantime, either C or A is sacrificed

CAP in practice

- **In practical distributed systems:**

- ▶ Partitions may occur
- ▶ This is not under your control, as a system designer

- **Designer's choice:**

- ▶ **You choose** whether you want your system in C or A, when/if (temporary) partitions occur

- **In summary:**

- ▶ CAP is a fundamental theorem stating the tradeoffs among different system properties
- ▶ **Practical distributed systems are either in CP or AP**
- ▶ **The choice (C vs. A) depends on your application logic**

CAP in theory

- **Historical notes:**

- ▶ First stated by Eric Brewer at the PODC 2000 keynote
- ▶ Formally proved by Gilbert and Lynch, 2002

- **GL Theorems:**

- ▶ Asynchronous / partially synchronous network models
- ▶ Read/Write data objects
- ▶ Finer definitions of Availability and Consistency

- **Further readings:**

- ▶ (Fischer, Lynch and Patterson) FLP impossibility result
- ▶ t-connected CAP

CAP: some illustrative choices

- **CP:**

- ▶ BigTable (Google), HBase, MongoDB, Redis, Memcachedb, ...
- ▶ (sometimes classified in CA) Paxos, Zookeeper, RDBMSs, ...

- **AP:**

- ▶ Amazon Dynamo, CouchDB, Cassandra, SimpleDB, Riak, Voldemort (LinkedIn), ...

Amazon Dynamo

Amazon Web Services

- **Amazon's cloud computing services**

- ▶ S3, EC2, RedShift, SimpleDB, Elastic MR, and many, many more
- ▶ Combined, they allow constructing Internet-scale applications

- **Infrastructure services requirements:**

- ▶ Security, scalability, availability, performance, cost-efficiency
- ▶ Serve millions of customers worldwide, continuously

Amazon Web Services

- **Important observations**

- ▶ No emphasis on consistency
- ▶ AWS is in AP, sacrificing consistency

- **AWS follows the BASE philosophy**

- ▶ BASE vs. ACID
- ▶ Basically Available
- ▶ Soft state
- ▶ Eventually consistent

Why favoring Availability over Consistency?

- **Even the shortest outage has significant financial consequences and impact customer trust**
- **Clearly, consistency violations may as well have a big impact**
 - ▶ But not in several Amazon's services
 - ▶ **Billing** is a separate story

Amazon Dynamo

- **Works behind the scenes in the context of AWS**

- ▶ Used to power client-facing services such as S3, and others
- ▶ Used to power internal Amazon services such as: shopping cart, customer session management, product catalog, recommendations, order fulfillment, sales rank, fraud detection, ...

- **What is Dynamo?**

- ▶ Highly available key-value storage system
- ▶ Favors availability over consistency under failures

What is a key-value store?

- **Think about Hash tables or dictionaries**

- ▶ Simple API: **get(key)**, **put(key, value)**
- ▶ Sometimes referred to read/write operations

- **Specifics of Dynamo API**

- ▶ Uses an additional argument to pass a “context”
- ▶ Context holds critical metadata
- ▶ Typically stores **small objects** (< 1 MB)

- **Specifics of services using Dynamo**

- ▶ Do not need transactions
- ▶ Often need only primary-key access to data

Amazon Dynamo: Features

● Main characteristics

- ▶ Low latency
- ▶ Scalable (hundreds of machines)
- ▶ Always-on available (especially for writes)
- ▶ Partition/Fault tolerance
- ▶ **Eventually** consistent

● How such features are obtained

- ▶ General distributed systems toolbox
- ▶ We review some of them here

Amazon Dynamo: Key Techniques (1)

- **Consistent hashing** [Karger97]
 - ▶ For data partitioning, replication and load balancing
- **Sloppy Quorums**
 - ▶ Boosts availability in presence of failures
 - ▶ May result in inconsistent versions of keys (data)

Amazon Dynamo: Key Techniques (2)

- **Vector clocks** [Fidge88/Mantern88]
 - ▶ For tracking causal dependencies among different versions of the same key (data)
- **Gossip-based group membership**
 - ▶ For maintaining information about alive nodes
- **Anti-entropy protocol based on Merkle trees**
 - ▶ Background synchronization of divergent replicas

Amazon Dynamo: Design Decisions

- **Always writable data store**

- ▶ E.g., think shopping cart service

- **How to handle data changes?**

- ▶ Replication, required for fault/disaster tolerance
- ▶ Allow multiple versions of data
- ▶ Reconcile and resolve conflicts **during reads**

- **How to reconcile data?**

- ▶ Application-side: depending on business logic
- ▶ Dynamo: deterministic, e.g., “last-write” wins

Amazon Dynamo: Architecture

Amazon Dynamo Architecture

- **Scalable and robust components for:**
 - ▶ **Load balancing and data partitioning**
 - ▶ **Membership, fault detection**
 - ▶ Failure recovery
 - ▶ **Replica synchronization**
 - ▶ Overload Handling
 - ▶ State transfer
 - ▶ Concurrency management
 - ▶ Scheduling
 - ▶ Request marshalling and routing
 - ▶ System monitoring
 - ▶ Configuration management

Amazon Dynamo: Data Partitioning

• Data partitioning

- ▶ Dynamic partitioning of **keys** over a set of storage **nodes**
- ▶ Technique used for DHTs, e.g., Chord

• Consistent Hashing

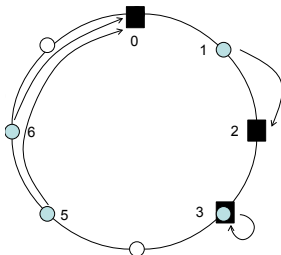
- ▶ Hashes of keys give key m -bit identifiers
- ▶ Hashes of nodes give m -bit identifiers
- ▶ Identifiers are ordered in an identifier circle

• Key assignment to storage nodes

- ▶ A key is assigned to the closest **successor node** ID
- ▶ Key k is assigned to the first node whose ID $\geq k$
- ▶ If such node does not exist, navigate the circle and find node with the smallest ID

Consistent Hashing Example

- Assume: $m = 3$ bit, 3 storage nodes (0,2,3), 4 keys (1,3,5,6)



Consistent Hashing: Key Properties (1)

- **Dynamic membership management**

- ▶ Storage nodes can come and go
- ▶ Allows incremental scalability

- **Storage node arrival/departures**

- ▶ n Joins: all keys previously assigned to node n 's successor are now assigned to n
- ▶ n Leaves: all keys currently assigned to node n are assigned to its successor

Consistent Hashing: Key Properties (2)

- **Load balancing** [Karger97]

- ▶ Each node is responsible for at most $(1 + \epsilon)K/N$ keys
- ▶ When a new node joins, only $O(K/n)$ keys must be moved (optimal)

- **Virtual Nodes**

- ▶ Each physical storage node mapped multiple times to the circle
- Improves load balancing
- Allows heterogeneous storage nodes

Amazon Dynamo: Data Replication

- **Goal: achieve high availability and durability**

- ▶ Each data item (key) replicated at N nodes
- ▶ Virtual nodes: same physical node skipped
- ▶ N is a configurable parameter per Dynamo instance

- **Example:**

- ▶ Assume $N = 3$
 - ▶ For key k , B is the “coordinator” node
 - ▶ B replicates k to $N - 1$ other successor nodes (C and D)
- B, C, D are a **preference list** for k

Amazon Dynamo: Data Versioning (1)

- **Data replication performed after an ACK is sent to a client put request**
 - ▶ **Asynchronous replication**
 - ▶ May result in inconsistencies under partitions
 - Read does not return the last value
- **Operations should not be lost!**
 - ▶ “Add to cart” should not be rejected but also not forgotten
 - ▶ If it is performed when the latest version is not available, then it is performed on a stale version of the data
 - We may have different version of a key/value pair

Amazon Dynamo: Data Versioning (2)

- **Precautions**

- ▶ Once a partition heals, versions are merged
- ▶ New versions subsume previous ones
- ▶ Applications **must be designed** with data versioning in mind

- **Key technique for versioning**

- ▶ Vector clocks
- ▶ Capture causality between different versions of an object

Vector Clocks (in Dynamo) (1)

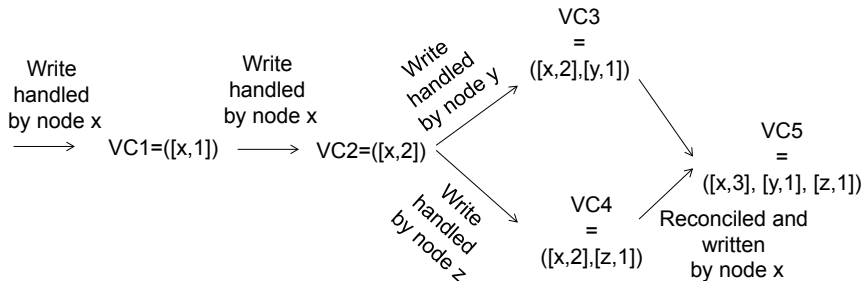
- **In theory:**

- ▶ Each `write` to a key k associated to a vector clock $VC(k)$
- ▶ $VC(k)$ is an array (map) of integers
- ▶ In theory, one entry $VC(k)[i]$ for each storage node i
- ▶ When node i handles a write for key k it increments $VC(k)[i]$

- **In practice:**

- ▶ $VC(k)$ will not have many entries \rightarrow only node from the preference list should have entries
- ▶ Dynamo truncates entries if more than a threshold

Vector Clocks (in Dynamo) (2)



NB: one VC per key

Anatomy of put and get operations

- **Storage nodes can receive requests for **any** key**

- ▶ Generic load balancer may chose a random node, not necessarily the coordinator
- ▶ Application may directly contact the coordinator in a preference list

- **Request routing**

- ▶ Node serves request only if in preference list
- ▶ Otherwise, routes the request to the first node in preference list
- ▶ 0-hop DHT routing: all nodes know all other nodes
- Not the most scalable, but excellent for low-latency

- **Extended preference list**

- ▶ Accounts for node failures

Amazon Dynamo: Quorums

- **Two important parameters**

- ▶ R : number of nodes involved in a get
- ▶ W : number of nodes involved in a put
- ▶ Quorum system: $R + W > N$, where N is the number of replicas

- **Handling put (by coordinator)**

- ▶ Generate new VC, write new version locally
- ▶ Send value, VC, to N nodes from preference list
- ▶ Wait for $W - 1$ acknowledgments

- **Handling get (by coordinator)**

- ▶ Send get to N selected nodes from preference list
- ▶ Wait for R responses
- ▶ Select highest versions using VC, reconcile/merge different versions
- ▶ Writeback reconciled version

Choosing R, W

- R, W **smaller than N**
 - ▶ To decrease latency
 - ▶ Slowest replica dictates query latency
- $W = 1$
 - ▶ Always available for writes
 - ▶ Yields $R = N \rightarrow$ reads pay the penalty
- **Typical values in Dynamo**
 - ▶ $W, R, N = 2, 2, 3$

Handling Failures

- **N selected nodes are the first N healthy nodes**
 - ▶ Might change from request to request
 - ▶ Hence the term “sloppy” quorums
- **Sloppy vs. strict quorums**
 - ▶ Allow availability under a much wider range of partitions
 - ▶ Sacrifice consistency
- **Data-center wide failures**
 - ▶ Power outages, cooling failures, network failures, ...
 - ▶ Preference lists account for this

Handling Temporary Failures

● Hinted Handoff

- ▶ If a replica in the preference list is down, then a new replica is created on a new node
- ▶ Coordinator selects a new replica node, but hints that the role is temporary
- ▶ When the new replica learns about failure recovery, it handles data to the node in the preference list

Amazon Dynamo: Anti-Entropy Synchronization

- **Uses Merkle Trees**

- ▶ A tree in which every non-leaf node is labelled with the hash of the labels of its children nodes

- **Storage nodes**

- ▶ Keep a Merkle tree for each of its key ranges (virtual nodes)
- ▶ Compare root of the tree with replicas
- ▶ If equal, replicas are in sync
- ▶ Otherwise, traverse the tree and synchronize keys that differ

Amazon Dynamo: Membership Management

- **Membership management initiated by administrator**
- **Gossip protocol to propagate membership changes**
 - ▶ Nodes contact a random node every second
 - ▶ 2 nodes reconcile membership information
 - ▶ Gossiping also used to handle metadata

Failure Detection

- **Unreliable failure detection**

- ▶ Detection is triggered by read/write requests
- ▶ Called “in-band” failure detection
- No dedicated component

- **Example:**

- ▶ With steady load on node A
- ▶ Node A periodically checks the status of nodes in the extended preference list
- ▶ Does not make the distinction between faults and partitions

Amazon Dynamo: Summary

- **Eventually consistent, highly available key value store**
 - ▶ In the CAP space, it is in AP
- **Focuses on low-latency**
 - ▶ Writes are super fast
 - ▶ Reconciliation in reads
- **Built atop of fundamental techniques in distributed systems**
 - ▶ Consistent hashing
 - ▶ Sloppy quorum-based replication
 - ▶ Merkle-tree based synchronization
 - ▶ Vector clocks, and gossip membership management

HBASE

Introduction

Why yet another storage architecture?

- **Relational Database Management Systems (RDBMS):**

- ▶ Around since 1970s
- ▶ Countless examples in which they actually do make sense

- **The dawn of Big Data:**

- ▶ Previously: ignore data sources because no cost-effective way to store everything
 - ★ One option was to prune, by retaining only data for the last N days
- ▶ Today: store everything!
 - ★ Pruning fails in providing a base to build useful mathematical models

Batch processing

- **Hadoop and MapReduce:**

- ▶ Excels at storing (semi- and/or un-) structured data
- ▶ Data interpretation takes place at analysis-time
- ▶ Flexibility in data classification

- **Batch processing: A complement to RDBMS**

- ▶ Scalable sink for data, processing launched when time is right
- ▶ Optimized for large file storage
- ▶ Optimized for “streaming” access

- **Random Access:**

- ▶ Users need to “interact” with data, especially that “crunched” after a MapReduce job
- ▶ This is historically where RDBMS excel: random access for structured data

Column-Oriented Databases

- **Data layout:**

- ▶ Save their data grouped by columns
- ▶ Subsequent column values are stored contiguously on disk
- ▶ This is substantially different from traditional RDBMS, which save and store data by row

- **Specialized databases for specific workloads:**

- ▶ Reduced I/O
- ▶ Better suited for compression → Efficient use of bandwidth
 - ★ Indeed, column values are often very similar and differ little row-by-row
- ▶ Real-time access to data

- **Important NOTE:**

- ▶ HBase is not a column-oriented DB in the typical term
- ▶ HBase uses an on-disk column storage format
- ▶ Provides key-based access to specific cell of data, or a sequential range of cells

Column-Oriented and Row-Oriented storage layouts

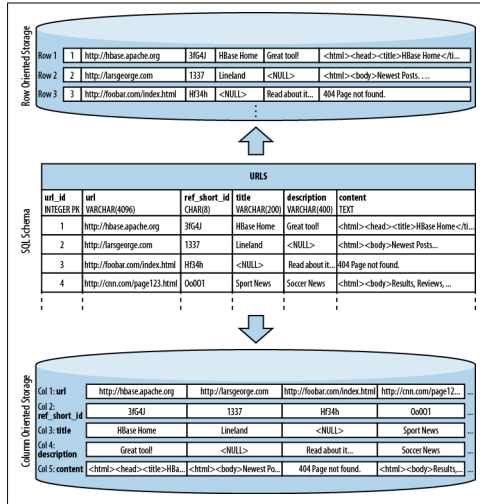


Figure: Example of Storage Layouts

The Problem with RDBMS

- **RDBMS are still relevant**

- ▶ Persistence layer for frontend application
- ▶ Store relational data
- ▶ Works well for a limited number of records

- **Example: Hush**

- ▶ Used throughout this course
- ▶ URL shortener service

- **Let's see the “scalability story” of such a service**

- ▶ Assumption: service must run with a reasonable budget

The Problem with RDBMS

- **Few thousands users: use a LAMP stack**

- ▶ *Normalize data*
- ▶ Use foreign keys
- ▶ Use Indexes

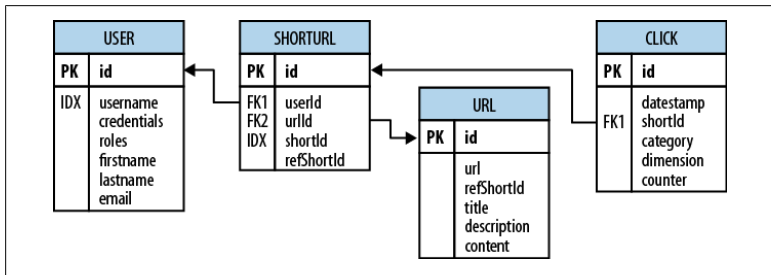


Figure: The Hush Schema expressed as an ERD

The Problem with RDBMS

- **Find all short URLs for a given user**

- ▶ JOIN `user` and `shorturl` tables
- ▶ Use the `WHERE` clause to select the given user

- **Stored Procedures**

- ▶ Consistently update data from multiple clients
- ▶ Underlying DB system guarantees coherency

- **Transactions**

- ▶ Make sure you can update tables in an *atomic* fashion
- ▶ RDBMS → *Strong Consistency* (ACID properties)
- ▶ *Referential Integrity*

The Problem with RDBMS

- **Scaling up to tens of thousands of users**

- ▶ Increasing pressure on the database server
- ▶ Adding more application servers is easy: they share their state on the same central DB
- ▶ CPU and I/O start to be a problem on the DB

- **Master-Slave architecture**

- ▶ Add DB server so that `READS` can be served in parallel
- ▶ Master DB takes all the writes (which are fewer in the Hush application)
- ▶ Slaves DB replicate Master DB and serve all reads (but you need a load balancer)

The Problem with RDBMS

● Scaling up to hundreds of thousands

- ▶ `READS` are still the bottlenecks
- ▶ Slave servers begin to fall short in serving clients requests

● Caching

- ▶ Add a caching layer, e.g. Memcached or Redis
- ▶ Offload `READS` to a fast in-memory system
- You lose consistency guarantees
- Cache invalidation is critical for having DB and Caching layer consistent

The Problem with RDBMS

● Scaling up more

- ▶ `WRITES` are the bottleneck
- ▶ The master DB is hit too hard by `WRITE` load
- ▶ *Vertical scalability*: beef up your master server
- This becomes costly, as you may also have to replace your RDBMS

● SQL JOINS becomes a bottleneck

- ▶ Schema de-normalization
- ▶ Cease using stored procedures, as they become slow and eat up a lot of server CPU
- ▶ Materialized views (they speed up `READS`)
- ▶ Drop secondary indexes as they slow down `WRITES`

The Problem with RDBMS

- **What if your application needs to further scale up?**
 - ▶ Vertical scalability vs. Horizontal scalability
- **Sharding**
 - ▶ Partition your data across multiple databases
 - ★ Essentially you break horizontally your tables and ship them to different servers
 - ★ This is done using fixed boundaries
 - Re-sharding to achieve load-balancing
 - This is an operational nightmare
 - ▶ Re-sharding takes a huge toll on I/O resources

Non-Relational DataBases

- **They originally do not support SQL**

- ▶ In practice, this is becoming a thin line to make the distinction
- ▶ One difference is in the data model
- ▶ Another difference is in the consistency model (ACID and transactions are generally sacrificed)

- **Consistency models and the CAP Theorem**

- ▶ Strict: all changes to data are atomic
- ▶ Sequential: changes to data are seen in the same order as they were applied
- ▶ Causal: causally related changes are seen in the same order
- ▶ Eventual: updates propagates through the system and replicas when in steady state
- ▶ Weak: no guarantee

Dimensions to classify NoSQL DBs

• Data model

- ▶ How the data is stored: key/value, semi-structured, column-oriented, ...
- ▶ How to access data?
- ▶ Can the schema evolve over time?

• Storage model

- ▶ In-memory or persistent?
- ▶ How does this affect your access pattern?

• Consistency model

- ▶ Strict or eventual?
- ▶ This translates in how fast the system handles `READS` and `WRITES` [?]

Dimensions to classify NoSQL DBs

- **Physical Model**

- ▶ Distributed or single machine?
- ▶ How does the system scale?

- **Read/Write performance**

- ▶ Top-down approach: understands well the workload!
- ▶ Some systems are better for `READS`, other for `WRITES`

- **Secondary indexes**

- ▶ Does your workload require them?
- ▶ Can your system emulate them?

Dimensions to classify NoSQL DBs

● Failure Handling

- ▶ How each data store handle server failures?
- ▶ Is it able to continue operating in case of failures?
 - ★ This is related to Consistency models and the CAP theorem
- ▶ Does the system support “hot-swap”?

● Compression

- ▶ Is the compression method pluggable?
- ▶ What type of compression?

● Load Balancing

- ▶ Can the storage system seamlessly balance load?

Dimensions to classify NoSQL DBs

- **Atomic read-modify-write**

- ▶ Easy in a centralized system, difficult in a distributed one
- ▶ Prevent race conditions in multi-threaded or shared-nothing designs
- ▶ Can reduce client-side complexity

- **Locking, waits and deadlocks**

- ▶ Support for multiple client accessing data simultaneously
- ▶ Is locking available?
- ▶ Is it wait-free, hence deadlock free?

Impedance Match

“One-size-fits-all” has been long dismissed: need to find the perfect match for your problem.

Database (De-)Normalization

- **Schema design at scale**

- ▶ A good methodology is to apply the DDI principle [?]
 - ★ Denormalization
 - ★ Duplication
 - ★ Intelligent Key design

- **Denormalization**

- ▶ Duplicate data in more than one table such that at `READ` time no further aggregation is required

- **Next: an example based on Hush**

- ▶ How to convert a classic relational data model to one that fits HBase
- ▶ This example will be covered in the LAB session 3

Example: Hush - from RDBMS to HBase

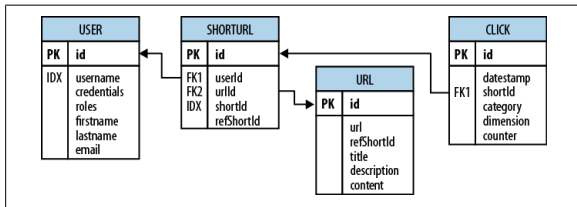


Figure: The Hush Schema expressed as an ERD

- `shorturl` table: contains the short URL
- `click` table: contains click tracking, and other statistics, aggregated on a daily basis (essentially, a counter)
- `user` table: contains user information
- `URL` table: contains a replica of the page linked to a short URL, including META data and content (this is done for batch analysis purposes)

Example: Hush - from RDBMS to HBase

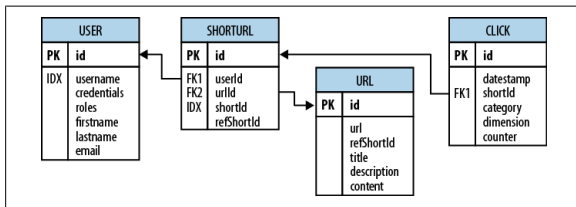


Figure: The Hush Schema expressed as an ERD

- `user` table is indexed on the `username` field, for fast user lookup
- `shorturl` table is indexed on the short URL (`shortId`) field, for fast short URL lookup

Example: Hush - from RDBMS to HBase

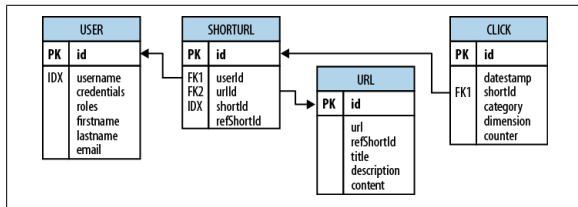


Figure: The Hush Schema expressed as an ERD

- `shorturl` and `user` tables are related through a foreign key relation on the `userId`
- `URL` table is related to `shorturl` table with a foreign key on the `URL id`
- `click` table is related to `shorturl` table with a foreign key on the `short URL id`
- **NOTE:** a web page is stored only once (even if multiple users link to it), but each users maintain separate statistics

Example: Hush - from RDBMS to HBase

Table: shorturl		
Row Key:	shortId	
Family:	data:	Columns: url, refShortId, userId, clicks
	stats-daily: [ttl: 7days]	Columns: YYYYMMDD, YYYYMMDD\x00<country-code>
	stats-weekly: [ttl: 4weeks]	Columns: YYYYWW, YYYYWW\x00<country-code>
	stats-monthly: [ttl: 12months]	Columns: YYYYMM, YYYYMM\x00<country-code>

Table: url		
Row Key:	MDS(url)	
Family:	data: [compressed]	Columns: refShortId, title, description
	content: [compressed]	Columns: raw

Table: user-shorturl		
Row Key:	username\x00shortId	
Family:	data:	Columns: timestamp

Table: user		
Row Key:	username	
Family:	data:	Columns: credentials, roles, firstname, lastname, email

- `shorturl` table: stores each short URL, usage statistics (various time-ranges in separate *column-families* with distinct *TTL* settings)
 - ▶ Note the dimensional postfix appended to the time information
- `url` table: stores the downloaded page, and the extracted details
 - ▶ This table uses compression

Figure: The Hush Schema in HBase

Example: Hush - from RDBMS to HBase

Table: shorturl		
Row Key:	shortId	
Family:	data:	Columns: url, refShortId, userId, clicks
	stats-daily: [ttl: 7days]	Columns: YYYYMMDD, YYYYMMDD\x00<country-code>
	stats-weekly: [ttl: 4weeks]	Columns: YYYYWW, YYYYWW\x00<country-code>
	stats-monthly: [ttl: 12months]	Columns: YYYYMM, YYYYMM\x00<country-code>

Table: url		
Row Key:	MD5(url)	
Family:	data: [compressed]	Columns: refShortId, title, description
	content: [compressed]	Columns: raw

Table: user-shorturl		
Row Key:	username\x00shortId	
Family:	data:	Columns: timestamp

Table: user		
Row Key:	username	
Family:	data:	Columns: credentials, roles, firstname, lastname, email

- `user-shorturl` table: this is a lookup table (basically an index) to find all shortIDs for a given user

- ▶ Note that this table is filled at *insert time*, it's not automatically generated by HBase

- `user` table: stores user details

Figure: The Hush Schema in HBase

Example: Hush - RDBMS vs HBase

● Same number of tables

- ▶ Their meaning is different
- ▶ `click` table has been absorbed by the `shorturl` table
- ▶ statistics are stored with the date as the key, so that they can be accessed *sequentially*
- ▶ The `user-shorturl` table is replacing the foreign key relationship, making user-related lookups faster

● Normalized vs. De-normalized data

- ▶ Wide tables and column-oriented design eliminates `JOINS`
- ▶ *Compound keys* are essential
- ▶ Data partitioning is based on keys, so a proper understanding thereof is essential

HBase building blocks

- **The backdrop: BigTable**

- ▶ GFS, The Google FileSystem [?]
- ▶ Google MapReduce [?]
- ▶ BigTable [?]

- **What is BigTable?**

- ▶ BigTable is a distributed storage system for managing structured data designed to scale to a very large size
- ▶ BigTable is a sparse, distributed, persistent multi-dimensional sorted map

- **What is HBase?**

- ▶ Essentially it's an open-source version of BigTable
- ▶ Differences listed in [?]

HBase building blocks

Tables, Rows, Columns, and Cells

- **The most basic unit in HBase is a *column***

- ▶ Each column may have multiple versions, with each distinct value contained in a separate *cell*
- ▶ One or more columns form a *row*, that is addressed uniquely by a *row key*

- A table is a collection of rows

- ▶ All rows are always *sorted lexicographically* by their row key

```
hbase(main):001:0> scan 'table1'
ROW                                COLUMN+CELL
row-1                             column=cf1:, timestamp=1297073325971 ...
row-10                             column=cf1:, timestamp=1297073337383 ...
row-11                             column=cf1:, timestamp=1297073340493 ...
row-2                             column=cf1:, timestamp=1297073329851 ...
row-22                             column=cf1:, timestamp=1297073344482 ...
row-3                             column=cf1:, timestamp=1297073333504 ...
row-abc                           column=cf1:, timestamp=1297073349875 ...
7 row(s) in 0.1100 seconds
```

HBase building blocks

Tables, Rows, Columns, and Cells

- **Lexicographical ordering of row keys**

- ▶ Keys are compared on a binary level, byte by byte, from left to right
- ▶ This can be thought of as a primary index on the row key!
- ▶ Row keys are *always unique*
- ▶ Row keys can be any *arbitrary array of bytes*

- **Columns**

- ▶ Rows are composed of columns
- ▶ Can have millions of columns
- ▶ Can be compressed or tagged to stay in memory

HBase building blocks

Tables, Rows, Columns, and Cells

● Column Families

- ▶ Columns are grouped into *column families*
- Semantical boundaries between data
- ▶ Column families and columns stored together in the same low-level storage file, called an *HFile*
- ▶ Defined when table is created
- ▶ Should not be changed too often
- ▶ The number of column families should be reasonable [WHY?]
- ▶ Column family name composed by printable characters

● References to columns

- ▶ Column “name” is called *qualifier*, and can be any arbitrary number of bytes
- ▶ Reference: `family:qualifier` (also called the **column key**)

HBase building blocks

Tables, Rows, Columns, and Cells

- **A note on the `NULL` value**

- ▶ In RDBMS `NULL` cells need to be set and occupy space
- ▶ In HBase, `NULL` cells or columns are simply not stored

- **A *cell***

- ▶ Every column value, or cell, is timestamped (implicitly or explicitly)
 - ★ This can be used to save multiple versions of a value that changes over time
 - ★ Versions are stored in decreasing timestamp, most recent first
- ▶ Cell versions can be constrained by *predicate deletions*
 - ★ Keep only values from the last week

HBase building blocks

Tables, Rows, Columns, and Cells

- **Access to data**

- ▶ (Table, RowKey, Family, Column, Timestamp) → Value
- ▶ `SortedMap<RowKey, List<SortedMap<Column, List<Value, Timestamp>>>>`
- ▶ The first `SortedMap` is the table, containing a `List` of column families
- ▶ The families contain another `SortedMap`, representing columns and a `List` of value, timestamp tuples

- **A note on consistency:**

- ▶ Row data access is **atomic** and includes any number of columns
 - ▶ There is no further guarantee or transactional feature spanning multiple rows
- HBase is strictly consistent

HBase building blocks

Automatic Sharding

● Region

- ▶ This is the basic unit of scalability and load balancing
- ▶ Regions are contiguous ranges of rows “stored together” → they are the equivalent of *range partitions* in sharded RDBMS
- ▶ Regions are *dynamically split* by the system when they become too large
- ▶ Regions can also be merged to reduce the number of storage files

● Regions in practice

- ▶ Initially, there is one region
- ▶ System monitors region size: if a threshold is attained, `SPLIT`
 - ★ Regions are split in two at the *middle key*
 - ★ This creates roughly two equivalent (in size) regions

HBase building blocks

Automatic Sharding

- **Region Servers**

- ▶ Each region is served by *exactly one Region Server*
- ▶ Region servers can serve multiple regions
- ▶ The number of region servers and their sizes depend on the capability of a single region server

- **Server failures**

- ▶ Regions allow for fast recovery upon failure
- ▶ Fine-grained Load Balancing is also achieved using regions as they can be easily moved across servers

HBase building blocks

Storage API

- **No support for SQL**

- ▶ CRUD operations using a standard API, available for many “clients”
- ▶ Data access is not declarative but imperative

- **Scan API**

- ▶ Allows for fast iteration over ranges of rows
- ▶ Allows to limit the number and which column are returned
- ▶ Allows to control the version number of each cell

- **Read-modify-write API**

- ▶ HBase supports single-row transactions
- ▶ Atomic read-modify-write on data stored in a single row key

HBase building blocks

Storage API

● Counters

- ▶ Values can be interpreted as counters and **updated atomically**
- ▶ Can be read and modified in one operation
- Implement global, strictly consistent, sequential counters

● Coprocessors

- ▶ These are equivalent to stored-procedures in RDBMS
- ▶ Allow to push user code in the address space of the server
- ▶ Access to server local data
- ▶ Implement lightweight batch jobs, data pre-processing, data summarization

HBase building blocks

HBase implementation

- **Data Storage**

- ▶ *Store* files are called `HFiles`
- ▶ Persistent and ordered **immutable** maps from key to value
- ▶ Internally implemented as sequences of blocks with an index at the end
- ▶ Index is loaded when the `HFile` is opened and kept in memory

- **Data lookups**

- ▶ Since `HFiles` have a block index, lookup can be done with a single disk seek
- ▶ First, the block possibly containing a given lookup key is determined with a **binary search** in the in-memory index
- ▶ Then a block read is performed to find the actual key

- **Underlying file system**

- ▶ Many are supported, usually HBase deployed on top of HDFS

HBase building blocks

HBase implementation

- **WRITE operation**

- ▶ First, data is written to a commit log, called WAL (write-ahead-log)
- ▶ Then data is moved into memory, in a structure called `memstore`
- ▶ When the size of the `memstore` exceeds a given threshold it is flushed to an `HFile` to disk

- **How can HBase write, while serving READS and WRITES?**

- ▶ Rolling mechanism
 - ★ new/empty slots in the `memstore` take the updates
 - ★ old/full slots are flushed to disk
- ▶ Note that data in `memstore` is sorted by keys, matching what happens in the `HFiles`

- **Data Locality**

- ▶ Achieved by the system looking up for server hostnames
- ▶ Achieved through intelligent key design

HBase building blocks

HBase implementation

• Deleting data

- ▶ Since HFiles are immutable, how can we delete data?
- ▶ A delete marker (also known as *tombstone marker*) is written to indicate that a given key is deleted
- ▶ During the read process, data marked as deleted is skipped
- ▶ Compactions (see next slides) finalize the deletion process

• READ operation

- ▶ Merge of what is stored in the `memstores` (data that is not on disk) and in the `HFiles`
- ▶ The WAL is never used in the `READ` operation
- ▶ Several API calls to read, scan data

HBase building blocks

HBase implementation

• Compactions

- ▶ Flushing data from `memstores` to disk implies the creation of new `HFiles` each time
- We end up with many (possibly small) files
- We need to do housekeeping [**WHY?**]

• Minor Compaction

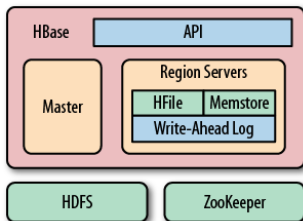
- ▶ Rewrites small `HFiles` into fewer, larger `HFiles`
- ▶ This is done using an n -way merge¹

• Major Compaction

- ▶ Rewrites all files within a column family or a region in a new one
- ▶ Drop deleted data
- ▶ Perform predicated deletion (e.g. delete old data)

¹What is MergeSort?

HBase: a glance at the architecture



- **Master node: HMaster**

- ▶ Assigns regions to region servers using ZooKeeper
- ▶ Handles load balancing
- ▶ Not part of the data path
- ▶ Holds metadata and schema

- **Region Servers**

- ▶ Handle `READs` and `WRITEs`
- ▶ Handle region splitting

Architecture

Seek vs. Transfer

- **Fundamental difference between RDBMS and alternatives**

- ▶ B+Trees
- ▶ Log-Structured Merge Trees

- **Seek vs. Transfer**

- ▶ Random access to individual cells
- ▶ Sequential access to data

B+ Trees

- **Dynamic, multi-level indexes**

- ▶ Efficient insertion, lookup and deletion
- ▶ **Q: What's the difference between a B+ Tree and a Hash Table?**
- ▶ Frequent updates may imbalance the trees → Tree optimization and re-organization is required (which is a costly operation)

- **Bounds on *page size***

- ▶ Number of keys in each branch
- ▶ Larger fanout compared to binary trees
- ▶ Lower number of I/O operations to find a specific key

- **Support for range scans**

- ▶ Leafs are linked and represent an in-order list of all keys
- ▶ No costly tree-traversal algorithms required

LSM-Trees

• Data flow

- ▶ Incoming data is first stored in a logfile, *sequentially*
- ▶ Once the log has the modification saved, data is pushed in memory
 - ★ In-memory store holds most recent updates for fast lookup
- ▶ When memory is “full”, data is flushed in a store file to disk, as a sorted list of `key` \rightarrow `record` pair
- ▶ At this point, the log file can be thrown away

• How store files are arranged

- ▶ Similar idea of a B+ Tree, but optimized for sequential disk access
- ▶ All nodes of the tree try to be filled up completely
- ▶ Updates are done in a **rolling merge** fashion
 - ★ The system packs existing on-disk multi-page blocks with in-memory data until the block reaches full capacity

LSM-Trees

● Clean-up process

- ▶ As flushes take place over time, a lot of store files are created
- ▶ Background process aggregates files into larger ones to limit disk seeks
- ▶ All store files are always sorted by key → no re-ordering required to fit new keys in

● Data Lookup

- ▶ Lookups are done in a merging fashion
 - ★ First lookup in the in-memory store
 - ★ If miss, the lookup in the on-disk store

● Deleting data

- ▶ Use a *delete marker*
- ▶ When pages are re-written, deleted markers and keys are eventually dropped
- ▶ Predicate deletion happens here

B+ Tree vs. LSM-Trees

● B+ Tree [?]

- ▶ Work well when there are not so many updates
- ▶ The more and the faster you insert data at random locations the faster pages get fragmented
- ▶ **Updates and deletes are done at disk seek rates, rather than transfer rates**

● LSM-Tree [?]

- ▶ Work at disk transfer rate and scale better to huge amounts of data
- ▶ Guarantee a consistent insert rate
 - ★ They transform random into sequential writes
- ▶ Reads are independent from writes
- ▶ Optimized data layout which offers predictable boundaries on disk seeks

Overview

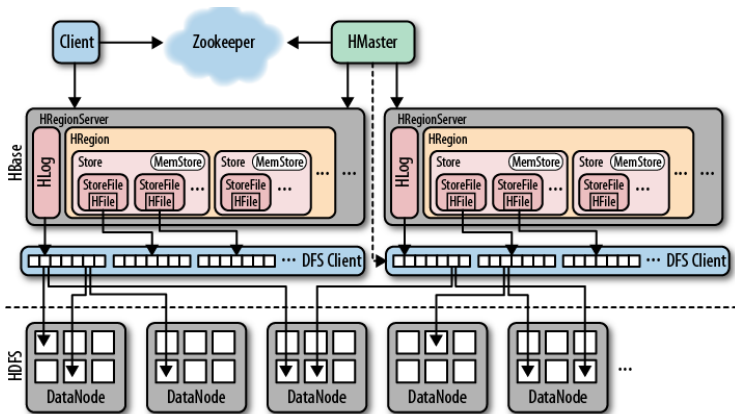


Figure: Overview of how HBase handles files in the filesystem

Storage

Overview

- **HBase handles two kinds of file types**

- ▶ One is used for the WAL
- ▶ One is used for the actual data storage

- **Who does what**

- ▶ `HMaster`
 - ★ Low-level operations
 - ★ Assigns region servers to key space
 - ★ Keeps metadata
 - ★ Talks to ZooKeeper
- ▶ `HRegionServer`
 - ★ Handles the WAL and `HFiles`
 - ★ These files are divided in to blocks and stored into HDFS
 - ★ Block size is a parameter

Storage

Overview

● General communication flow

- ▶ A client contacts ZooKeeper when trying to access a particular row
- ▶ Recovers from ZooKeeper the server name that host the `-ROOT-` region
- ▶ Using the `-ROOT-` information the client retrieves the server name that host the `.META.` table region
 - ★ The `.META.` table region contains the row key in question
- ▶ Contact the reported `.META.` server and retrieve the server name that has the region containing the row key in question

● Caching

- ▶ Generally, lookup procedures involve caching row key locations for faster subsequent lookups

Storage

Overview

● Important Java Classes

- ▶ `HRegionServer` handles one or more regions and create the corresponding `HRegion` object
- ▶ When an `HRegion` object is opened it creates a `Store` instance for each `HColumnFamily`
- ▶ Each `Store` instance can have:
 - ★ One or more `StoreFile` instances
 - ★ A `MemStore` instance
- ▶ `HRegionServer` has a shared `HLog` instance

Storage

Write Path

- **External client insert data in HBase**

- ▶ Issues an `HTable.put(Put)` request to `HRegionServer`
- ▶ `HRegionServer` hands the request to the `HRegion` instance that matches the request [Q: What is the matching criteria?]

- **How the system reacts to a write request**

- ▶ Write data to the WAL, represented by the `HLog` class
 - ★ The WAL stores `HLogKey` instances in a HDFS `SequenceFile`
 - ★ These keys contain a sequence number and the actual data
 - ★ In case of failure, this data can be used to replay not-yet-persisted data
- ▶ Copy data in the `MemStore`
 - ★ Check if `MemStore` size has reached a threshold
 - ★ If yes, launch a *flush request*
 - ★ Launch a thread in the `HRegionServer` and flush `MemStore` data to an `HFile`

Storage

HBase Files

- **What and where are HBase files (including WAL, HFile,...) stored?**
 - ▶ HBase has a root directory set to “/hbase” in HDFS
 - ▶ Files can be divided into:
 - ★ Those that reside under the HBase root directory
 - ★ Those that are in the *per-table* directories
- /hbase
 - ▶ .logs
 - ▶ .oldlogs
 - ▶ .hbase.id
 - ▶ .hbase.version
 - ▶ /example-table

Storage

HBase Files

- /example-table
 - ▶ .tableinfo
 - ▶ .tmp
 - ▶ "...Key1..."
 - ★ .oldlogs
 - ★ .regioninfo
 - ★ .tmp
 - ★ colfam1/
- colfam1/
 - ▶ "...column-key1..."

Storage

HBase: Root-level files

- **.logs directory**

- ▶ WAL files handled by `HLog` instances
- ▶ Contains a subdir for each `HRegionServer`
- ▶ Each subdir contains many `HLog` files
- ▶ All regions from that `HRegionServer` share the same `HLog` files

- **.oldlogs directory**

- ▶ When data is persisted to disk (from `Memstores`) log files are decommissioned to the `.oldlogs` dir

- **hbase.id and hbase.version**

- ▶ Represent the unique ID of the cluster and the file format version

Storage

HBase: Table-level files

- **Every table has its own directory**

- ▶ `.tableinfo`: stores the serialized `HTableDescriptor`
 - ★ This include the table and column family schema
- ▶ `.tmp` directory
 - ★ Contains temporary data

Storage

HBase: Region-level files

- **Inside each table dir, there is a separate dir for every region in the table**
 - ▶ The name of each of this dirs is the MD5 hash of a region name
 - ★ Inside each region there is a directory for each column family
 - ★ Each column family directory holds the actual data files, namely `HFiles`
 - ★ Their name is just an arbitrary random number
 - ▶ Each region directory also has a `.regioninfo` file
 - ★ Contains the serialized information of the `HRegionInfo` instance
- **Split Files**
 - ▶ Once the region needs to be split, a `splits` directory is created
 - ★ This is used to stage two daughter regions
 - ★ If split is successful, daughter regions are moved up to the table directory

Storage

HBase: A note on region splits

- **Splits triggered by store file (region) size**
 - ▶ Region is split in two
 - ▶ Region is closed to new requests
 - ▶ `.META.` is updated
- **Daughter regions initially reside on the same server**
 - ▶ Both daughters are compacted
 - ▶ Parent is cleaned up
 - ▶ `.META.` is updated
- **Master schedules new regions to be moved off to other servers**

Storage

HBase: Compaction

- **Process that takes care of re-organizing store files**
 - ▶ Essentially to conform to underlying filesystem requirements
 - ▶ Compaction check when memstore is flushed
- **Minor and Major compactions**
 - ▶ Always from the oldest to the newest files
 - ▶ Avoid all servers to perform compaction concurrently

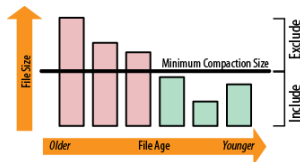


Figure: A set of store files showing the minimum compaction threshold

Storage

HFile format

- **Store files are implemented by the `HFile` class**
 - ▶ Efficient data storage is the goal
- **HFiles consist of a variable number of blocks**
 - ▶ Two fixed blocks: *info* and *trailer*
 - ▶ *index* block: records the offsets of the *data* and *meta* blocks
 - ▶ Block size: *large* → sequential access; *small* → random access

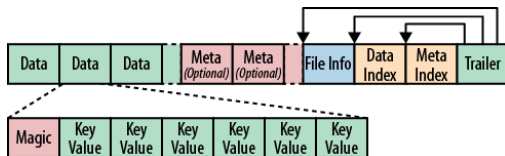


Figure: The HFile structure

Storage

HFile size and HDFS block size

- **HBase uses any underlying filesystem**
- **In case HDFS is used**
 - ▶ HDFS block size is generally 64MB
 - ▶ This is 1,024 times the default `HFile` block size (64 KB)
 - There is no correlation between HDFS block and HFile sizes

Storage

The KeyValue Format

- **Each KeyValue in the HFile is a low-level byte array**
 - ▶ It allows for *zero-copy* access to the data
- **Format**
 - ▶ Fixed-length preamble indicates the length of the key and value
 - ★ This is useful to offset into the array to get direct access to the value, ignoring the key
 - ▶ Key format
 - ★ Contains row key, column family name, column qualifier...
 - ★ [TIP]: consider small keys to avoid overhead when storing small data

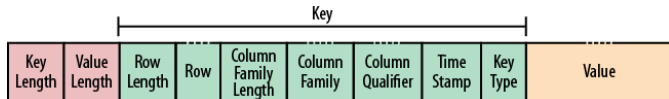


Figure: The KeyValue Format

The Write-Ahead Log

- **Main tool to ensure resiliency to failures**

- ▶ Region servers keep data in-memory until enough is collected to warrant a flush
- ▶ What if the server crashes or power is lost?

- **WAL is a common approach to address fault-tolerance**

- ▶ Every data update is first written to a log
- ▶ Log is persisted (and replicated, since it resides on HDFS)
- ▶ Only when log is written, client is notified a successful operation on data

The Write-Ahead Log

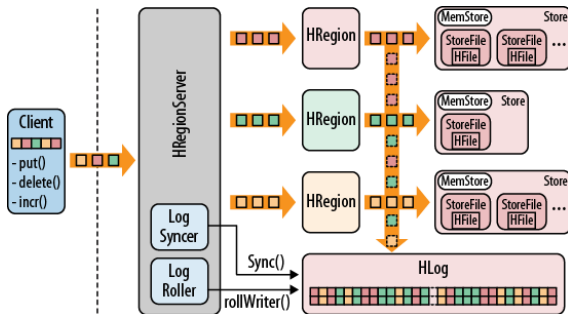
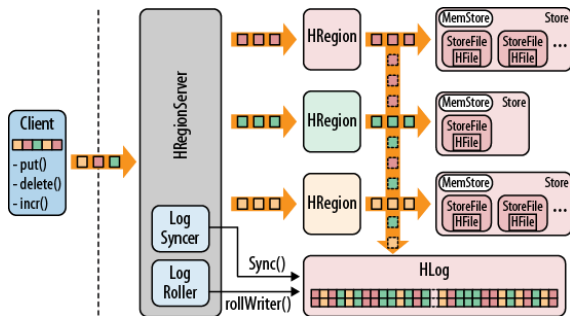


Figure: The write path of HBase

- **WAL records all changes to data**

- ▶ Can be replayed in case of server failure
- ▶ If write to WAL fails, the whole operations has to fail

The Write-Ahead Log



Write Path

- ▶ Client modifies data (`put()`, `delete()`, `increment()`)
- ▶ Modifications are wrapped into a **KeyValue** object
- ▶ Objects are batched to the corresponding **HRegionServer**
- ▶ Objects are routed to the corresponding **HRegion**
- ▶ Objects are written to **WAL** and in the **MemStore**

Read Path

- **HBase uses multiple store files per column family**
 - ▶ These can be either in-memory and/or materialized on disk
 - ▶ Compactions and clean-up background processes take care of store files maintenance
 - ▶ Store files are immutable, so deletion is handled in a special way
- **The anatomy of a get command**
 - ▶ HBase uses a `QueryMatcher` in combination with a `ColumnTracker`
 - ▶ First, an exclusion check is performed to filter skip files (and eventually tombstone labelled data)
 - ▶ Scanning data is implemented by a `RegionScanner` class which retrieves a `StoreScanner`
 - ▶ `StoreScanner` includes both the `MemStore` and `HFiles`
 - ▶ Read/Scans happen in the same order as data is saved

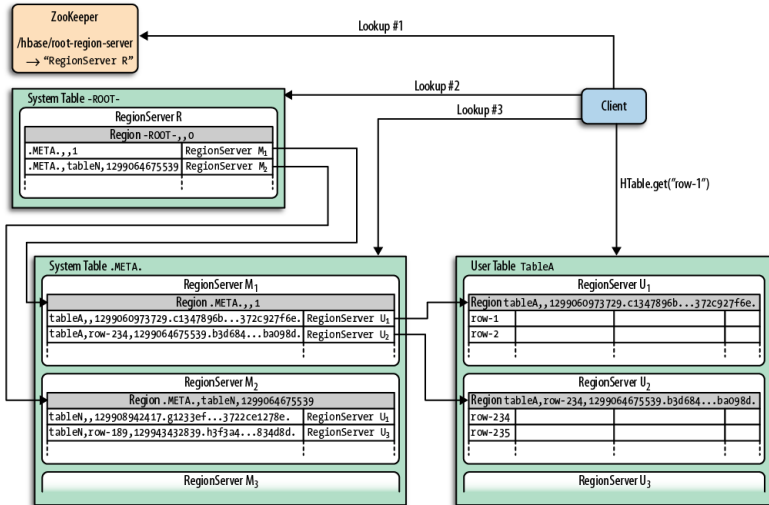
Region Lookups

- **How does a client find the region server hosting a specific row key range?**
 - ▶ HBase uses two special catalog tables, `-ROOT-` and `.META.`
 - ▶ The `-ROOT-` table is used to refer to all regions in the `.META.` table
- **Three-level B+ Tree -like operation**
 - ▶ Level 1: a node stored in ZooKeeper, containing the location (region server) of the `-ROOT-` table
 - ▶ Level 2: Lookup in the `-ROOT-` table to find a matching meta region
 - ▶ Level 3: Retrieve the table region from the `.META.` table

Region Lookups

- **Where to send requests when looking for a specific row key?**
 - ▶ This information is cached, but the first time or when the cache is stale or when there is a miss due to compaction, the following procedure applies
- **Recursive discovery process**
 - ▶ Ask the region server hosting the matching `.META.` table to retrieve the row key address
 - ▶ If the information is invalid, it backs out: asks the `-ROOT-` table where the relevant `.META.` region is
 - ▶ If this fails, ask ZooKeeper where the `-ROOT-` table is

Region Lookups



Key Design

Concepts

- **HBase has two fundamental key structures**

- ▶ Row key
- ▶ Column key

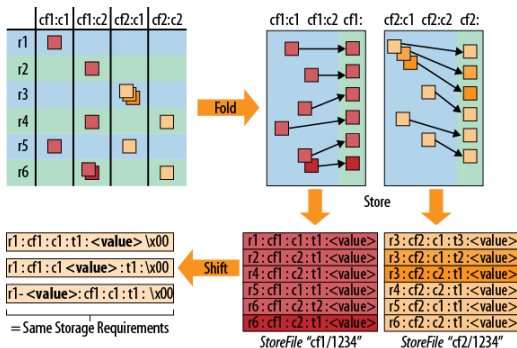
- **Both can be used to convey meaning**

- ▶ Because they store particularly meaningful data
- ▶ Because their sorting order is important

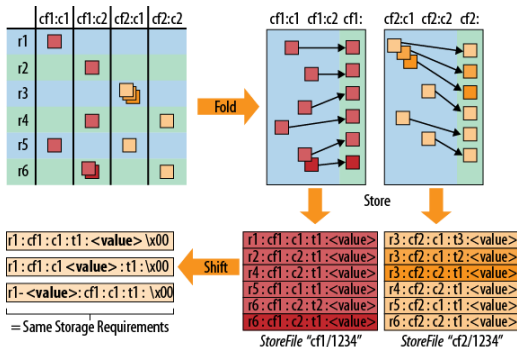
Concepts

● Logical vs. on-disk layout of a table

- ▶ Main unit of separation within a table is the *column family*
- ▶ The actual columns (as opposed to other column-oriented DB) are not used to separate data
- ▶ Although cells are stored logically in a table format, rows are stored as linear sets of the cells
- ▶ Cells contain all the vital information inside them



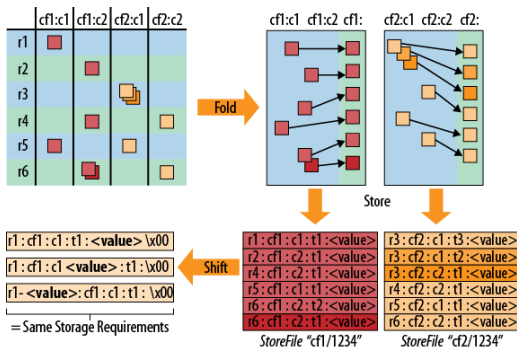
Concepts



Logical Layout (Top-Left)

- ▶ Table consists of rows and columns
 - ▶ Columns are the combination of a column family name and a column qualifier
- $\langle cf \text{ name: } qualifier \rangle$ is the **column key**
- ▶ Rows have a **row key** to address all columns of a single logical row

Concepts



● Folding the Logical Layout (Top-Right)

- ▶ The cells of each row are stored one after the other
- ▶ Each column family are stored separately
- On disk all cells of one family reside on an individual `StoreFile`
- ▶ HBase does not store unset cells
- **Row and column key is required to address every cell**

Concepts

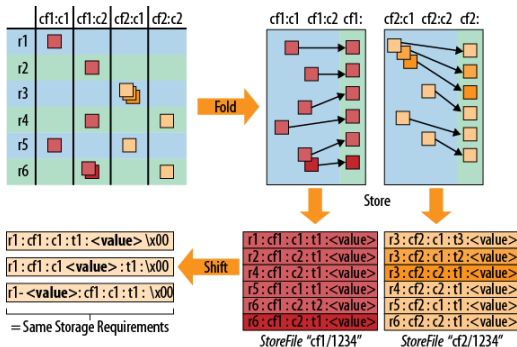
● Versioning

- ▶ Multiple versions of the same cell stored consecutively, together with the *timestamp*
- ▶ Cells are sorted in descending order of timestamp
- Newest value first

● KeyValue object

- ▶ The entire cell, with all the structural information, is a `KeyValue` object
- ▶ Contains: `row key, <column family: qualifier> → column key, timestamp and value`
- ▶ Sorted by row key first, then by column key

Concepts



Physical Layout (Lower-Right)

- ▶ Select data by row key
 - ★ This reduces the amount of data to scan for a row or a range of rows
- ▶ Select data by row key and column key
 - ★ This focuses the system on an individual storage file
- ▶ Select data by column qualifier
 - ★ Exact lookups, including filters to omit useless data

Concepts

- Summary of key lookup properties

	Key				
KeyValue	Row	Column Family	Column Qualifier	Timestamp	Value
Skip Rows	✓	X	X	X	X
Skip Store Files	✓	✓	X	✓	X
Filter Compatible	✓	✓	✓	✓	✓

← Performance

Increased Cardinality →

Tall-Narrow vs. Flat-Wide Tables

- **Tall-Narrow Tables**

- ▶ Few columns
- ▶ Many rows

- **Flat-Wide Tables**

- ▶ Many columns
- ▶ Few rows

- **Given the query granularity explained before**

- Store parts of the cell data in the row key
 - ▶ Furthermore, HBase splits at row boundaries
- It is recommended to go for Tall-Narrow Tables

Tall-Narrow vs. Flat-Wide Tables

● Example: email data - version 1

- ▶ You have all emails of a user in a single row (e.g. `userID` is the row key)
- ▶ There will be some outliers with orders of magnitude more emails than others
- A single row could outgrow the maximum file/region size and work against split facility

● Example: email data - version 2

- ▶ Each email of a user is stored in a separate row (e.g. `userID:messageID` is the row key)
- ▶ On disk this makes no difference (see the disk layout figure)
 - ★ If the `messageID` is in the column qualifier or the row key, each cell still contains a single email message
- The table can be split easily and the query granularity is more fine-grained

Partial Key Scans

- **Partial Key Scans reinforce the concept of Tall-Narrow Tables**

- ▶ From the email example: assume you have a separate row per message, across all users
- ▶ If you don't have an exact combination of user and message ID you cannot access a particular message

- **Partial Key Scan solves the problems**

- ▶ Specify a *start* and *end* key
- ▶ The start key is set to the exact `userID` only, with the end key set at `userID+1`
- This triggers the internal lexicographic comparison mechanism
 - ★ Since the table does not have an exact match, it positions the scan at:
`<userID>:<lowest-messageID>`
- ▶ The scan will then iterate over all the messages of an exact user, parse the row key and get the `messageID`

Partial Key Scans

- **Composite keys and atomicity**

- ▶ Following the email example: a single user inbox now spans many rows
- ▶ It is no longer possible to modify a single user inbox in one atomic operation

- **If this is acceptable or not, depends on the application at hand**

Time Series Data

- **Stream processing of events**

- ▶ E.g. data coming from a sensor, stock exchange, monitoring system ...
- ▶ Such data is a time series → **The row key represents the event time**
- HBase will store all rows sorted in a distinct range, namely regions with specific start and stop keys

- **Sequential monotonously increasing nature of time series data**

- ▶ All incoming data is written to the same region (and hence the same server)
- **Regions become HOT!**
- ▶ Performance of the whole cluster is bound to that of a single machine

Time Series Data

● Solution to achieve load balancing: Salting

- ▶ We want data to be spread over all region servers
- ▶ This can be done, e.g., by prefixing the row key with a non-sequential number

Salting example

```
byte prefix = (byte) (Long.hashCode(timestamp) % <number of  
region servers>);  
byte[] rowkey = Bytes.add(Bytes.toBytes(prefix),  
Bytes.toBytes(timestamp));
```

- Data access needs to be *fanned out* across many servers
- + Use multiple threads to read for I/O performance: e.g. use the Map phase of MapReduce

Time Series Data

- **Solution to achieve load balancing: Field swap/promotion**
 - ▶ Move the timestamp field of the row key or prefix it with another field
 - ★ If you already have a composite row key, simply *swap* elements
 - ★ Otherwise if you only have the timestamp, you need to *promote* another field
 - ▶ The sequential, monotonously increasing timestamp is moved to a secondary position in the row key
- You can only access data (especially time ranges) for a given swapped or promoted field (but this could be a feature)
- + You achieve load balancing

Time Series Data

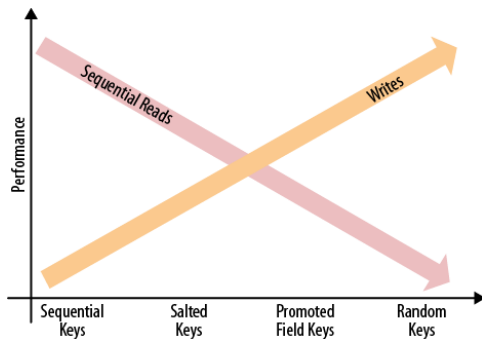
- **Solution to achieve load balancing: Randomization**

- ▶ `byte[] rowkey = MD5(timestamp)`
- ▶ This gives you a random distribution of the row key across all available region servers

- Less than ideal for range scans
- + Since you can re-hash the timestamp, this solution is good for **random access**

Time Series Data

- Summary



Cassandra

Cassandra: Overview (1)

- **Distributed key value store**

- ▶ Stores large amounts of data
- ▶ Linear scalability, high availability, no SPFL

- **Tunable consistency**

- ▶ Often eventually consistent, hence in AP
- ▶ Can guarantee strong consistency, shifting it to CP

- **Column-oriented data model**

- ▶ One key per row

Cassandra: Overview (2)

- **Combines techniques from Amazon Dynamo and HBase**

- ▶ HBase data model
 - ★ One key per row
 - ★ Columns, column families
- ▶ Dynamo-like architecture
 - ★ Partitioning, placement (using consistent hashing)
 - ★ Replication, gossip-based membership, anti-entropy

- **Some key differences**

- ▶ Many of them recently added

Data Partitioning

- **Uses consistent hashing**

- ▶ Random Partitioner
- ▶ ByteOrdered Partitioner

- **Partitioning strategy can be changed on-the-fly**

- ▶ **All** data needs to be reshuffled
- ▶ Needs to be chosen carefully

Random Partitioner

- **Hash-based identifiers for keys (data) and storage nodes**
 - ▶ Supports virtual nodes
- **Consistent hashing + load monitoring per ring**
 - ▶ Lightly loaded nodes move on the ring to alleviate heavily loaded ones
 - ▶ Make deterministic choices about load balancing, e.g., divides the hash-ring evenly w.r.t. to number of nodes
- **Node addition / suppression**
 - ▶ Requires re-balancing the cluster if no virtual nodes

ByteOrdered Partitioner

- **Supports range queries**

- ▶ Ensures row keys to be stored in sorted order
- ▶ Very different from consistent hashing

- **Key partitioning**

- ▶ There is still a ring
- ▶ Keys are ordered lexicographically along the ring by their value²

- **Precautions**

- ▶ Might be bad for load balancing
- ▶ Range scan can be obtained by using column family indexes

²The key value is different from the value associated to a key

Data Replication

- **Asynchronous replication**

- ▶ Walk down the ring and choose $N - 1$ successor nodes as replicas
- ▶ Builds a **preference list**

- **Replication strategies**

- ▶ Simple Strategy:
 - ★ Main replica = node responsible for a key
 - ★ Additional $N - 1$ replicas placed on successor nodes, clockwise in the ring, w/o rack or datacenter information
- ▶ NetworkTopology Strategy
 - ★ Allows better performance when knowledge of the datacenter layout is available
 - ★ Reads served locally
 - ★ Replica placement is independent in each datacenter
 - ★ Rack-aware placement like in HDFS

Data Replication Strategies: Implications

- **Focus on the NetworkTopology strategy**

- ▶ Requires **Snitches**³ and optionally Zookeeper
- ▶ Mechanism to discover the underlying cluster configuration

- **Potential problems**

- ▶ Unbalanced load across datacenter
- ▶ Consider datacenter-specific key rings

³We don't cover the details here: refer to the official documentation or the additional slides provided in the lecture notes.

Data Model

KeySpace

Column Family

Sorted by Key ↓

Key	Column Name	Column Name	Column Name
	Value	Value	Value

Key	Column Name	Column Name
	Value	Value

Key	Column Name	Column Name	Column Name	Column Name
	Value	Value	Value	Value

Column Family

Sorted by Key ↓

Key	Column Name	Column Name	Column Name
	Value	Value	Value

Key	Column Name	Column Name
	Value	Value

column_name
value
timestamp

Provided by
Application

Data Model: Special Columns

● Counter columns

- ▶ Store counters
- ▶ Timestamp information automatically generated (use NTP!)

● Expiring columns

- ▶ Specify a TTL value after which, data is removed
- ▶ Tombstone marker, as for HBase

● Super columns

- ▶ Additional nesting levels
- ▶ Group multiple columns on a common lookup value
 - ★ E.g.: “home address” super column, grouping “street”, “city”, “ZIP” columns
- ▶ No timestamps

Anatomy of Read/Write Operations

● Request routing

- ▶ Proxy-based mechanism (coordinator, in Cassandra terms)
- ▶ Proxy route request to **any** replica

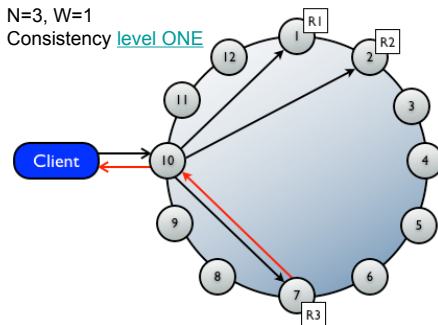
● Proxy nodes

- ▶ Handle interaction between a client and Cassandra
- ▶ First, determine replicas for a given key
- ▶ Zookeeper may be useful here

Write Requests (1)

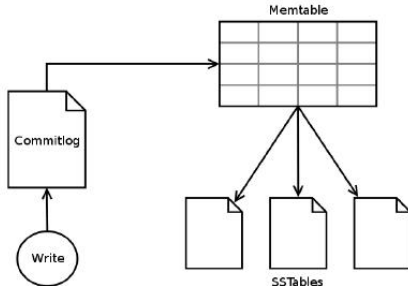
- **Proxy nodes forward write requests**

- ▶ Request routed to **all** N replicas
- ▶ This is true, regardless of consistency configuration



Write Requests (2)

- **Write request:** similar mechanism to HBase
 - ▶ Write to the commit log
 - ▶ Write to in-memory data structure (`memtable`)
 - Write is considered successful now
 - ▶ Writes are batched and periodically flushed to a persistent data structure called a sorted string table (`SSTable`)



Write Requests (3)

- **Memtables**

- ▶ Organized in sorted order by row key
- ▶ Flushed to SSTables sequentially, no random seeks

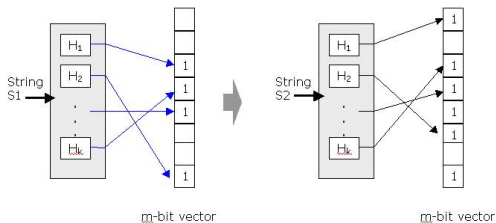
- **SSTables**

- ▶ Immutable (no rewrite after flushing)
- ▶ A single row can be stored in many SSTables
- At **read time**, rows must be combined from all SSTables (on disk or from memtables) to produce the requested data
- ▶ Use **Bloom Filters** to optimize the process

Bloom Filters (1)

• Bloom Filters in a nutshell

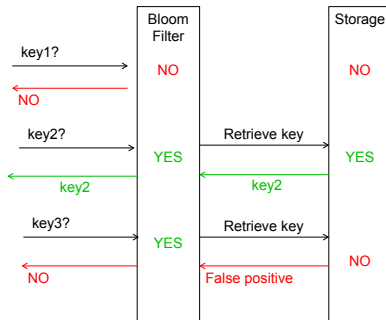
- ▶ Used to check for set membership
- ▶ k hash functions hashing into the same m -bit space



Bloom Filters (1)

• One bloom filter per SSTable

- ▶ Used in combining from row data from multiple “sources”
- ▶ Check if a requested row key exists in the SSTables, before doing any disk seeks



Read Requests (1)

- **Similar mechanism to Dynamo**

- ▶ Proxy initiates a read repair (a.k.a. writeback) if it detects inconsistent replicas
- ▶ This is done in the background, after the read has been served to the client

- **The number of replicas contacted upon a read request depend on the consistency level**

- ▶ Proxy routes the requests to the closest replica
- ▶ Proxy routes requests to all replicas and wait for a quorum

Read Requests (2)

- **When a node receives a read request**

- ▶ Row must be combined from all SSTables on that node
- ▶ Data not yet flushed to SSTables, i.e. stored in memtables, must be considered as well
- This produces the requested data

- **Key techniques to achieve high performance**

- ▶ Row-level column index
- ▶ Bloom filters

- **Cutting read latency**

- ▶ Combining data before serving it can be slow
- ▶ Read cache (in memory)
- ▶ Advanced topics: cache invalidation, consistency...

Consistency

- **Consistency in Cassandra is tunable**

- ▶ Hence is availability, as per CAP
- ▶ Read and Write consistency levels can be independent

- **Given N replicas in the preference list**

- ▶ **Write request**: all N replicas are contacted
 - ★ Ends when W respond (i.e. acknowledgment)
- ▶ **Read request**: only R replicas are contacted
 - ★ This is optimistic, may need to contact all N replicas

- **Choices of W and R define consistency level**

- ▶ Dynamo: $W + R > N$ (recall extended preference list + sloppy quorum)
- ▶ Cassandra: $W + R > N$ not mandatory

Consistency Levels: ONE

- $W = 1$
 - ▶ One replica must write to commit log and memtable
- $R = 1$
 - ▶ Returns a response from the closest replica (as determined by the snitch)
 - ▶ By default, a read repair runs in the background to make the other replicas consistent
- **This is true regardless of the replication factor N**

Consistency Levels: QUORUM

● QUORUM

- ▶ $W = \text{floor}(N/2 + 1)$: a majority
 - ★ A write is written to the commit log and memtable on a quorum of W replicas
- ▶ $R = \text{floor}(N/2 + 1)$: a majority
 - ★ Read returns the record with the most recent timestamp, once a quorum of size R has responded
 - ★ Timestamp = application timestamp

● LOCAL_QUORUM

- ▶ Restricted to a local datacenter

● EACH_QUORUM

- ▶ QUORUM invariant must be satisfied across datacenters

Consistency Levels: ALL, ANY

• ALL

- ▶ $W = N$: all replica nodes must acknowledge
- ▶ $R = N$: returns the record with the most recent timestamp across all replicas

• ANY

- ▶ Additional consistency for writes
- ▶ Allow writes to complete even if all N replicas are down
- ▶ Hinted handoff mechanism

Lightweight Transactions

- **Simple mechanism at the single key level**

- ▶ Single object transactions
- ▶ No support for multi-key transactions
- ▶ “Consistency” level: SERIAL

- **Compare and Swap (CAS) mechanism**

- ▶ Enhancements available in Cassandra 2.0
- ▶ Paxos based mechanism
- ▶ Address the problem of solving the agreement for 2 processes, that requires using locks