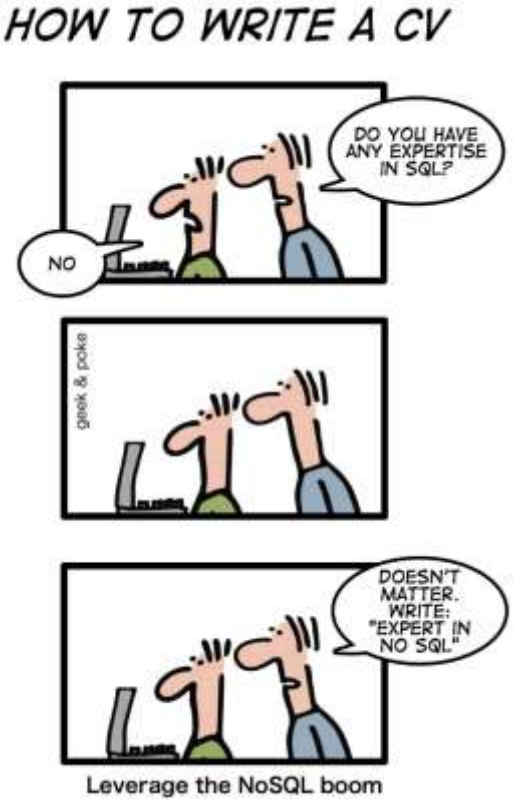
NoSQL
Apache HBase

Prof. Dr. Stephan Trahasch
Hochschule Offenburg



Outline

- Introduction
- HBase
- Table Design
- Bloom Filter
- Use Case

Why NoSQL?

- Internet Applications like Google, FB …
  → very large distributed systems
- Applications spanning over huge geographic areas
- Many concurrent users
- Different data characteristics
- Rise of Big Data

RDBMS

- Data stored in columns and tables
- Relational model with schemas
- Powerful, flexible query language
- Transactions
  ACID - Atomicity, Consistency, Isolation, Durability

Distributed RDBMS

- Partitioning
- Replication
- Caching

→ Keep replicas in sync?

→ Synchronize transactions across multiple partitions?

Limitations of RDBMS

- 2PC is slow
- Cost
- Schema
  Must design up front, painful to evolve

- What if I'm willing to give up consistency for scalability?
- What if I'm willing to give up the relational model for something more flexible?
- What if I just want a cheaper solution?

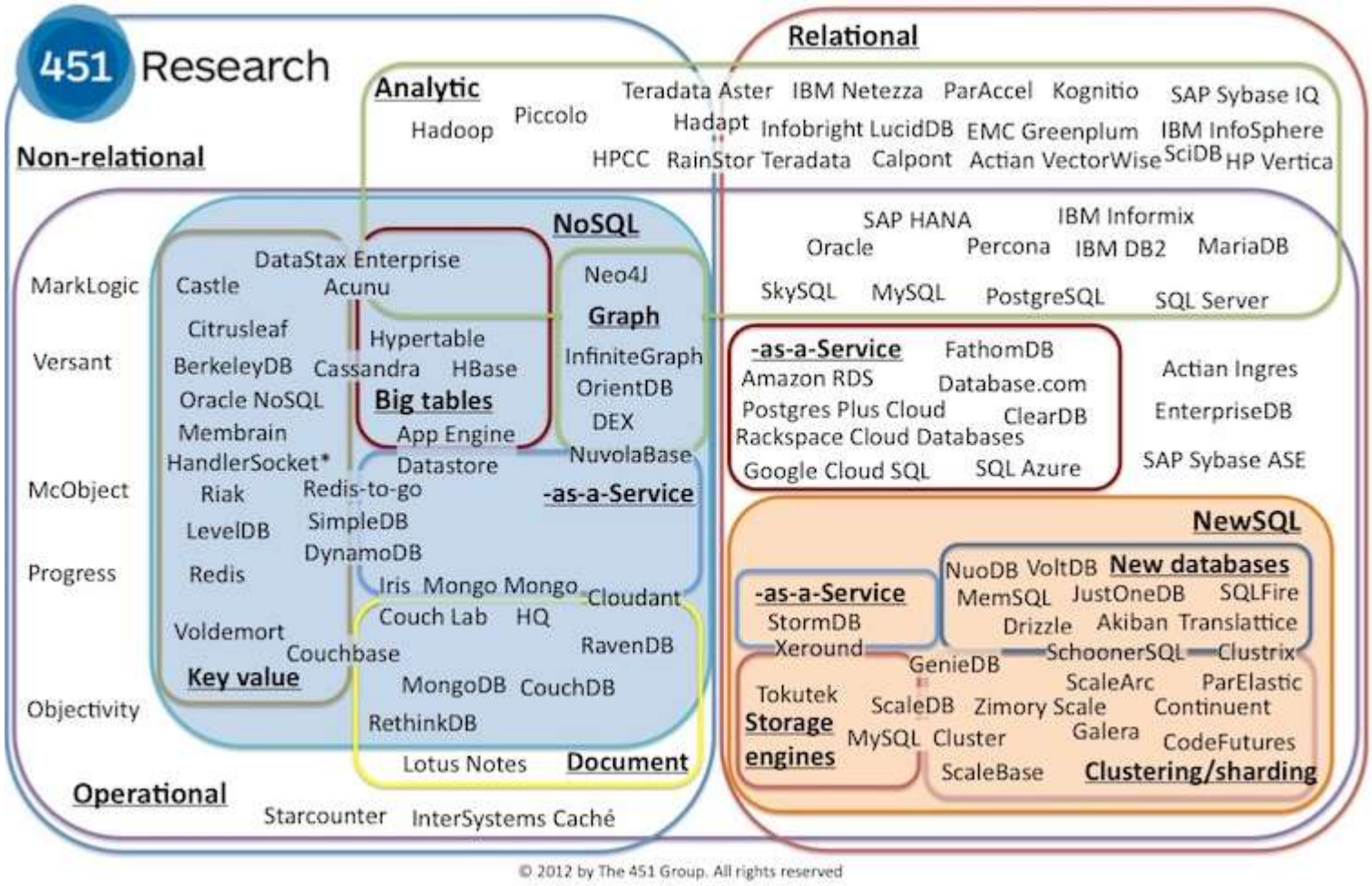
NoSQL System Characteristics

- Ability to scale horizontally
- Distribution and replication of data over many servers
- Simple interfaces, not necessary SQL
- Weaker concurrency models than ACID
- Utilization of distributed indexes and memory
- Flexible schemata

… and often Open Source

Source: R. Cattell, “Scalable SQL and NoSql Data Stores.” SIGMOD Record, 39(4), 27-Dec-2010.

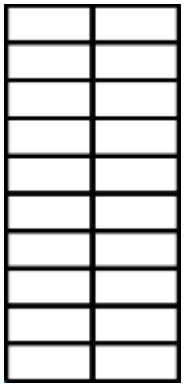
Database landscape



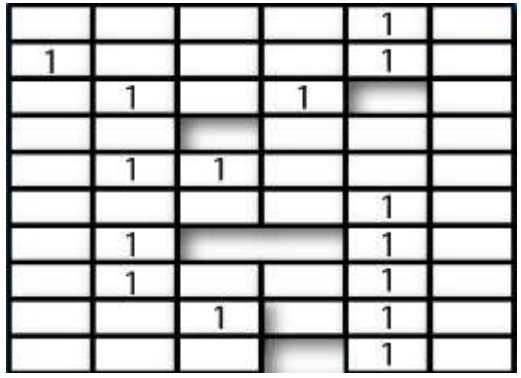
https://blogs.the451group.com/information_management/files/2012/11/DBlandscape.jpg

(Major) Types of NoSQL databases

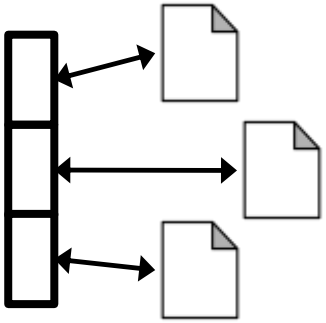
Key Value



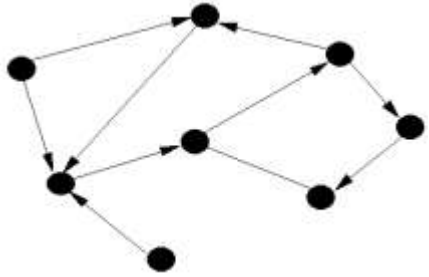
Column Family



Document



Graph



NoSQL Systems

CAP Theorem (also known as Brewer's theorem)

CAP theorem states that in a distributed database you can only have two of the following properties:

1. **Consistency** equivalent to having a single up-to-date copy of the data
   All requests at the same time retrieve the same value.
2. High **Availability** of that data
   The retrieval of data is always possible as long as at least one server is running.
3. Tolerance to Network **Partitions**
   The system will function even if the communication is broken.

CAP theorem stated at the Symposium on Principles of Distributed Computing (PODC) by Eric Brewer in 2000

Formal proof by Seth Gilbert, Nancy Lynch in 2002

CAP Theorem

Assume a server (single node, no cluster) has performance problems.

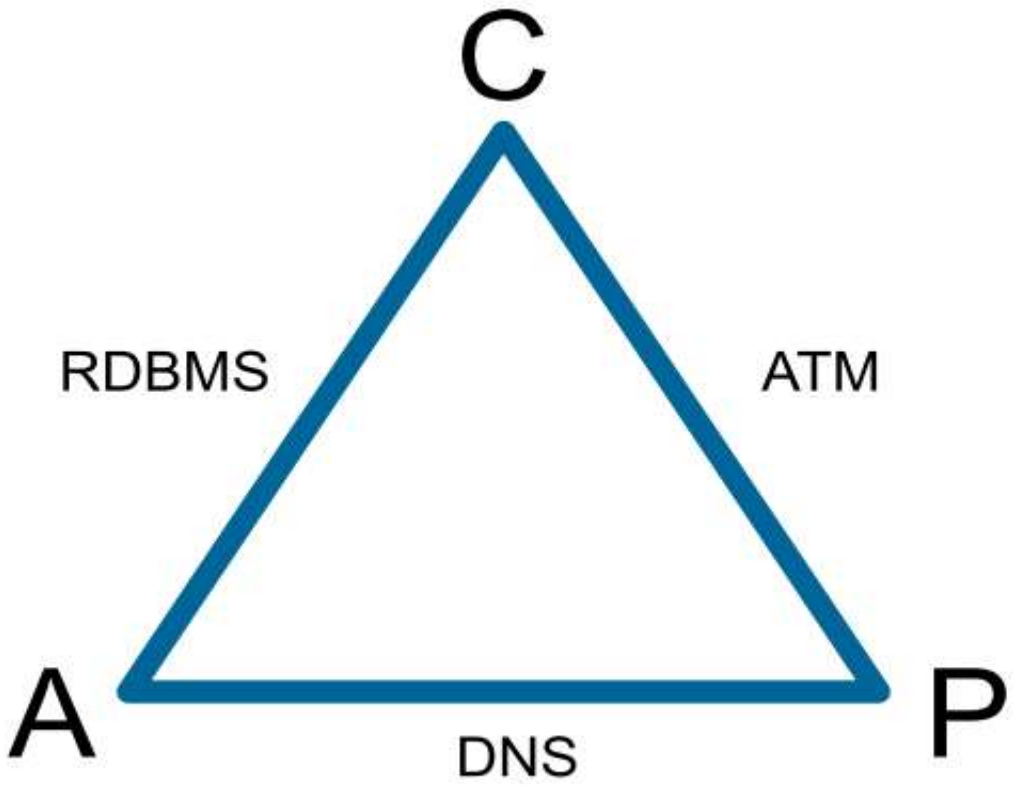
→Solution: Add another node to increase performance. Now we have a distributed system.

A new problem occurs in our two-node cluster: When data is written to both nodes the data is not consistent if it‘s not synchronized (the system is still available and partition tolerant).

→Solution: Each node propagates updates to other node. That requires that both nodes are online all the time.

If one node is down, the other can‘t function anymore and the system is not available anymore (but still consistent and partition tolerant).

→Solution: The nodes offline will perform the updates (stored in a queue) when they are online again. Not partition tolerant (but always consistent and available).


C
RDBMS
ATM
A
P
DNS
Social Media Sites (were weak consistency is okay)

CAP Theorem Revisited

"The '2 of 3' formulation was always misleading because it tended to oversimplify the tensions among properties. Now such nuances matter. CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare.

Although designers still need to choose between consistency and availability when partitions are present, there is an incredible range of flexibility for handling partitions and recovering from them. The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application. Such an approach incorporates plans for operation during a partition and for recovery afterward, thus helping designers think about CAP beyond its historically perceived limitations."

Source: Eric Brewer, "CAP twelve years later: How the 'rules' have changed", IEEE Explore, Volume 45, Issue 2 (2012), pg. 23-29.
Additional reading:
Daniel Abadi (February 2012), "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story", IEEE Computer Society Press 45(2):27-42

CAP Theorem Revisited

"Of the CAP theorem's Consistency, Availability, and Partition Tolerance, Partition Tolerance is mandatory in distributed systems. You cannot not choose it."

Coda Hale, Yammer Software Engineer

http://codahale.com/you-cant-sacrifice-partition-tolerance/

"An important observation is that in larger distributed-scale systems, network partitions are a given; therefore, consistency and availability cannot be achieved at the same time."

Werner Vogels, Amazon CTO

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

"So in reality, there are only two types of systems: CP/CA and AP. I.e., if there is a partition, does the system give up availability or consistency?"

Daneil Abadi, Co-founder of Hadapt, Associate Professor at Yale University

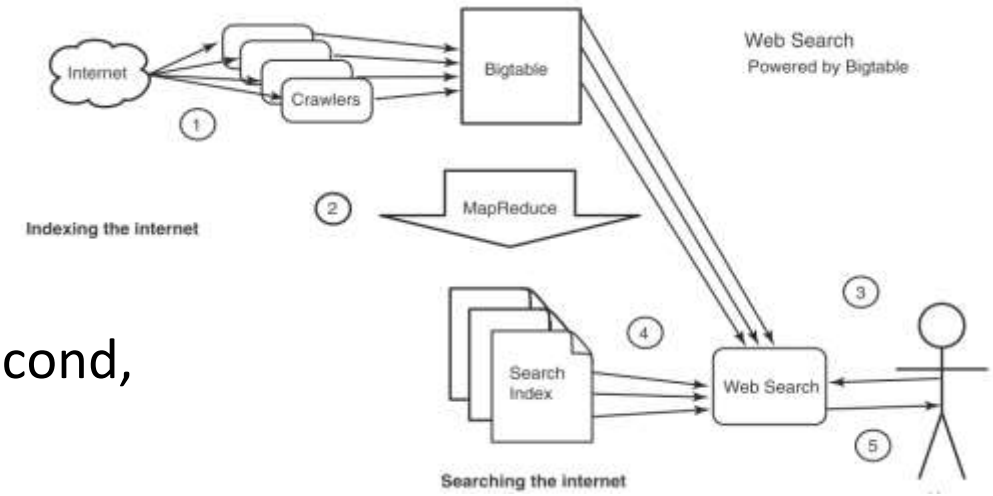
http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

Outline

- NoSQL
- HBase
- Table Design
- Bloom Filter
- Use Case

Google Bigtable

- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance
- Extremely popular at Google (as of 2008)
  - Web indexing, personalized search, Google Earth, Google Analytics, Google Finance, …



„Bigtable: A Distributed Storage System for Structured Data“
Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

Apache HBase

- Non-relational, distributed, column-based database
- Table consists of rows, column families and columns
- Stored as key value pairs
  - Key, CF, CQ and value
- Distribution to so-called regions
  - Partitions to which data is assigned
- Storage on distributed file system HDFS or other fs

Row Key | Column Family

Row Key	BaseData		TestData			
ProcessID	Machine	Product	Temperature	Voltage	Accuracy	Size
1	M1	Thermometer	23.1	3.3	0.1	
2	M1	LightSensor		5.1		0.4
3	M2	Microchip		1.7		0.05
4	M3	Service				

Column Qualifiers

Cell

Data Model

- A table in HBase is a sparse, distributed, persistent multidimensional **sorted map**
- Map indexed by a row key, column key, and a timestamp (Table, RowKey, Family, Column, Timestamp) → Value
- Supports lookups, inserts, deletes
  - Single row transactions only

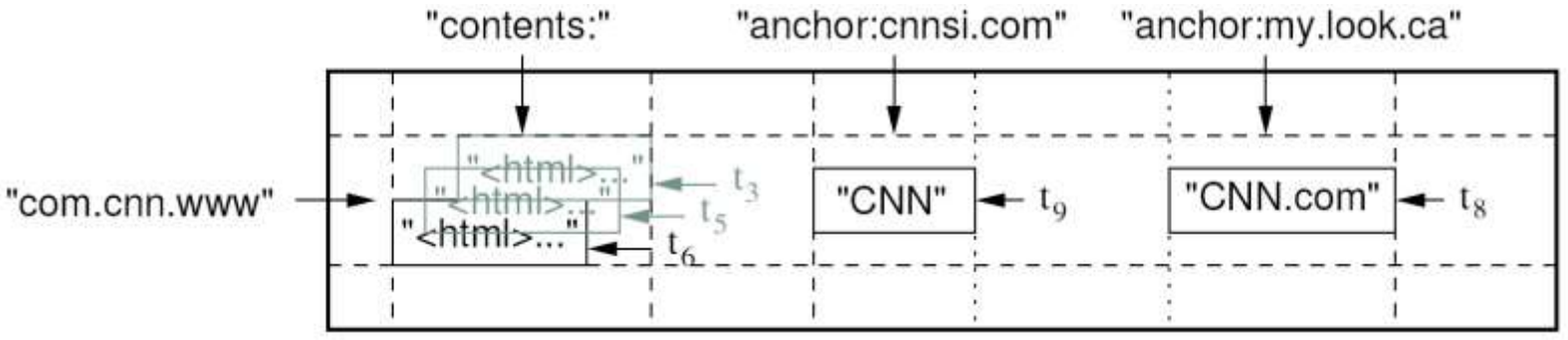


Image Source: Chang et al., OSDI 2006

Data Model

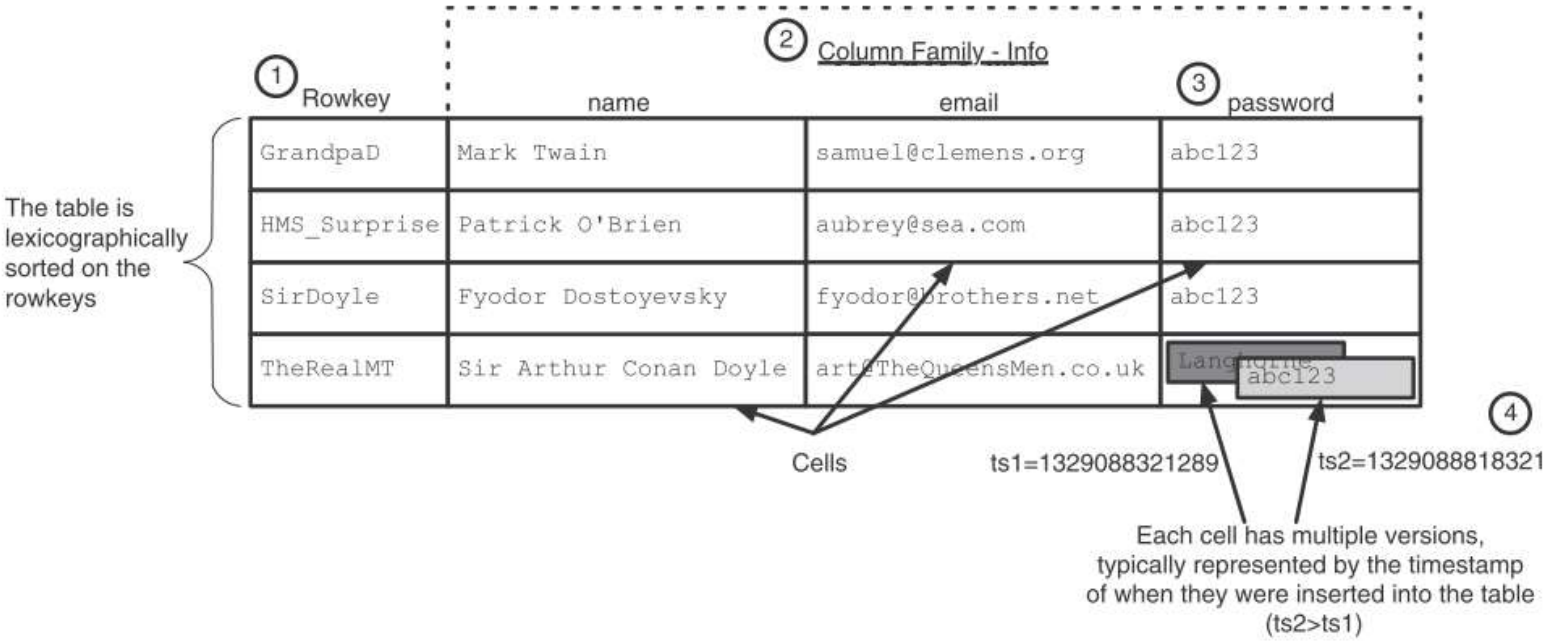
Row Key	Time Stamp	ColumnFamily contents:…	ColumnFamily anchor:…	ColumnFamily people:…
"com.cnn.www"	t9		anchor:cnnsi.com = "CNN"	
"com.cnn.www"	t8	contents:html = "<html>…"	anchor:my.look.ca = "CNN.com"	
"com.cnn.www"	t6	contents:html = "<html>…"		
"com.cnn.www"	t5	contents:html = "<html>…"		
"com.cnn.www"	t3	contents:txt = „This is a text…"		

Rows and Columns

- Rows are composed of columns, which are grouped into column families
- Column families group semantically related values
  - Column key = `family:qualifier`
  - Each column family is stored as one file (HFile) in HDFS
  - One column family can have millions of columns
  - Hint: not more than 3 families
- Rows maintained in sorted lexicographic order
  - Applications can exploit this property for efficient row scans
  - Row ranges dynamically partitioned into tablets

At the end of the day, it's all key-value pairs!

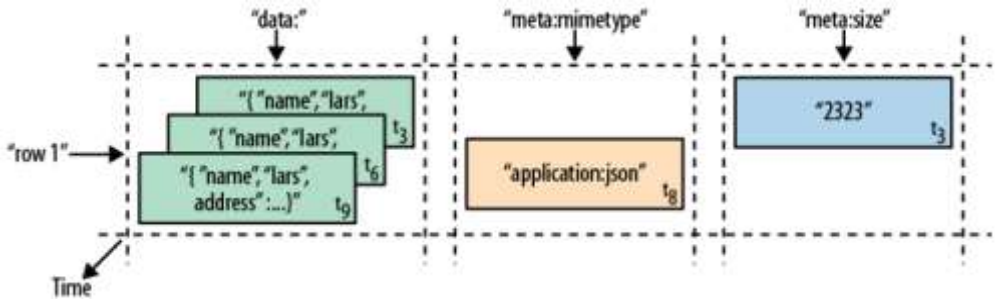
Columns and Column Families



- versioning of data in a cell
  - Current timestamp
  - Delete content older than „X“ minutes, days etc.
- Search
  - „K“-most current versions
  - or search für content of one family:type

Timestamp

- Versioning: used to store different versions of data in a cell
  - 64-bits integer (UNIX timestamp)
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Lookup options
  - “Return most recent K values”
  - “Return all values in timestamp range (or all values)”
- Garbage collection:
  - “Only retain most recent K values in a cell”
  - “Keep values until they are older than K seconds”



Column Families

Columns are managed in isolation. Different applications access only a subset of the columns.

Advantage of vertical partitioning: unbound attributes of an entity do not need to be explicitly saved

Konzeptionell

Row Key	Time Stamp	Column "contents:"	Column "anchor:"		Column "mime:"
"com.cnn.www"	t9	"<html>abc..."	"anchor: cnnsi.com"	"CNN"	
	t8	"<html>def..."	"anchor: my.look.ca"	"CNN.com"	
	t6	"<html>ghi..."			"text/html"

Intern

Row Key	Time Stamp	Column "contents:"
"com.cnn.www"	t9	"<html>abc..."
	t8	"<html>def..."
	t6	"<html>ghi..."

Row Key	Time Stamp	Column "anchor:"
"com.cnn.www"	t9	"anchor: cnnsi.com"
	t8	"anchor: my.look.ca"

...

Partitions are then saved individually, because multiple Column Families are not accessed at the same time (definition!) in one request.

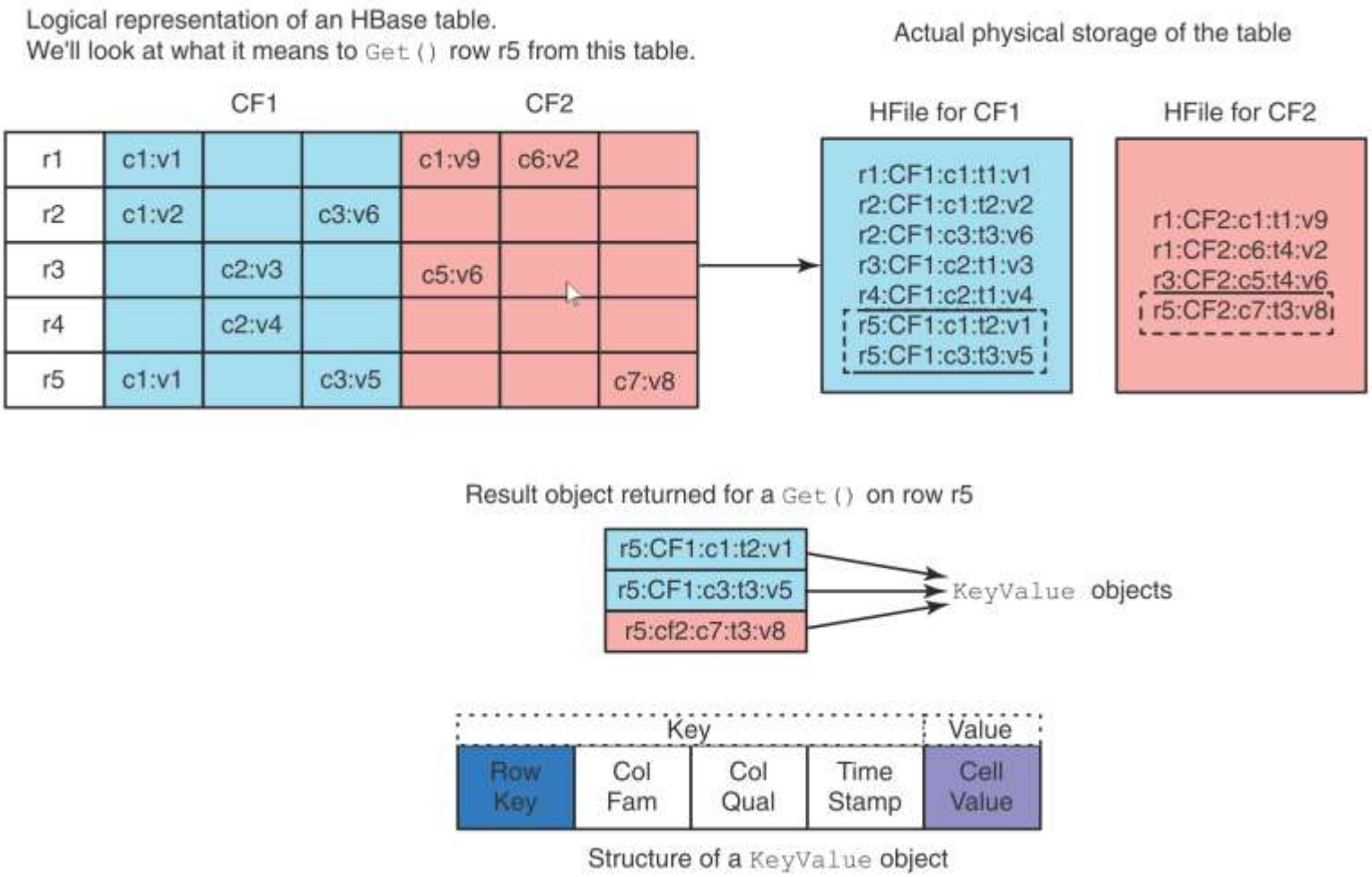
Logical Data Model
A sparse, multi-dimensional, sorted map

Table A

rowkey	column family	column qualifier	timestamp	value
a	cf1	"bar"	1368394583	7
			1368394261	"hello"
		"foo"	1368394583	22
			1368394925	13.6
			1368393847	"world"
	cf2	"2011-07-04"	1368396302	"fourth of July"
		1.0001	1368387684	"almost the loneliest number"
b	cf2	"thumb"	1368387247	[3.6 kb png data]

- Rows are sorted by rowkey.
- Within a row, values are located by column family and qualifier.
- Values also carry a timestamp; there can me multiple versions of a value.
- Within a column family, data is schemaless. Qualifiers and values are treated as arbitrary bytes.

Logical to physical translation of an HBase table



Dimiduk, N., & Khurana, A. (2013). HBase in action. Shelter Island, NY: Manning.

Writing Data

```
HTablePool pool = new HTablePool();
HTableInterface usersTable = pool.getTable("users");
... // work with the table
Put p = new Put(Bytes.toBytes(“AverageJoe"));
p.add(Bytes.toBytes("info"),
    Bytes.toBytes("name"),
    Bytes.toBytes(“Joe Average"));
p.add(Bytes.toBytes("info"),
    Bytes.toBytes("email"),
    Bytes.toBytes(“joe@average.org"));
p.add(Bytes.toBytes("info"),
    Bytes.toBytes("password"),
    Bytes.toBytes(“1711!"));
usersTable.put(p);
usersTable.close();
```

- Cell is identified by [rowkey, column family, column qualifier]
- Cell in the example has the coordinates [JoeAverage, info, …]

Reading Data

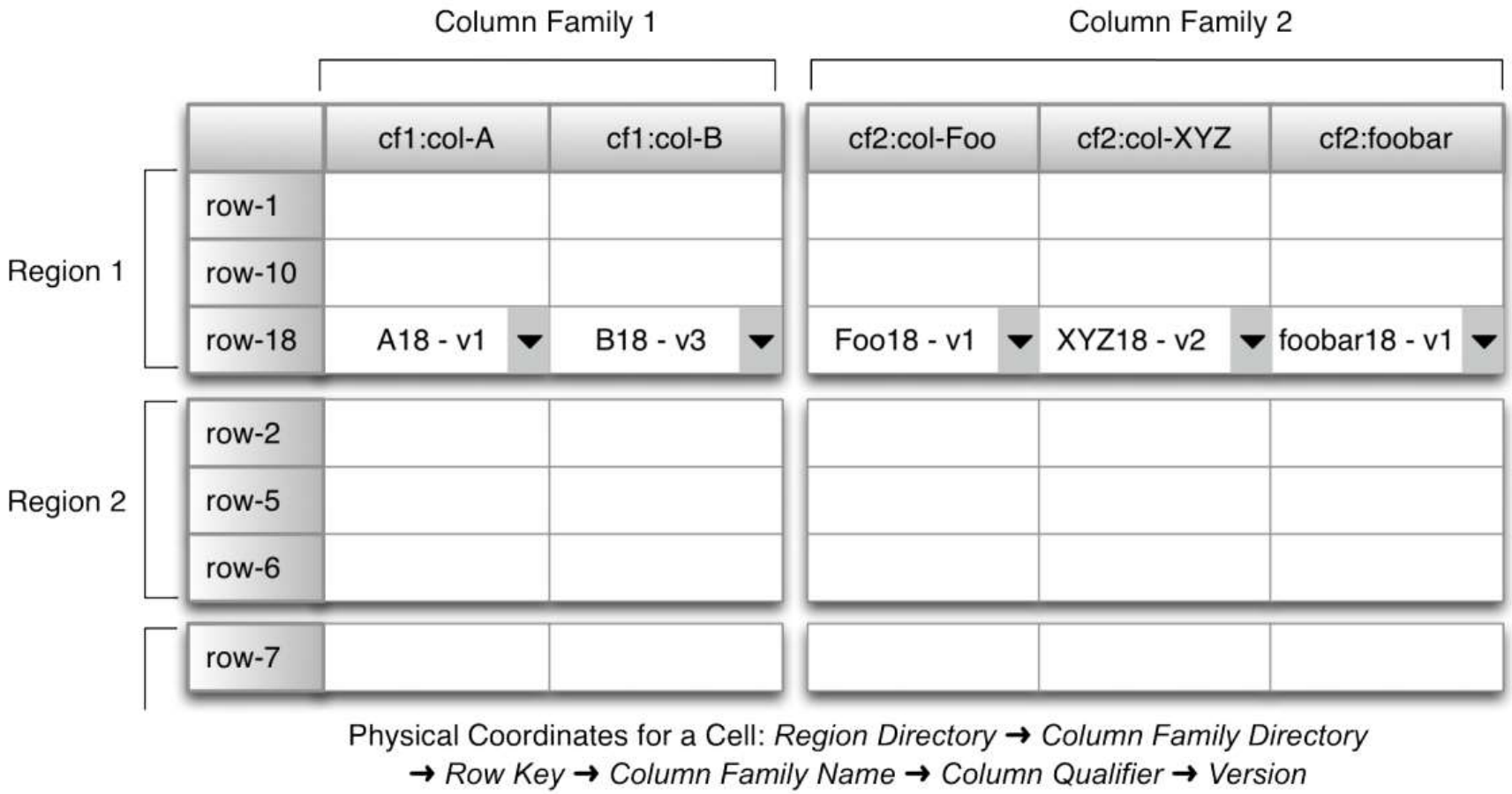
```
Get g = new Get(Bytes.toBytes("AverageJoe"));
g.addColumn(    //addFamily()
    Bytes.toBytes("info"),
    Bytes.toBytes("password"));
Result r = usersTable.get(g);

//Retrieve the specific value
Get g = new Get(Bytes.toBytes("AverageJoe"));
g.addFamily(Bytes.toBytes("info"));
byte[] b = r.getValue(
    Bytes.toBytes("info"),
    Bytes.toBytes("email"));
String email = Bytes.toString(b); // joe@average.org
```

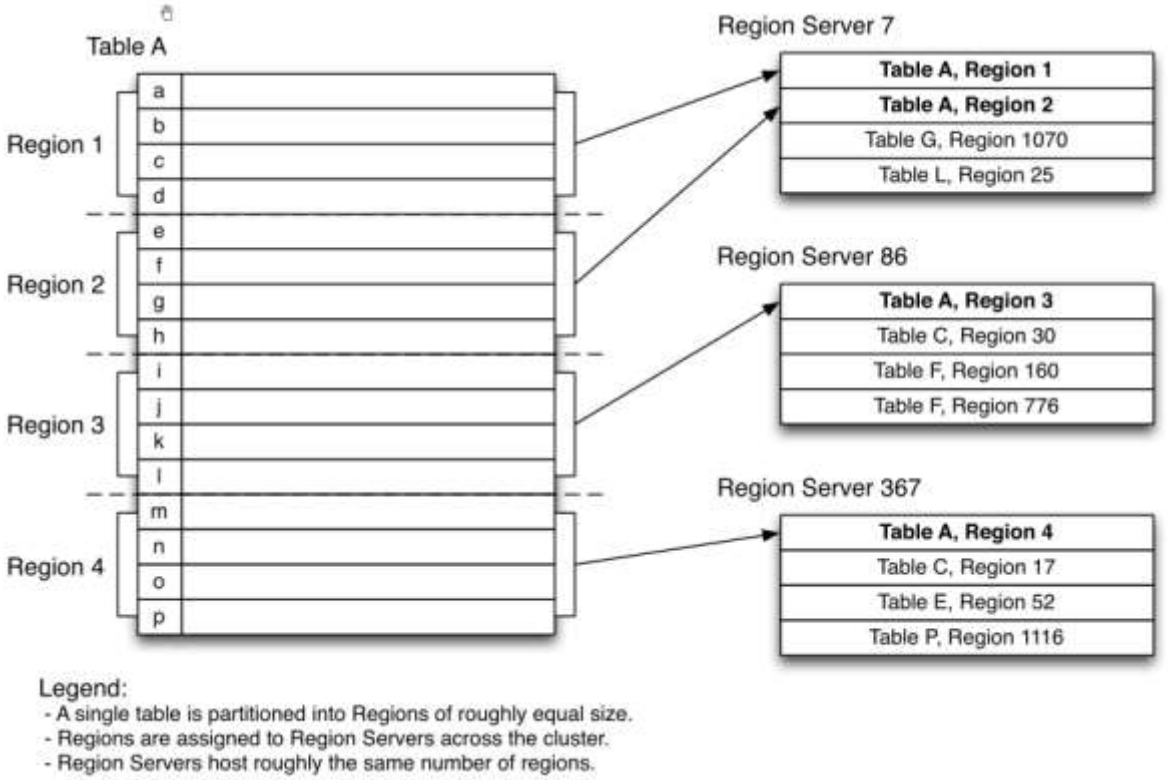
HBase Regions

For scalability, tables are split into regions by key.

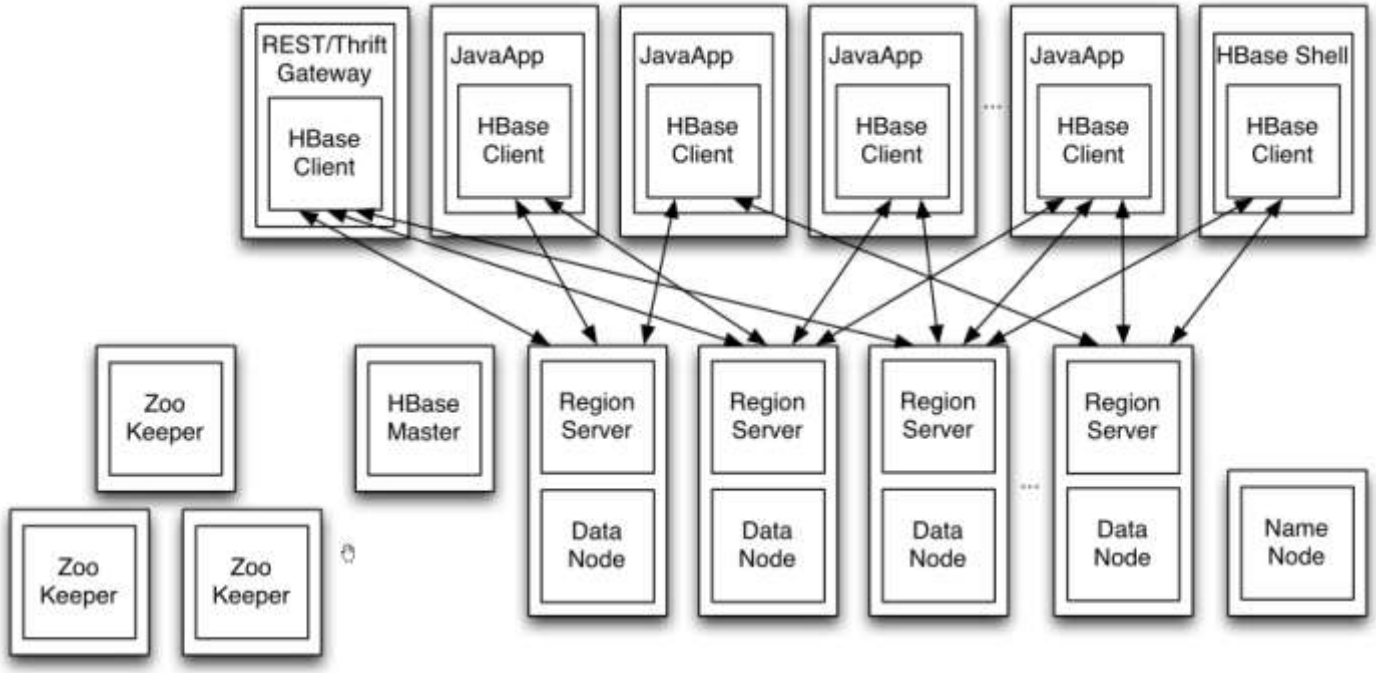
Each region is assigned to a region server.



Logical Architecture



Physical Architecture



Legend:
- An HBase RegionServer is collocated with an HDFS DataNode.
- HBase clients communicate directly with Region Servers for sending and receiving data.
- HMaster manages Region assignment and handles DDL operations.
- Online configuration state is maintained in ZooKeeper.
- HMaster and ZooKeeper are NOT involved in data path.

System Overview

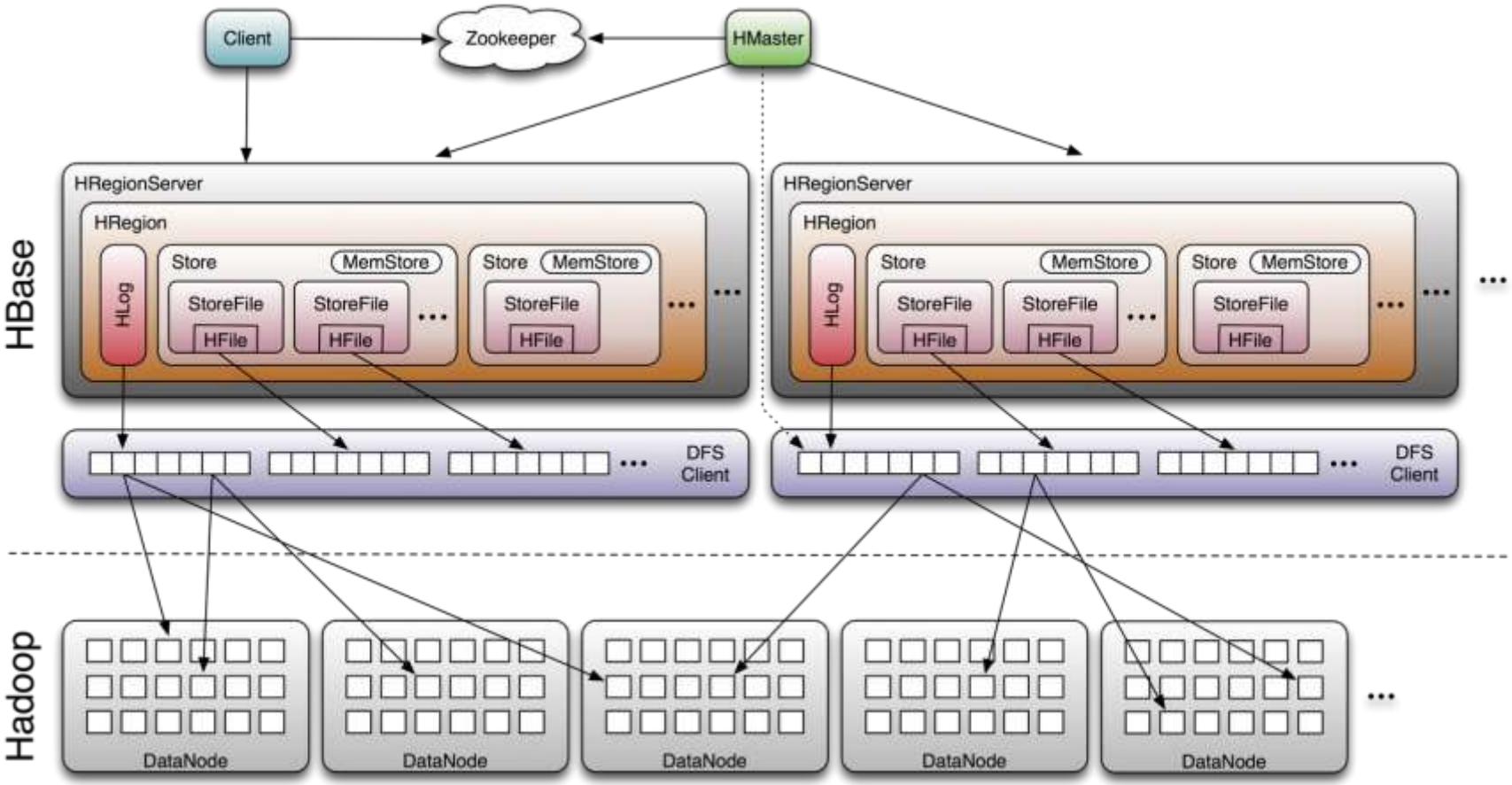
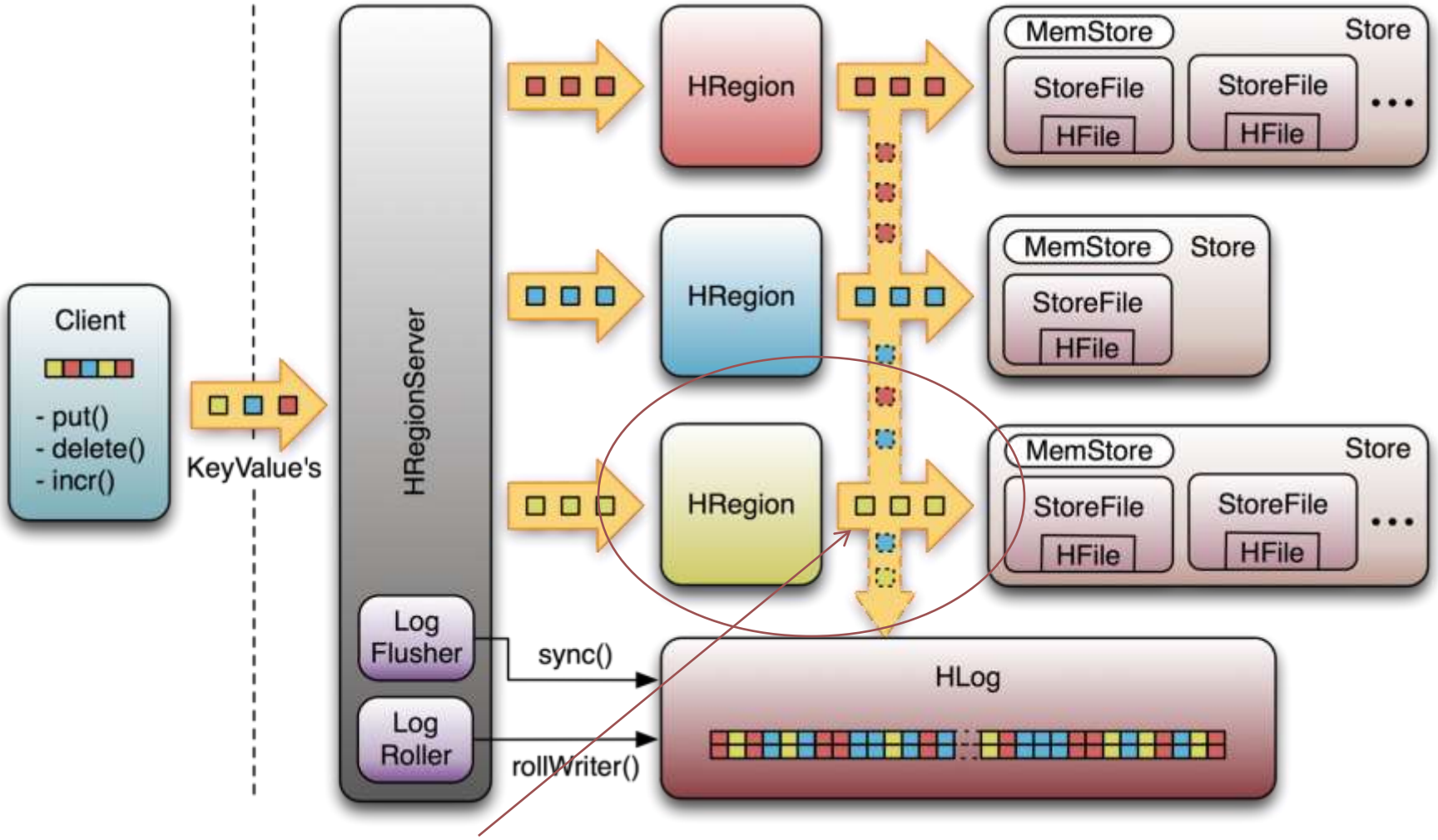


Image Source: http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

Architecture Components and Responsibilities

- **Master**
  - Monitor RegionServer
  - Runs LoadBalancer to transfer regions between servers
  - Check and clean the meta table
  - Typically runs on HDFS NameNode
- **RegionServer**
  - Hosts a subsequent span of keys (Region) for tables
  - Executes Client Request Filters
  - Runs periodic compaction
  - Typically runs on HDFS DataNode
  - Memstore: Accumulates all writes
  - If filled, data is flushed to new store files
  - Multiple smaller files can be compacted into fewer
  - After flushes/compaction the region may be split
- **Client**
  - Identify location of HBase:meta from ZooKeeper
  - Query HBase:meta for identifying the RegionServers
  - May use Client Request Filters

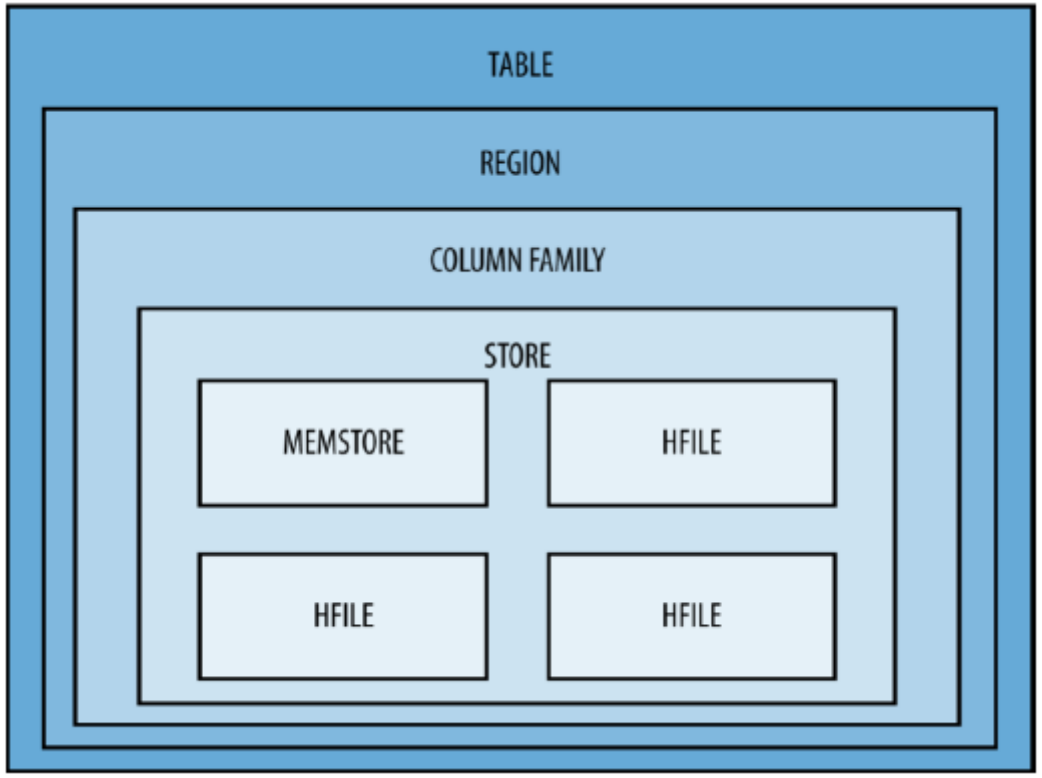
Strong Consistency: HBase Write-Ahead Log



Write to HLog before writing to MemStore
Thus can recover from failure

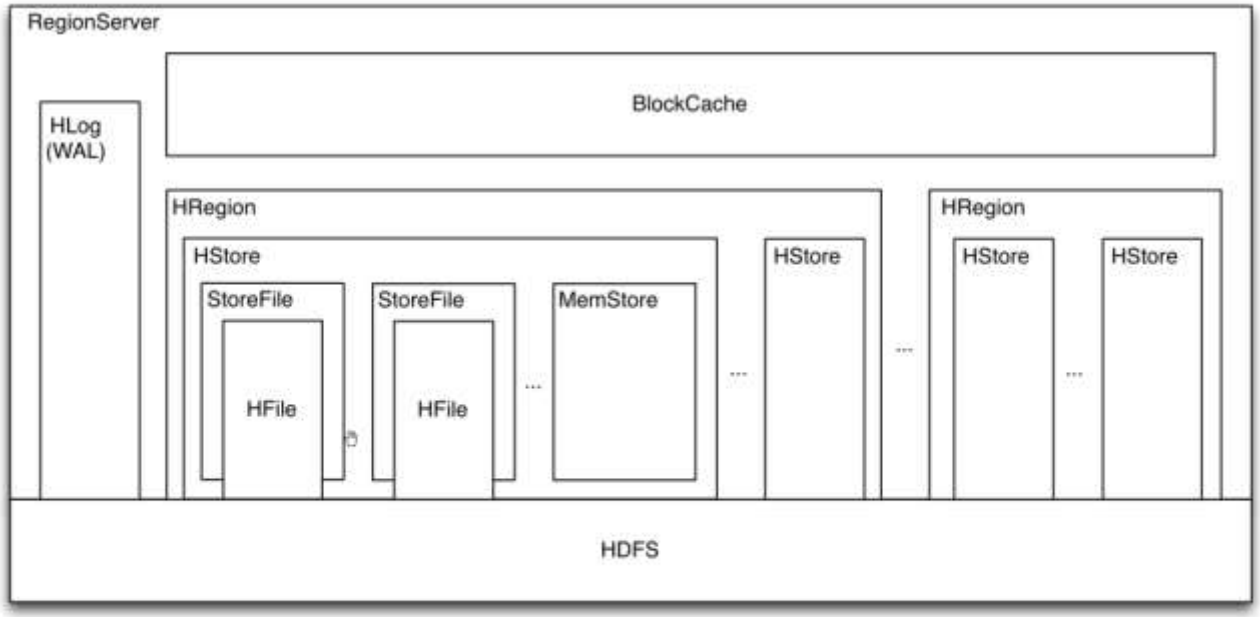
Source: http://www.larsgeorge.com/2010/01/hbase-architecture-101-write-ahead-log.html

HBase Storage



O'Dell, J.-M. S. K. (2016). *Architecting HBase Applications*. S.l.: O'Reilly Media, Inc.

HBase Storage at a Region Server



Legend:
- A RegionServer contains a single WAL, single BlockCache, and multiple Regions.
- A Region contains multiple Stores, one for each Column Family.
- A Store consists of multiple StoreFiles and a MemStore.
- A StoreFile corresponds to a single HFile.
- HFiles and WAL are persisted on HDFS.

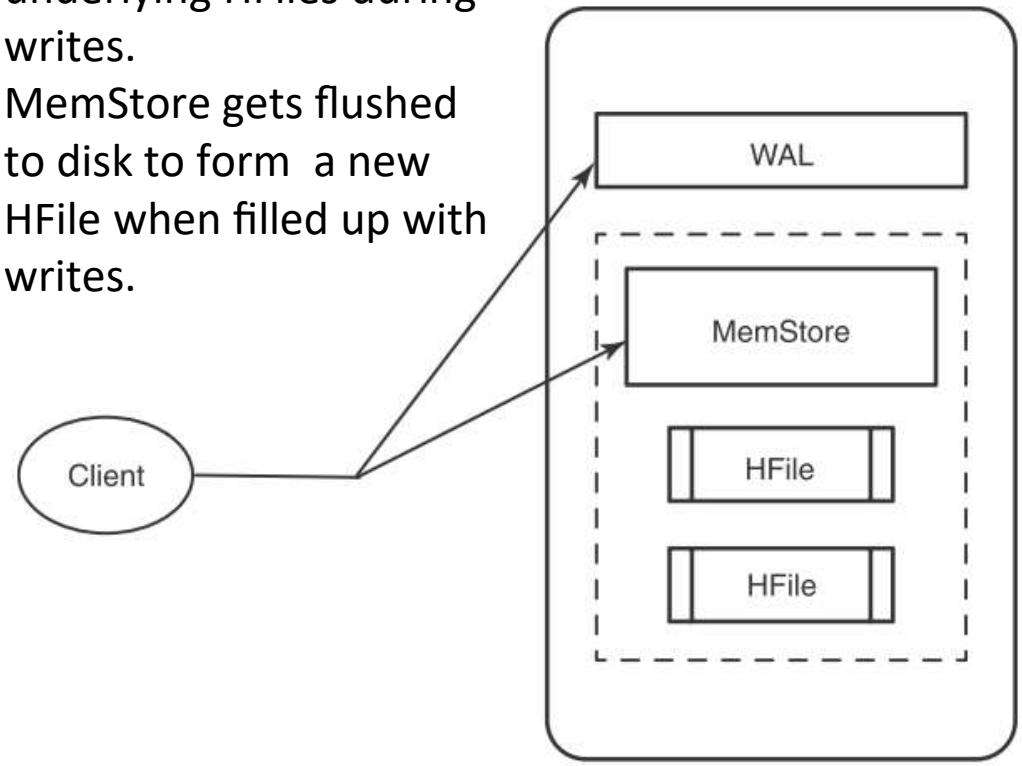
Theree types of files
- Write Ahead Logs
- Data files
  (a.k.a. store files, hfiles)
- References / symbolic or logical links
  (0 length files)

Every file is 3-way replicated

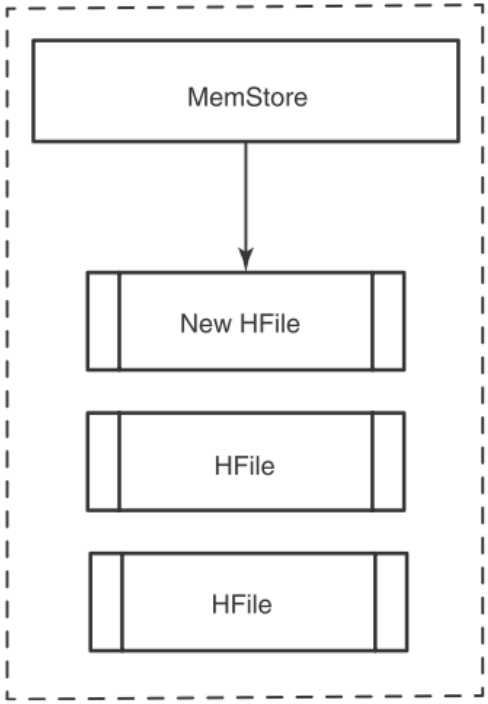
The write cache memory area for a given RegionServer is shared by all the column families configured for all the regions hosted by the given host

Write Path

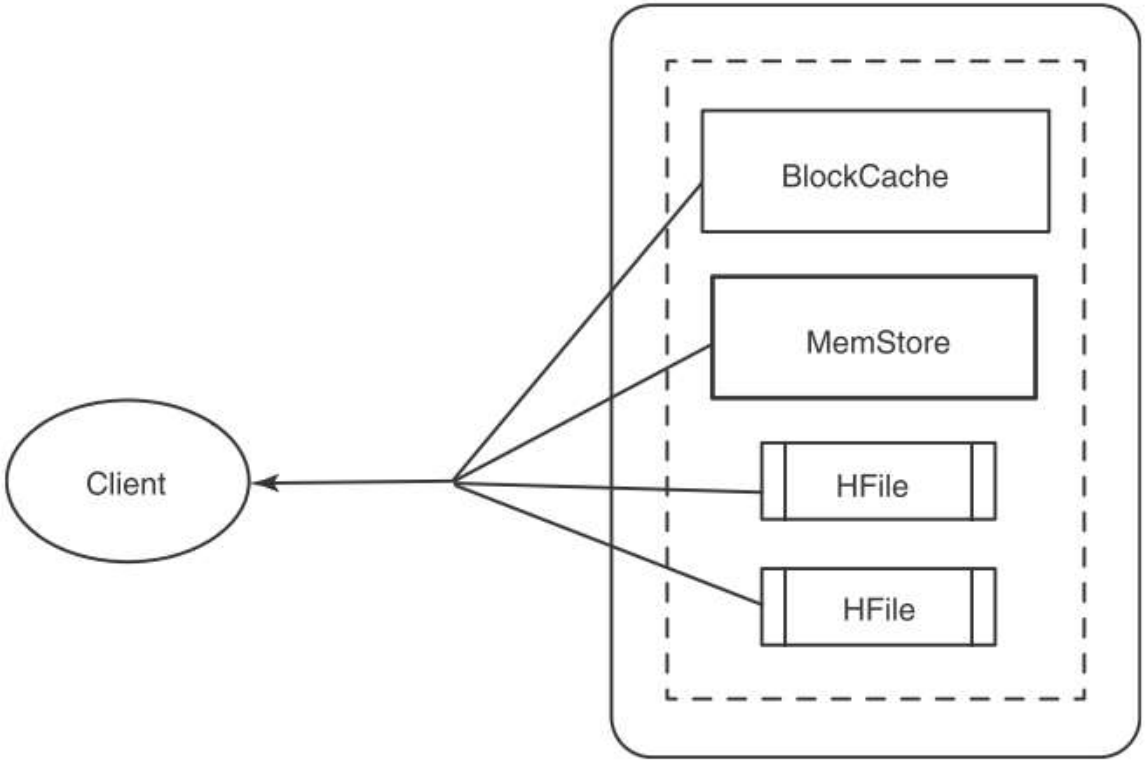
Write goes to the write-ahead log and MemStore. Clients don't interact directly with the underlying HFiles during writes.
MemStore gets flushed to disk to form a new HFile when filled up with writes.



MemStore gets flushed to disk to form a new HFile when filled up with writes.



Read Path



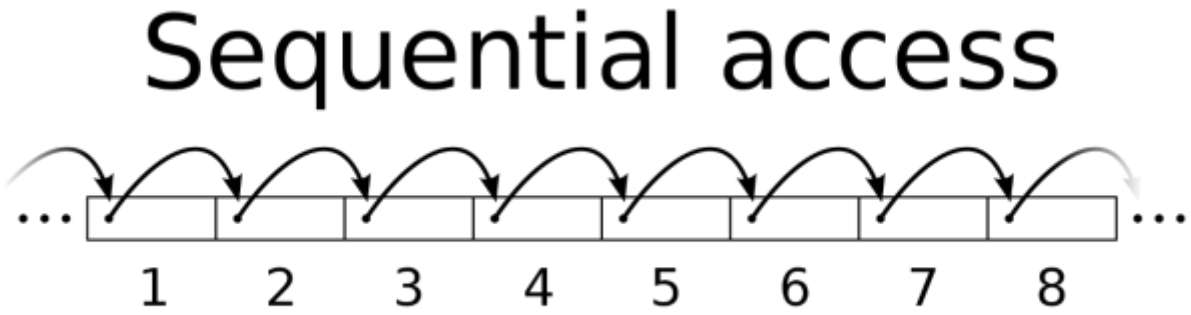
Stores and HFiles

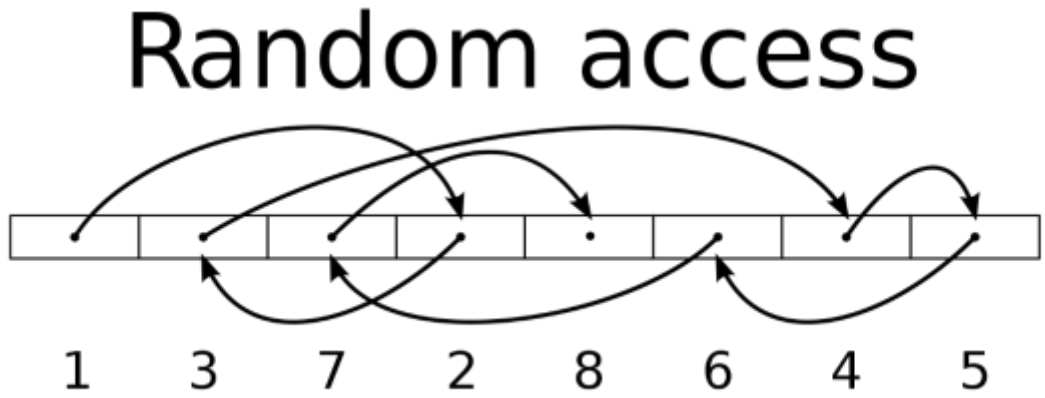
- Stores
  One store per column family. A store object regroups one memstore and zero or more store files (called HFiles). This is the entity that will store all the information written into the table and will also be used when data needs to be read from the table.
- HFiles
  HFiles are created when the memstores are full and must be flushed to disk. Hfiles are eventually compacted together over time into bigger files. They are the HBase file format used to store table data. HFiles are composed of different types of blocks (e.g., index blocks and data blocks). HFiles are stored in HDFS, so they benefit from Hadoop persistence and replication.

HFiles are composed of blocks.

- Those blocks should not be confused with HDFS blocks. One HDFS block might contain multiple HFile blocks.
- HFile blocks are usually between 8 KB and 1 MB, but the default size is 64 KB.
- However, if compression is configured for a given table, HBase will still generate 64 KB blocks but will then compress them. The size of the compressed block on the disk might vary based on the data and the compression format.
- Larger blocks will create a smaller number of index values and are good for sequential table access.
- While smaller blocks will create more index values and are better for random read accesses.

Random, realtime read/write access with HBase





Hadoop File Formats

SequenceFile file format: append key/value pairs
If you want to lookup a specified key, you've to read through the file until you find your key.

MapFile is an extension of the SequenceFile. MapFile is a directory that contains two SequenceFiles:
1. data file "/data"
2. index file "/index".

MapFile allows you to append sorted key/value pairs and every N keys it stores the key and the offset in the index.

- This allows for quite a fast lookup, since instead of scanning all the records you scan the index which has less entries.
- Once you've found your block, you can then jump into the real data file.

MapFile fast lookup.
But there are still two problems:

1. How can I delete or replace a key/value?
2. When my input is not sorted, I can not use MapFile.

HBase Key consists of:
row key, column family, column qualifier, timestamp and a type.

Row Length *short*	**Row Key** *byte[]*	Family Length *byte*	Column **Family** byte[]	Column **Qualifier** *byte[]*	**Timestamp** *long*	**Key Type** *byte*

1. deleting key/value pairs: use the “type” field to mark key as deleted (tombstone markers).
   replacing key/value pairs: picking the later timestamp
2. non-ordered key: keep the last added key-values in memory. When you’ve reached a threshold, HBase flush it to a MapFile. In this way, you end up adding sorted key/values to a MapFile.

In-Memory → MapFile

- When you add a value with table.put(), your key/value is added to the MemStore.
- When the per-memstore threshold is reached or the RegionServer is using too much memory for memstores, data is flushed on disk as a new MapFile.

The result of each flush is a new MapFile, and this means that to find a key you have to search in more than one file.
This takes more resources and is potentially slower.

Compaction

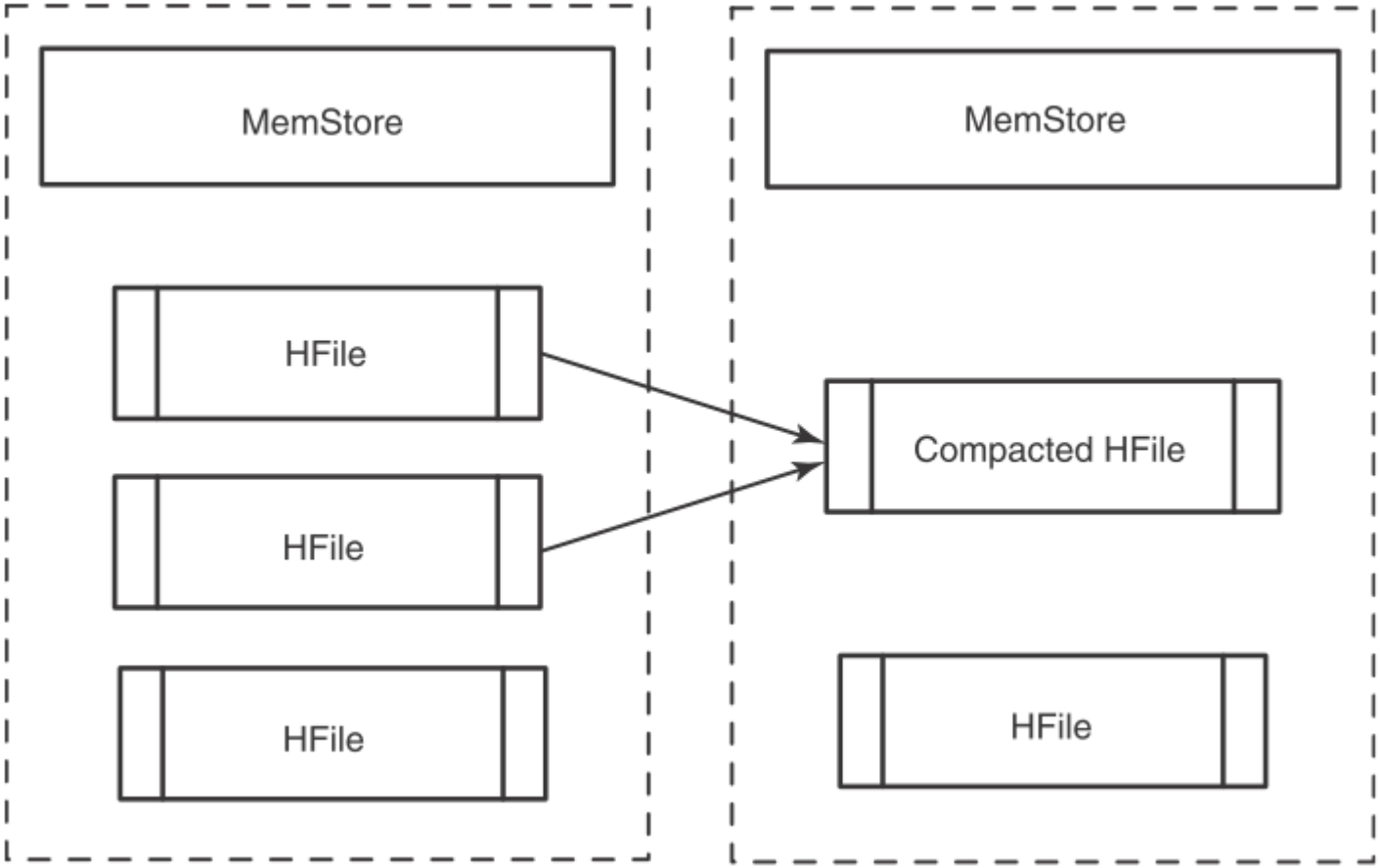
Each time a get or a scan is issued, HBase scan through each file to find the result.

To avoid jumping around too many files, there's a thread that will detect when you've reached a certain number.

It tries to merge them together in a process called compaction, which basically creates a new large file as a result of file merge.

- "minor compaction":
  merges two or more small files into one
- "major compaction":
  picks up all files in the region, merges them and performs some cleanup. In major compaction, deleted key/values are removed. New file doesn't contain the tombstone markers and all the duplicate key/values are removed.

Compaction

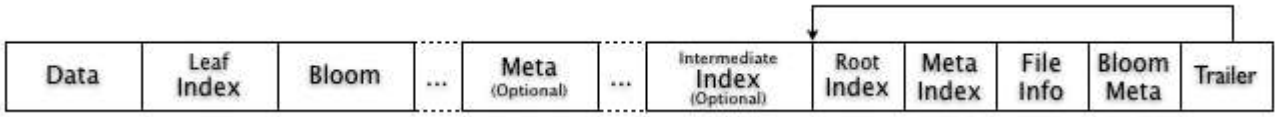

MemStore
HFile
HFile
HFile
MemStore
Compacted HFile
HFile

Following main block types can be encountered in an HFile

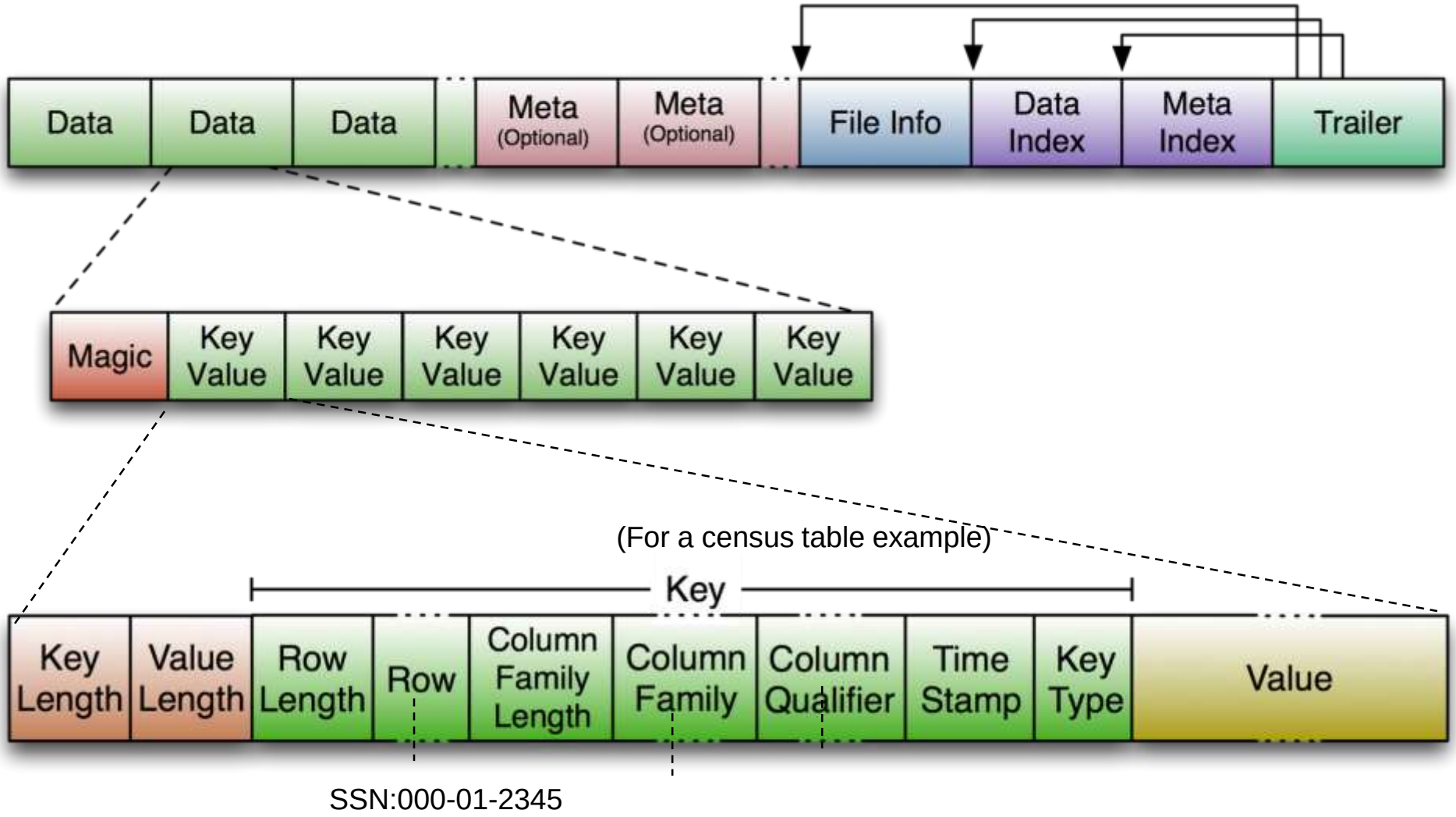
- Data blocks
  - A data block will contain data that is either compressed, or uncompressed, but not a combination of both. A data block includes the delete markers as well as the puts.
- Index blocks
  - When looking up a specific row, index blocks are used by HBase to quickly jump to the right location in an HFile.
- Bloom filter block
  - These blocks are used to store bloom filter index related information. Bloom filter blocks are used to skip parsing the file when looking for a specific key.
- Trailer block
  - This block contains offsets to the file's other variable-sized parts. It also contains the HFile version.

HFile

- HFile's are the actual storage files, specifically created to serve one purpose:
  → store HBase's data fast and efficiently.
- based on Hadoop's TFile and mimic the SSTable format used in Googles BigTable architecture.
- default block size is 64KB (HDFS 128MB = HFile x 2048)



HFile



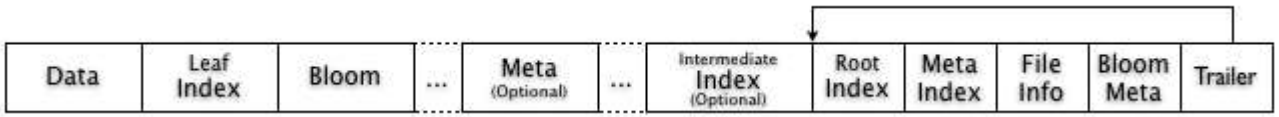
Source: http://blog.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/

Bloom Filter

- data structure which is designed to predict whether a given element is a member of a set of data.
- A positive result from a Bloom filter is not always accurate, but a negative result is guaranteed to be accurate.
- Bloom filters are designed to be "accurate enough" for sets of data which are so large that conventional hashing mechanisms would be impractical.
- Bloom filters provide a lightweight in-memory structure to reduce the number of disk reads for a given Get operation to only the StoreFiles likely to contain the desired Row. Potential performance gain increases with the number of parallel reads.
- Bloom filters are stored in metadata of each HFile and never need to be updated. When an HFile is opened the Bloom filter is loaded into memory.

The main feature of HFile v2 are “inline blocks”

- break index and Bloom Filter per block, instead of having the whole index and Bloom Filter of the whole file in memory.
- In this way you can keep in ram just what you need.
- Since the index is moved to block level you then have a multi-level index, meaning each block has its own index (leaf-index). The last key of each block is kept to create the intermediate/index that makes the multilevel-index b+tree like.

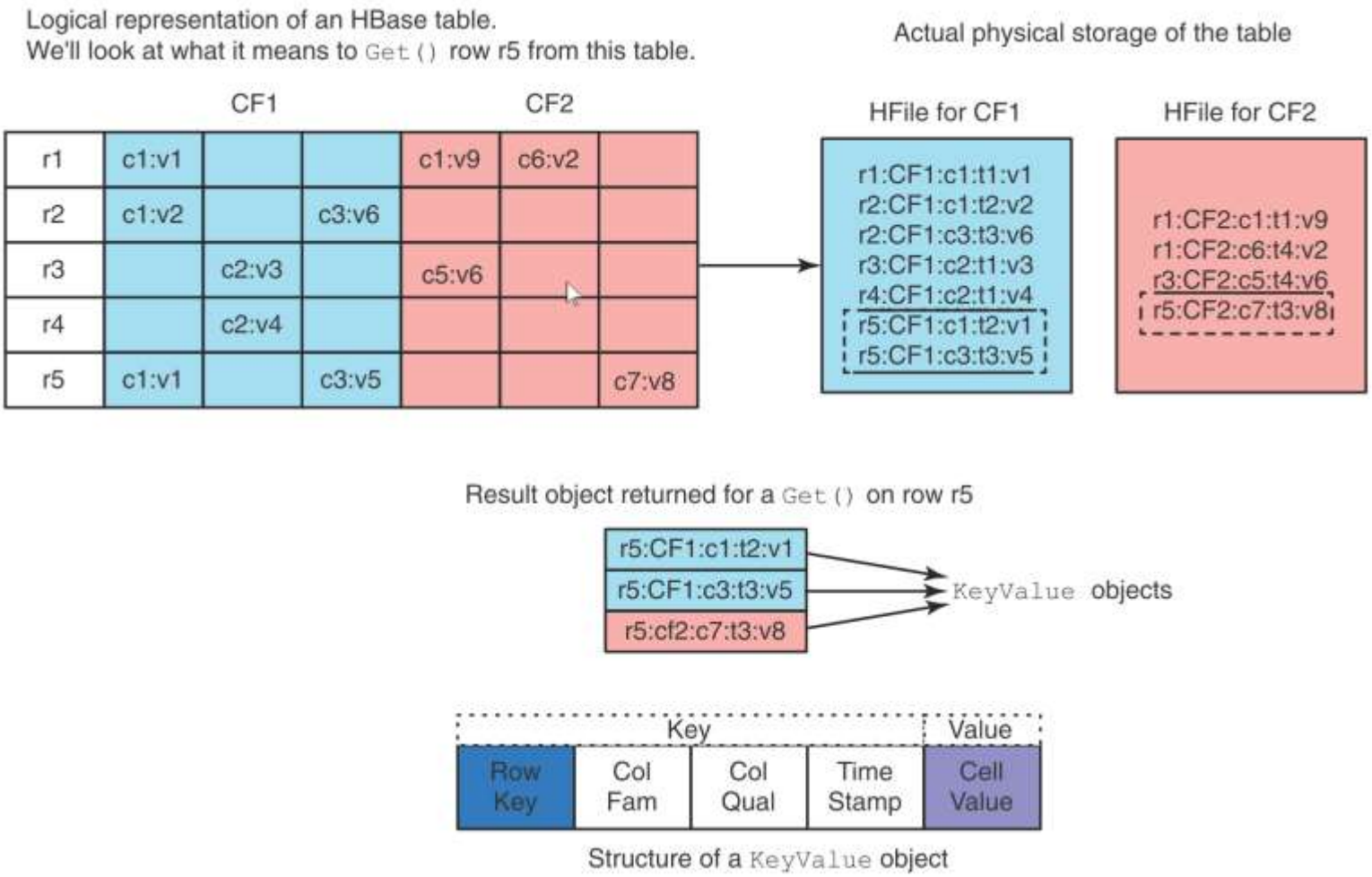


Outline

- Introduction
- HBase
- Table Design
- Bloom Filter
- Use Case

- How many column families should the table have?
- What data goes into what column family?
- How many columns should be in each column family?
- What should the column names be? Although column names don’t have to be defined on table creation, you need to know them when you write or read data.
- What information should go into the cells?
- How many versions should be stored for each cell?
- What should the rowkey structure be, and what should it contain?

Logical to physical translation of an HBase table



Dimiduk, N., & Khurana, A. (2013). HBase in action. Shelter Island, NY: Manning.

How many Column Families?

- Rule of thumb: not more than 3
- All data for a given column family goes into a single store on HDFS. A store may consist of multiple HFiles, but ideally, on compaction, you achieve a single HFile.
- Columns in a column family are all stored together on disk, and that property can be used to isolate columns with different access patterns by putting them in different column families. This is also why HBase is called a column-family-oriented store.

How to approach schema design

	Key			
	Rowkey	Col fam	Col qual	Time stamp
Limit rows	✓	X	X	X
Limit HFiles	✓	✓	X	✓
Limit disk I/O	✓	✓	X	✓
Limit network I/O	✓	✓	✓	✓

Outline

- Introduction
- HBase
- Table Design
- Bloom Filter
- Use Case

Approximate set membership problem

Trade-off between the space and the false positive probability.

Generalize the hashing idea.

Suppose we have a set $S = \{s_1, s_2, \ldots, s_m\} \subseteq$ universe U

Represent S in such a way we can quickly answer

"Is x an element of S ?"

To take as little space as possible,
we allow false positive (i.e. $x \notin S$, but we answer yes)

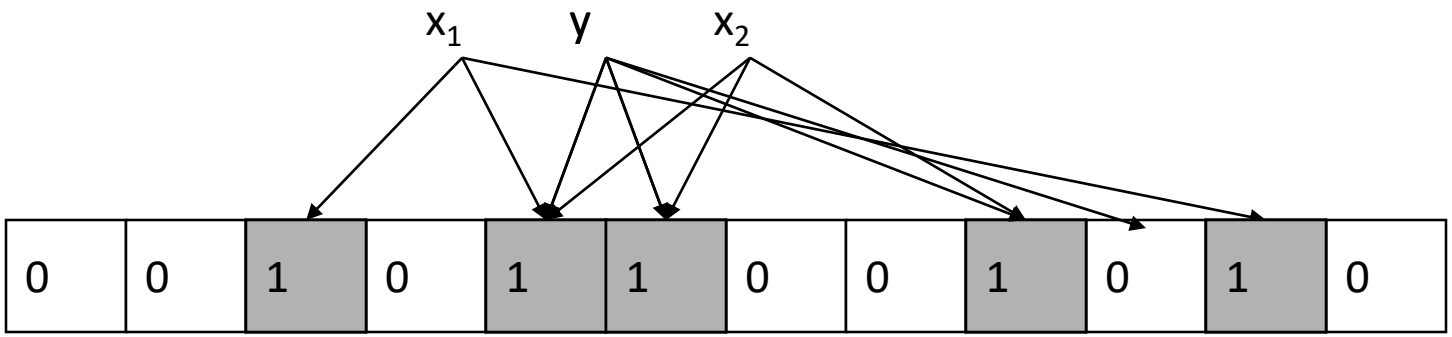
If $x \in S$, we must answer yes .

Bloom filter

Consist of an arrays A[n] of n bits (space), and k independent random hash functions

$h_1,\ldots,h_k : U \dashrightarrow \{0, 1, \ldots, n-1\}$

1. Initially set the array to 0
2. $\forall s \in S$, $A[h_i(s)] = 1$ for $1 \leq i \leq k$
   (an entry can be set to 1 multiple times, only the first times has an effect)
3. To check if $x \in S$, we check whether all location $A[h_i(x)]$ for $1 \leq i \leq k$ are set to 1
   a. If not, clearly $x \notin S$.
   b. If all $A[h_i(x)]$ are set to 1 ,we assume $x \in S$



If only 1s appear, conclude that y is in S
This may yield false positive

Bloom filter

- A Bloom filter is like a hash table, and simply uses one bit to keep track whether an item hashed to the location.
- If k=1 , it's equivalent to a hashing based fingerprint system.
- If n=cm for small constant c, such as c=8 , then k=5 or 6, the false positive probability is just over 2% .
- It's interesting that when k is optimal

  k=ln(2)*(n/m) , then p= 1/2.

  An optimized Bloom filter looks like a random bit-string

Outline

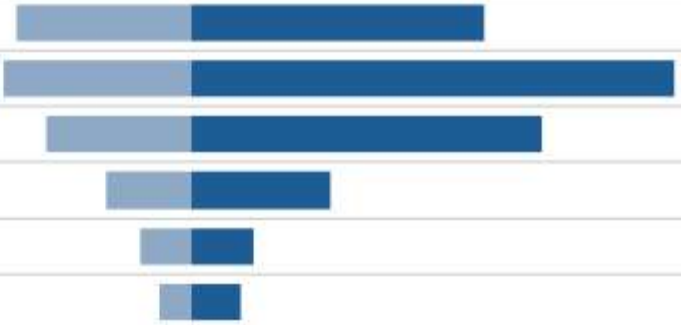
- Introduction
- HBase
- Bloom Filter
- Use Case

facebook

Real-time Analytics at Facebook

Zheng Shao
10/18/2011

Age	Female (31%)	Male (62%)
13–17	7.9%	13%
18–24	8.5%	22%
25–34	6.6%	16%
35–44	3.9%	6.3%
45–54	2.3%	2.8%
55+	1.5%	2.3%



Language	
English (US)	(34%)
Spanish	(12%)
Arabic	(6.1%)
Turkish	(6%)
English (UK)	(5.3%)
French (France)	(5.2%)
Spanish (Spain)	(4.6%)
Indonesian	(3.9%)
Italian	(3.7%)
German	(2.7%)
Portuguese (Brazil)	(2.6%)
Thai	(1.2%)
Russian	(0.96%)
Polish	(0.91%)
More	

Country	
United States	(13%)
Turkey	(5.9%)
Philippines	(5%)
Indonesia	(4.5%)
India	(4.4%)
Mexico	(4.3%)
Egypt	(4.1%)
Italy	(3.8%)
Brazil	(2.7%)
Argentina	(2.7%)
Germany	(2.6%)
Malaysia	(2.6%)
United Kingdom	(2.3%)
Spain	(1.8%)
More	

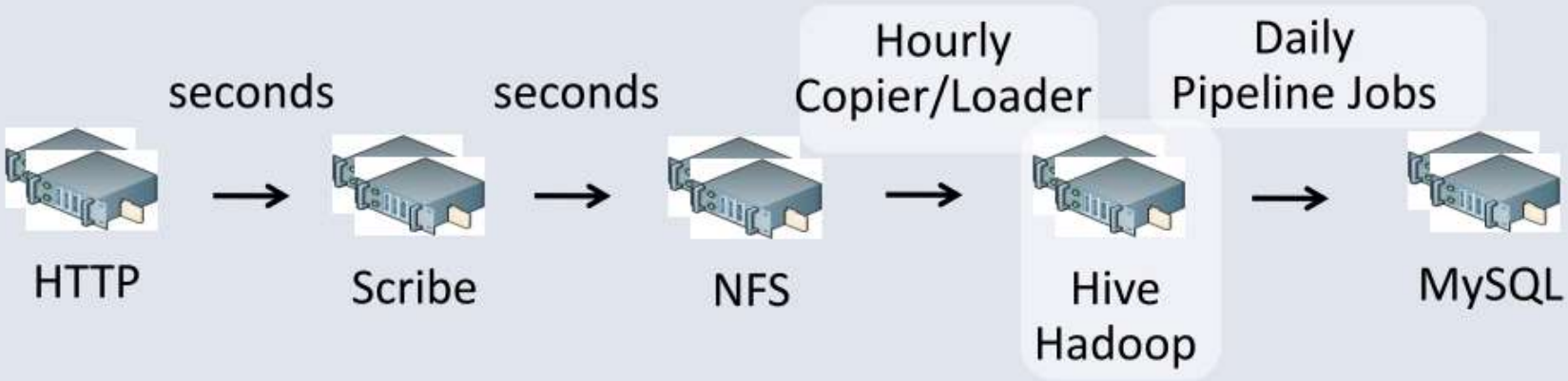
Facebook

Example: Facebook Insights

- > 20B Events per Day
- 1M Counter Updates per Second
  - 100 Nodes Cluster
  - 10K OPS per Node
- ”Like” button triggers AJAX request
- Event written to log file
- 30mins current for website owner

Web ➜ Scribe ➜ Ptail ➜ Puma ➜ HBase

Analytics based on Hadoop/Hive

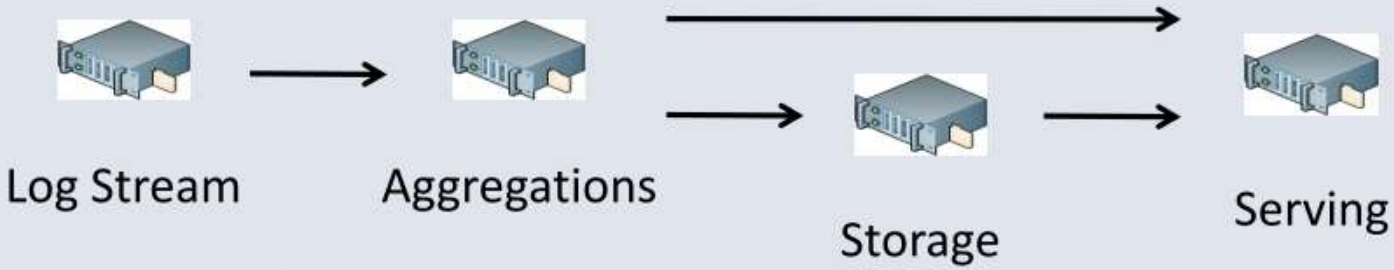


- 3000-node Hadoop cluster
- Copier/Loader: Map-Reduce hides machine failures
- Pipeline Jobs: Hive allows SQL-like syntax
- Good scalability, but poor latency! 24 – 48 hours.

How to Get Lower Latency?

- Small-batch Processing
  - Run Map-reduce/Hive every hour, every 15 min, every 5 min, …
  - How do we reduce per-batch overhead?
- Stream Processing
  - Aggregate the data as soon as it arrives
  - How to solve the reliability problem?

Overview

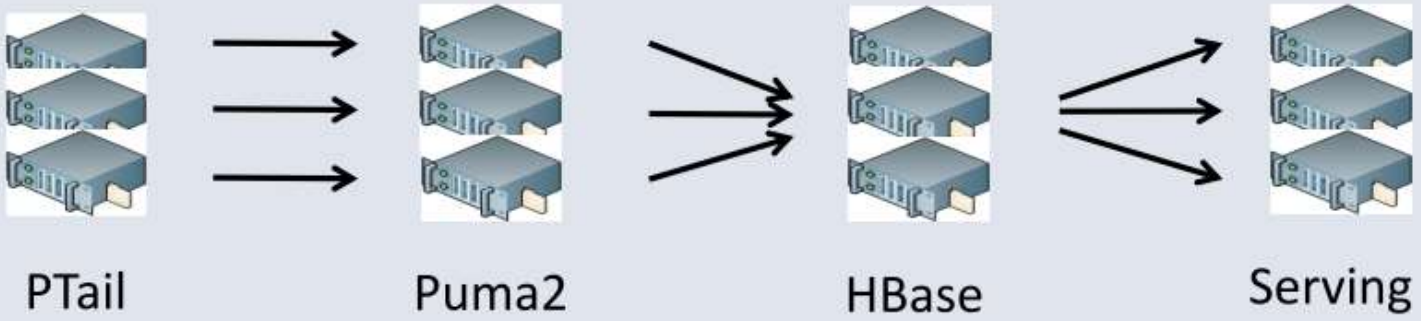


- ~ 1M log lines per second, but light read
- Multiple Group-By operations per log line
- The first key in Group By is always time/date-related
- Complex aggregations: Unique user count, most frequent elements

MySQL and HBase: one page

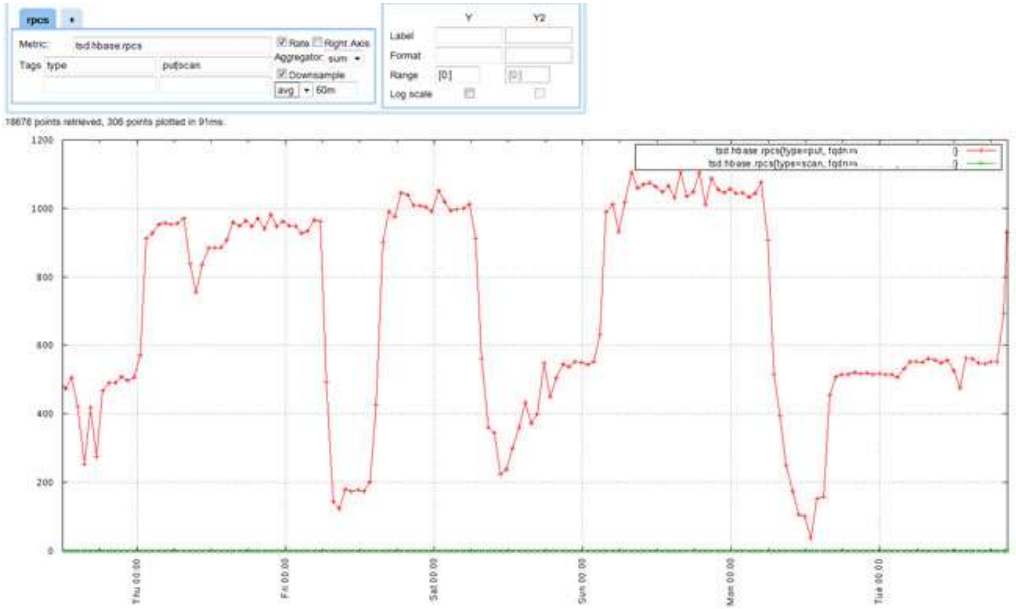
	MySQL	HBase
Parallel	Manual sharding	Automatic load balancing
Fail-over	Manual master/slave switch	Automatic
Read efficiency	High	Low
Write efficiency	Medium	High
Columnar support	No	Yes

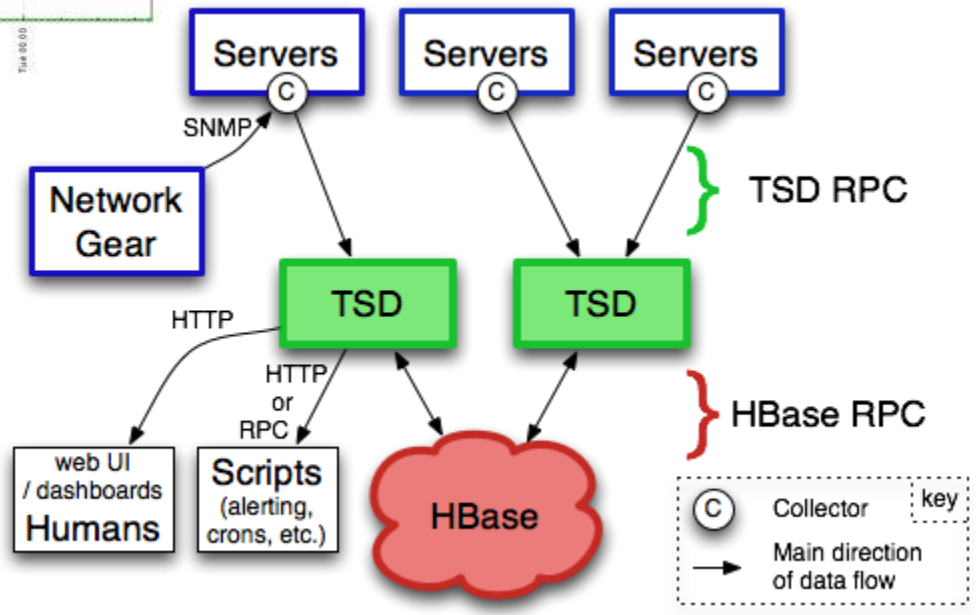
Puma2 Architecture



- PTail provide parallel data streams
- For each log line, Puma2 issue “increment” operations to HBase. Puma2 is symmetric (no sharding).
- HBase: single increment on multiple columns

OpenTSDB


rpcs
Metric:
tsd.hbase.rpcs
Tags type
putscan
Rate
Right Axis
Aggregator: sum
Downsample
avg
60m
Label
Format
Range
Log scale
Y
Y2
18676 points retrieved, 306 points plotted in 91ms.
1200
1000
800
600
400
200
0
Thu 00:00
Fri 00:00
Sat 00:00
Sun 00:00
Mon 00:00
Tue 00:00


Servers
C
Servers
C
Servers
C
SNMP
Network Gear
TSD RPC
TSD
TSD
HTTP
HTTP or RPC
HBase RPC
web UI / dashboards
Humans
Scripts (alerting, crons, etc.)
HBase
C
Collector
key
Main direction of data flow