Unit V: Distributed Data Storage and Big Data

References:

1. G. Coulouris, J. Dollimore and T. Kindberg; **Distributed Systems Concepts and Design,4th Edition.**

2. Andrew S. Tanenbaum and Maarten van Steen**; Distributed Systems: Principles and Paradigms, 2nd Edition.**

Distributed Data Storage and Big Data

5.1 Distributed File Systems

5.1.1 Google File System (GFS):
Architecture, Operations, features and use cases

5.1.2 Hadoop Distributed File System (HDFS):
Architecture, Operations, features and use cases

5.2 NoSQL Databases:

5.2.1 Cassandra: Architecture, design principles,
Data Model and Query Language, features and use cases

5.2.2 MongoDB: Architecture, design principles,
Data Model and Query Language, features and use cases

5.3 Big Data Processing Frameworks

5.3.1 Hadoop: Key features and use cases

5.3.2 Spark : Key Features and use cases

DISTRIBUTED FILE SYSTEMS

- File system were originally developed for centralized computer systems and desktop computers.
- File system was as an operating system facility providing a convenient programming interface to disk storage.
- A **Distributed File System** (DFS) is simply a classical model of a file system distributed across multiple machines. The purpose is to promote sharing of dispersed files.
- The resources on a particular machine are **local** to itself. Resources on other machines are **remote**.
- A file system provides a service for clients. The server interface is the normal set of file operations: create, read, etc. on files.

DISTRIBUTED FILE SYSTEMS

- Distributed file systems support the sharing of information in the form of files throughout the intranet.

- A distributed file system enables programs to store and access remote files exactly as they do on local ones, allowing users to access files from any computer on the intranet.

- Recent advances in higher bandwidth connectivity of switched local networks and disk organization have lead high performance and highly scalable file systems.

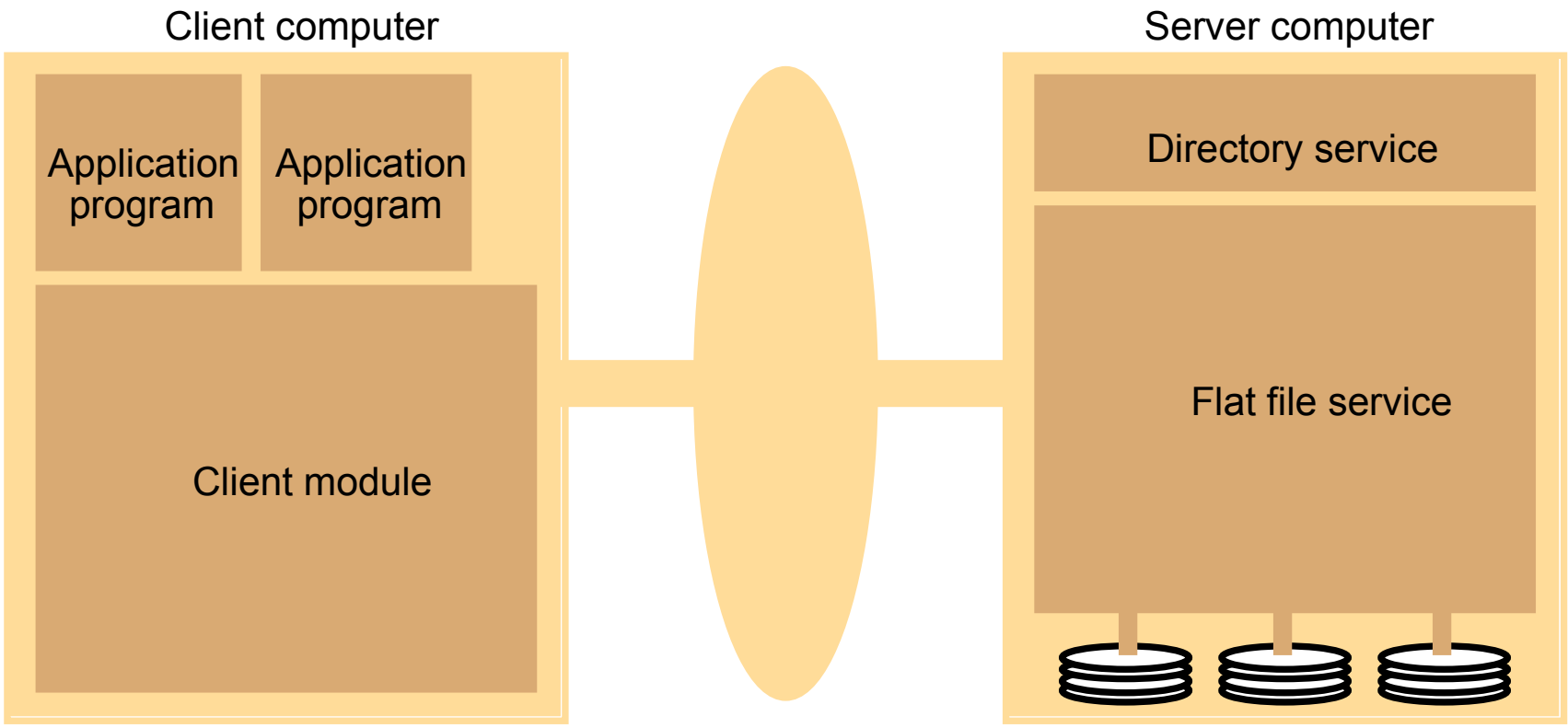
DISTRIBUTED FILE SYSTEMS

- Clients, servers, and storage are dispersed across machines. Configuration and implementation may vary -
  a) Servers may run on dedicated machines, OR
  b) Servers and clients can be on the same machines.
  c) The OS itself can be distributed (with the file system a part of that distribution.
  d) A distribution layer can be interposed between a conventional OS and the file system.

- Clients should view a DFS the same way they would a centralized FS; the distribution is hidden at a lower level.

- Performance is concerned with throughput and response time.

DISTRIBUTED FILE SYSTEMS



File service architecture

Distributed File system requirements

- Related requirements in distributed file systems are:
  - Transparency
  - Concurrency
  - Replication
  - Heterogeneity
  - Fault tolerance
  - Consistency
  - Security
  - Efficiency

Naming and Transparency

Naming and Transparency

Naming is the mapping between logical and physical objects.

- Example: A user filename maps to <cylinder, sector>.
- In a conventional file system, it's understood where the file actually resides; the system and disk are known.
- In a **transparent** DFS, the location of a file, somewhere in the network, is hidden.
- **File replication** means multiple copies of a file; mapping returns a SET of locations for the replicas.

Location transparency -

a) The name of a file does not reveal any hint of the file's physical storage location.
b) File name still denotes a specific, although hidden, set of physical disk blocks.

Remote File Access

Remote File Access

CACHING

- Reduce network traffic by retaining recently accessed disk blocks in a cache, so that repeated accesses to the same information can be handled locally.
- If required data is not already cached, a copy of data is brought from the server to the user.
- Perform accesses on the cached copy.
- Files are identified with one master copy residing at the server machine,
- Copies of (parts of) the file are scattered in different caches.
- **Cache Consistency Problem** -- Keeping the cached copies consistent with the master file.
- A remote service ((RPC) has these characteristic steps:
  a) The client makes a request for file access.
  b) The request is passed to the server in message format.
  c) The server makes the file access.
  d) Return messages bring the result back to the client.

- This is equivalent to performing a disk access for each request.

Remote File Access

Remote File Access

STATEFUL VS. STATELESS SERVICE:

Stateful: A server keeps track of information about client requests.

- It maintains what files are opened by a client; connection identifiers; server caches.
- Memory must be reclaimed when client closes file or when client dies.

Stateless: Each client request provides complete information needed by the server (i.e., filename, file offset).

- The server can maintain information on behalf of the client, but it's not required.

Remote File Access

STATEFUL VS. STATELESS SERVICE:

Performance is better for stateful.

- Don't need to parse the filename each time, or "open/close" file on every request.

Fault Tolerance: A stateful server loses everything when it crashes.

- Server must poll clients in order to renew its state.
- Client crashes force the server to clean up its encached information.
- Stateless remembers nothing so it can start easily after a crash.

Remote File Access

FILE REPLICATION:

- Duplicating files on multiple machines improves availability and performance.
- Placed on failure-independent machines (they won't fail together).
- The main problem is consistency - when one copy changes, how do other copies reflect that change? Often there is a tradeoff: consistency versus availability and performance.

Google File System (GFS)

Google Disk Farm

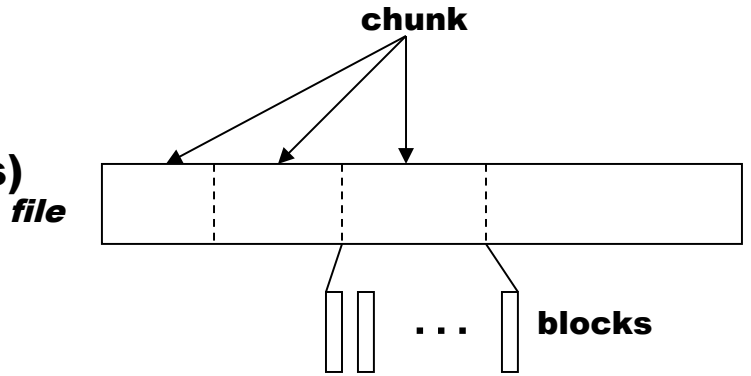
Early days…

…today

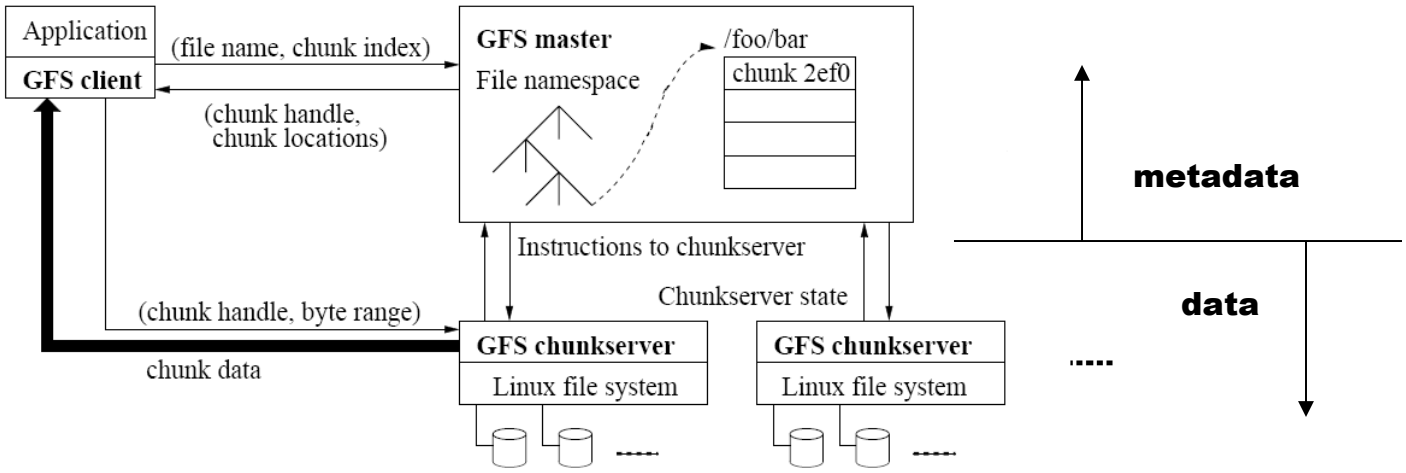
Design

- Design factors
  - Failures are common (built from inexpensive commodity components)
  - Files
    - large (multi-GB)
    - mutation principally via appending new data
    - low-overhead atomicity essential
  - Co-design applications and file system API
  - Sustained bandwidth more critical than low latency

- File structure
  - **Divided into 64 MB chunks**
  - **Chunk identified by 64-bit handle**
  - **Chunks replicated (default 3 replicas)**
  - **Chunks divided into 64KB blocks**
  - **Each block has a 32-bit checksum**



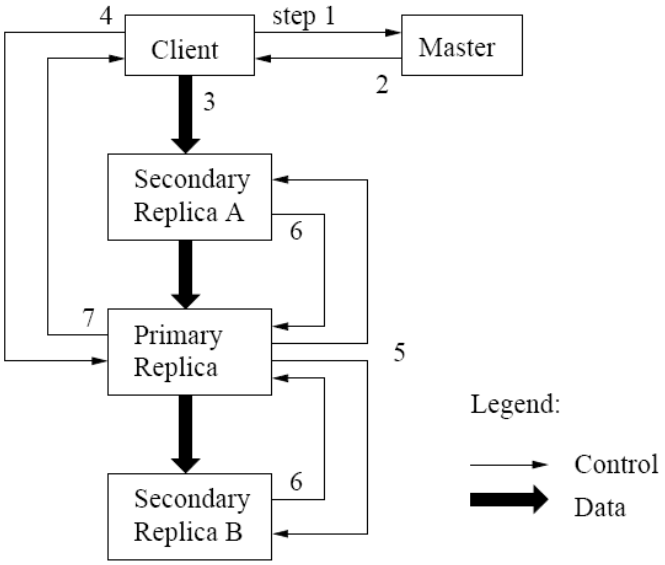
Architecture



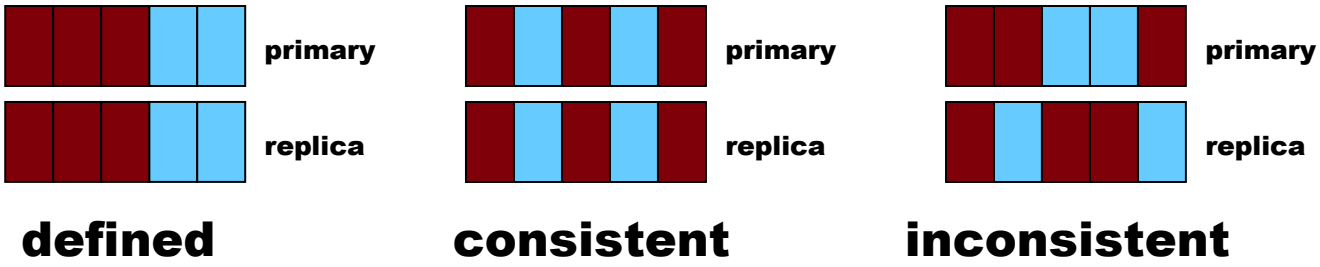
- Master
  - Manages namespace/metadata
  - Manages chunk creation, replication, placement
  - Performs snapshot operation to create duplicate of file or directory tree
  - Performs checkpointing and logging of changes to metadata
- Chunkservers
  - Stores chunk data and checksum for each block
  - On startup/failure recovery, reports chunks to master
  - Periodically reports sub-set of chunks to master (to detect no longer needed chunks)

Mutation operations

- Primary replica
  - Holds lease assigned by master (60 sec. default)
  - Assigns serial order for all mutation operations performed on replicas
- Write operation
  - 1-2: client obtains replica locations and identity of primary replica
  - 3: client pushes data to replicas (stored in LRU buffer by chunk servers holding replicas)
  - 4: client issues update request to primary
  - 5: primary forwards/performs write request
  - 6: primary receives replies from replica
  - 7: primary replies to client
- Record append operation
  - Performed atomically (one byte sequence)
  - At-least-once semantics
  - Append location chosen by GFS and returned to client
  - Extension to step 5:
    - If record fits in current chunk: write record and tell replicas the offset
    - If record exceeds chunk: pad the chunk, reply to client to use next chunk



Consistency Guarantees



- Write
  - **Concurrent writes may be consistent but undefined**
  - **Write operations that are large or cross chunk boundaries are subdivided by client into individual writes**
  - **Concurrent writes may become interleaved**
- Record append
  - **Atomically, at-least-once semantics**
  - **Client retries failed operation**
  - **After successful retry, replicas are defined in region of append but may have intervening undefined regions**
- Application safeguards
  - **Use record append rather than write**
  - **Insert checksums in record headers to detect fragments**
  - **Insert sequence numbers to detect duplicates**

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Metadata management

Logical structure

pathname	lock	chunk list
/home	read	Chunk4400488,…
/save		Chunk8ffe07783,…
/home/user/foo	write	Chunk6254ee0,…
/home/user	read	Chunk88f703,…

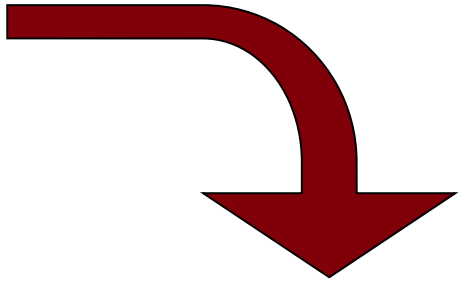
- Namespace
  - Logically a mapping from pathname to chunk list
  - Allows concurrent file creation in same directory
  - Read/write locks prevent conflicting operations
  - File deletion by renaming to a hidden name; removed during regular scan
- Operation log
  - Historical record of metadata changes
  - Kept on multiple remote machines
  - Checkpoint created when log exceeds threshold
  - When checkpointing, switch to new log and create checkpoint in separate thread
  - Recovery made from most recent checkpoint and subsequent log
- Snapshot
  - Revokes leases on chunks in file/directory
  - Log operation
  - Duplicate metadata (not the chunks!) for the source
  - On first client write to chunk:
    - Required for client to gain access to chunk
    - Reference count > 1 indicates a duplicated chunk
    - Create a new chunk and update chunk list for duplicate

Chunk/replica management

- Placement
  - On chunkservers with below-average disk space utilization
  - Limit number of “recent” creations on a chunkserver (since access traffic will follow)
  - Spread replicas across racks (for reliability)
- Reclamation
  - Chunk become garbage when file of which they are a part is deleted
  - Lazy strategy (garbage college) is used since no attempt is made to reclaim chunks at time of deletion
  - In periodic “HeartBeat” message chunkserver reports to the master a subset of its current chunks
  - Master identifies which reported chunks are no longer accessible (i.e., are garbage)
  - Chunkserver reclaims garbage chunks
- Stale replica detection
  - Master assigns a version number to each chunk/replica
  - Version number incremented each time a lease is granted
  - Replicas on failed chunkservers will not have the current version number
  - Stale replicas removed as part of garbage collection

Performance

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB



Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

HDFS

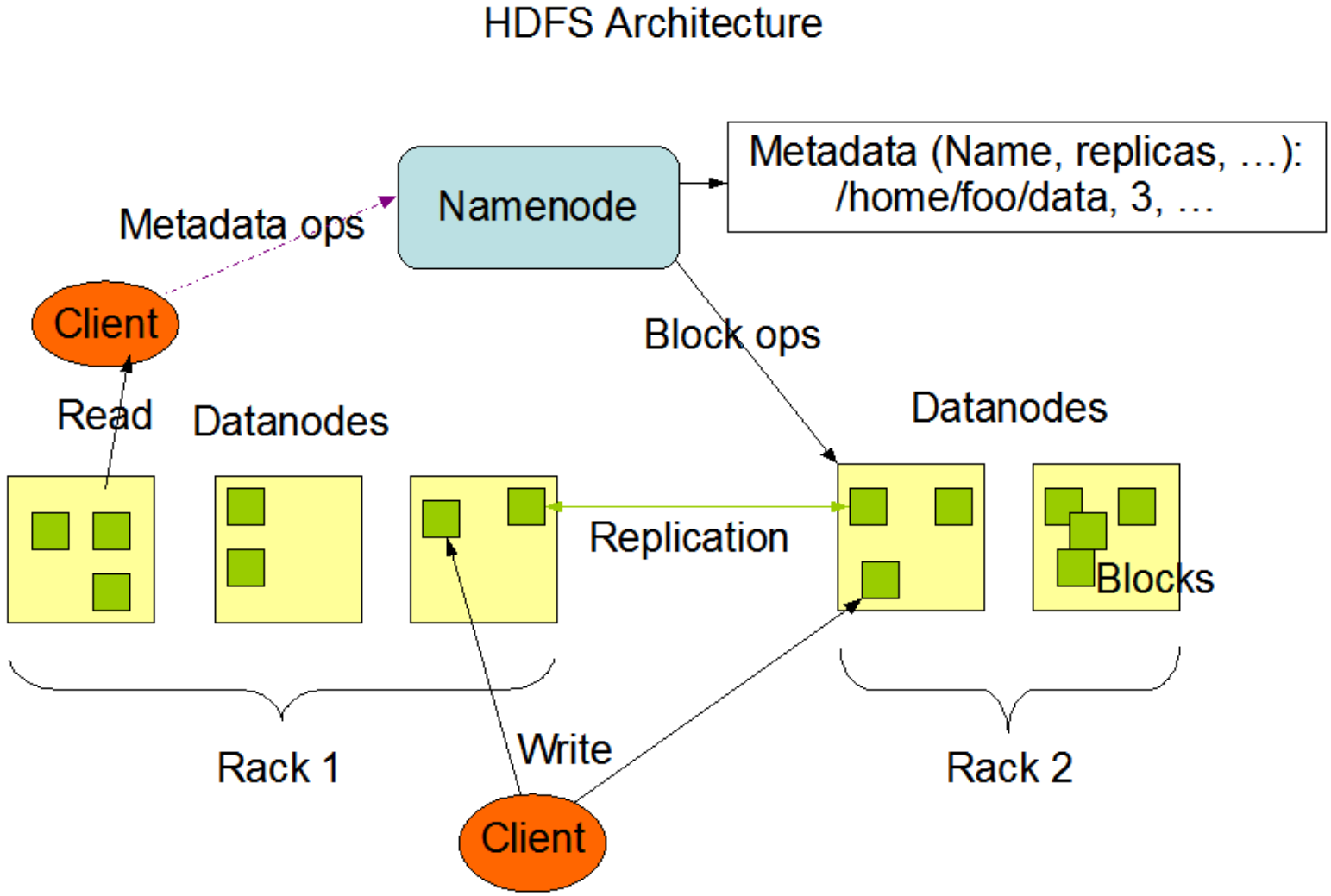
GOALS OF HDFS

- Very Large Distributed File System
  - 10K nodes, 100 million files, 10PB
- Assumes Commodity Hardware
  - Files are replicated to handle hardware failure
  - Detect failures and recover from them
- Optimized for Batch Processing
  - Data locations exposed so that computations can move to where data resides
  - Provides very high aggregate bandwidth

HADOOP DISTRIBUTED FILE SYSTEM

- Single Namespace for entire cluster
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Files are broken up into blocks
  - Typically 64MB block size
  - Each block replicated on multiple DataNodes
- Intelligent Client
  - Client can find location of blocks
  - Client accesses data directly from DataNode

HDFS ARCHITECTURE



FUNCTIONS OF A NAMENODE

- Manages File System Namespace
  - Maps a file name to a set of blocks
  - Maps a block to the DataNodes where it resides
- Cluster Configuration Management
- Replication Engine for Blocks

NAMENODE METADATA

- Metadata in Memory
  - The entire metadata is in main memory
  - No demand paging of metadata
- Types of metadata
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g. creation time, replication factor
- A Transaction Log
  - Records file creations, file deletions etc

DATANODE

- A Block Server
  - Stores data in the local file system (e.g. ext3)
  - Stores metadata of a block (e.g. CRC)
  - Serves data and metadata to Clients
- Block Report
  - Periodically sends a report of all existing blocks to the NameNode
- Facilitates Pipelining of Data
  - Forwards data to other specified DataNodes

BLOCK PLACEMENT

- Current Strategy
  - One replica on local node
  - Second replica on a remote rack
  - Third replica on same remote rack
  - Additional replicas are randomly placed
- Clients read from nearest replicas
- Would like to make this policy pluggable

HEARTBEATS

- DataNodes send hearbeat to the NameNode
  - Once every 3 seconds
- NameNode uses heartbeats to detect DataNode failure

REPLICATION ENGINE

- NameNode detects DataNode failures
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes

DATA CORRECTNESS

- Use Checksums to validate data
  - Use CRC32
- File Creation
  - Client computes checksum per 512 bytes
  - DataNode stores the checksum
- File access
  - Client retrieves the data and checksum from DataNode
  - If Validation fails, Client tries other replicas

NAMENODE FAILURE

- A single point of failure
- Transaction Log stored in multiple directories
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)
- Need to develop a real HA solution

SECONDARY NAMENODE

- Copies FsImage and Transaction Log from Namenode to a temporary directory
- Merges FSImage and Transaction Log into a new FSImage in temporary directory
- Uploads new FSImage to the NameNode
  - Transaction Log on NameNode is purged

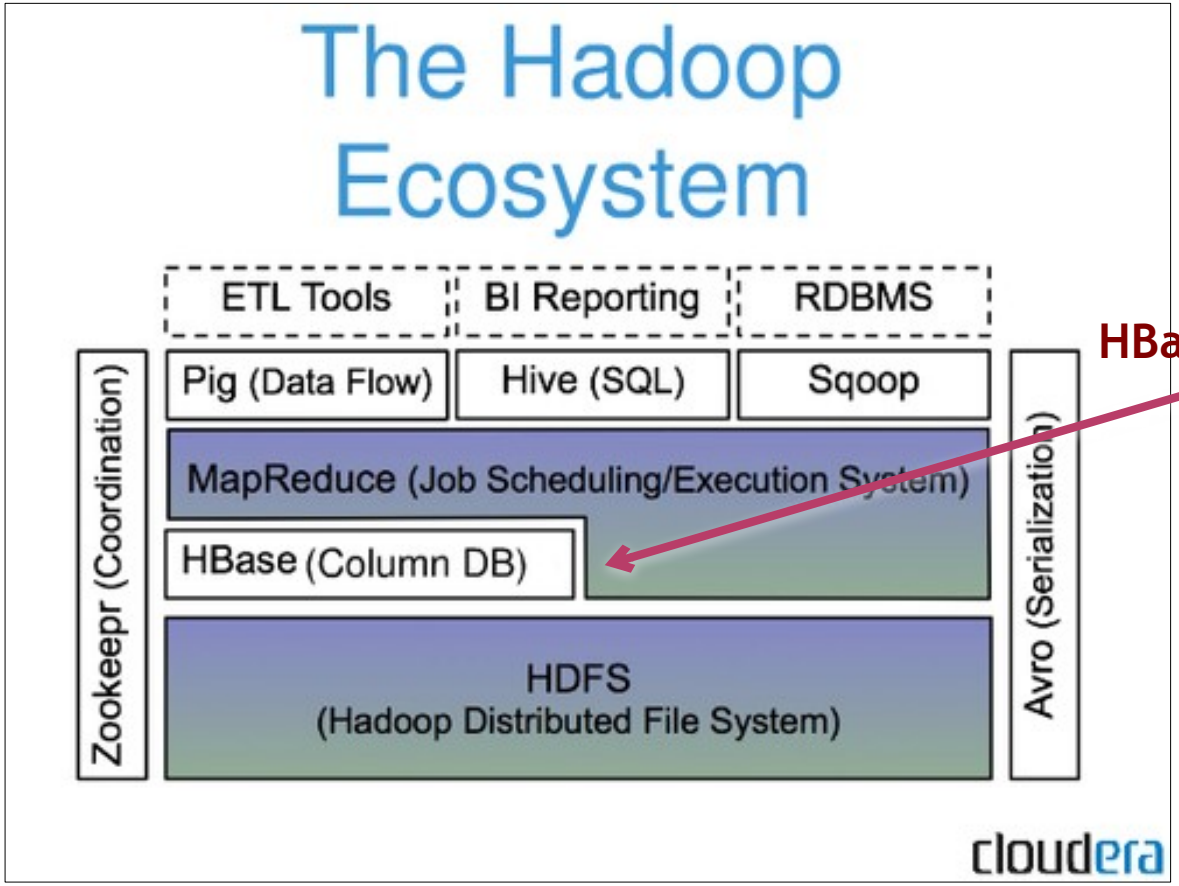
USER INTERFACE

- Commads for HDFS User:
  - hadoop dfs -mkdir /foodir
  - hadoop dfs -cat /foodir/myfile.txt
  - hadoop dfs -rm /foodir/myfile.txt
- Commands for HDFS Administrator
  - hadoop dfsadmin -report
  - hadoop dfsadmin -decommision datanodename
- Web Interface
  - http://host:port/dfshealth.jsp

: OVERVIEW

- HBase is a distributed column-oriented data store built on top of HDFS
- HBase is an Apache open source project whose goal is to provide storage for the Hadoop Distributed Computing
- Data is logically organized into tables, rows and columns

HBASE: PART OF HADOOP'S ECOSYSTEM



HBase is built on top of HDFS

HBase files are internally stored in HDFS

HBASE VS. HDFS (CONT'D)

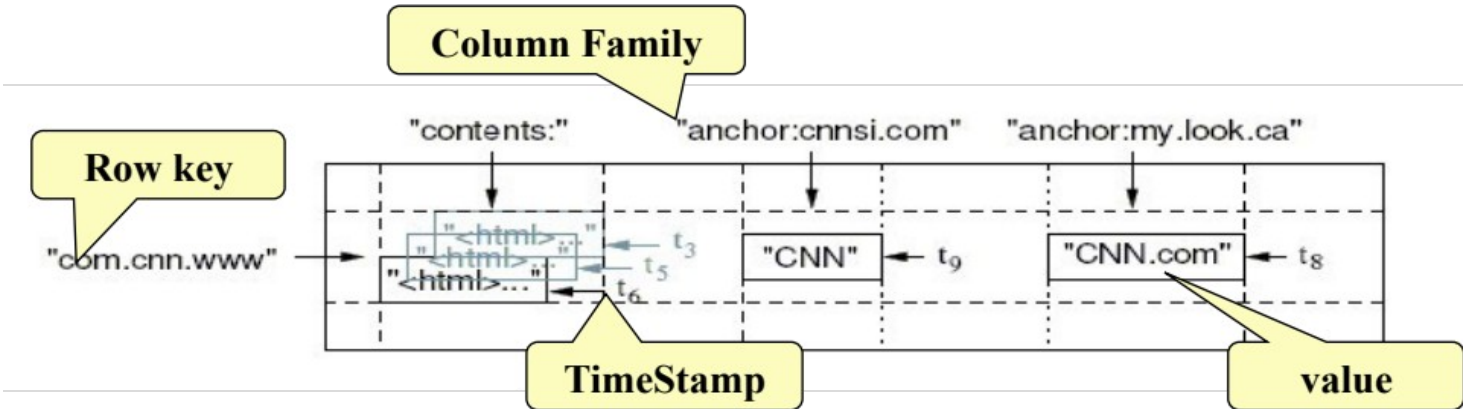
- ***HBase*** is designed to efficiently address the above points
  - Fast record lookup
  - Support for record-level insertion
  - Support for updates (not in place)

- HBase updates are done by creating new versions of values

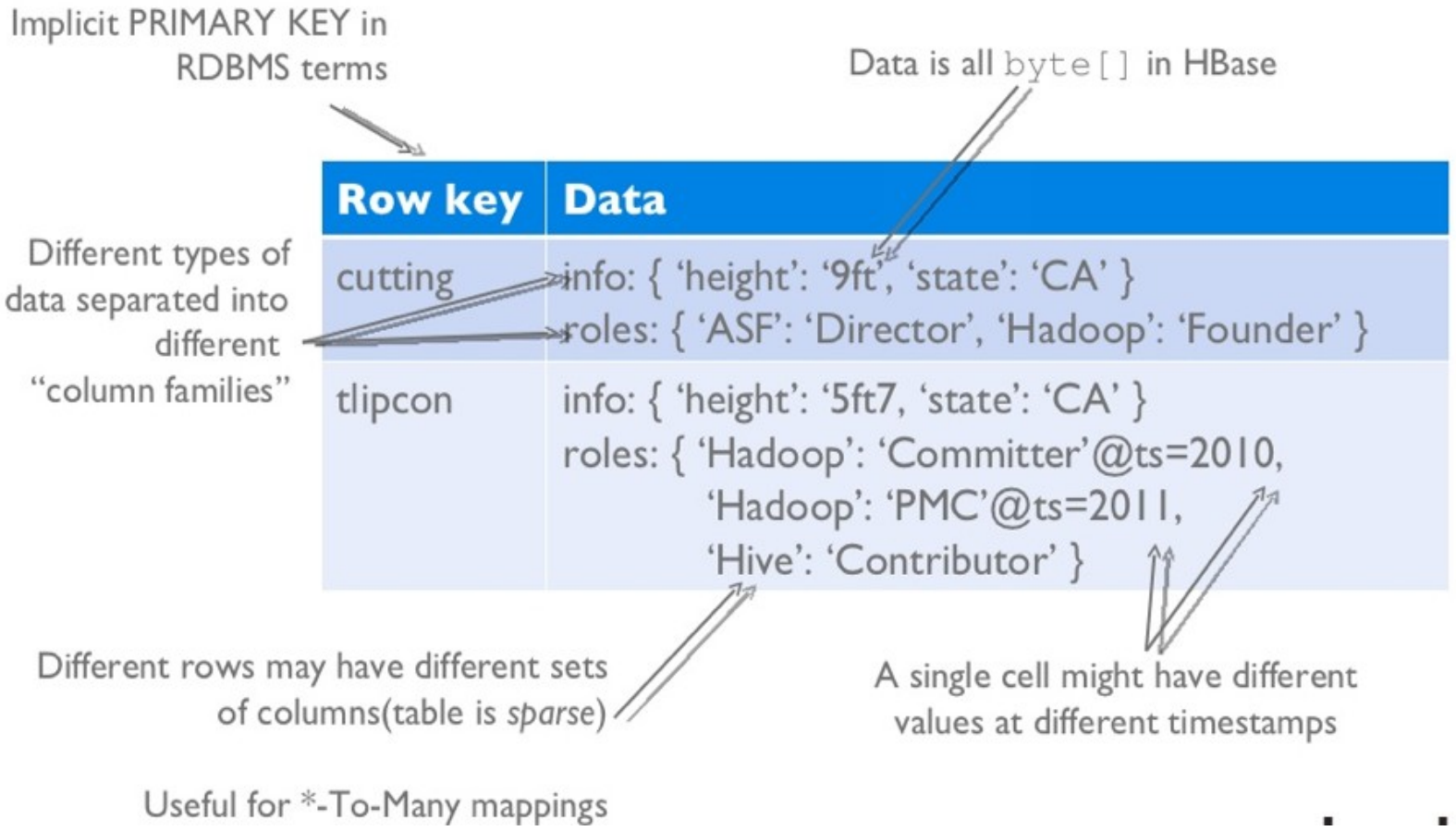
HBASE DATA MODEL

HBASE DATA MODEL

- HBase is based on Google's Bigtable model
  - Key-Value pairs



HBASE LOGICAL VIEW



HBASE: KEYS AND COLUMN FAMILIES

Each record is divided into *Column Families*

Each row has a *Key*

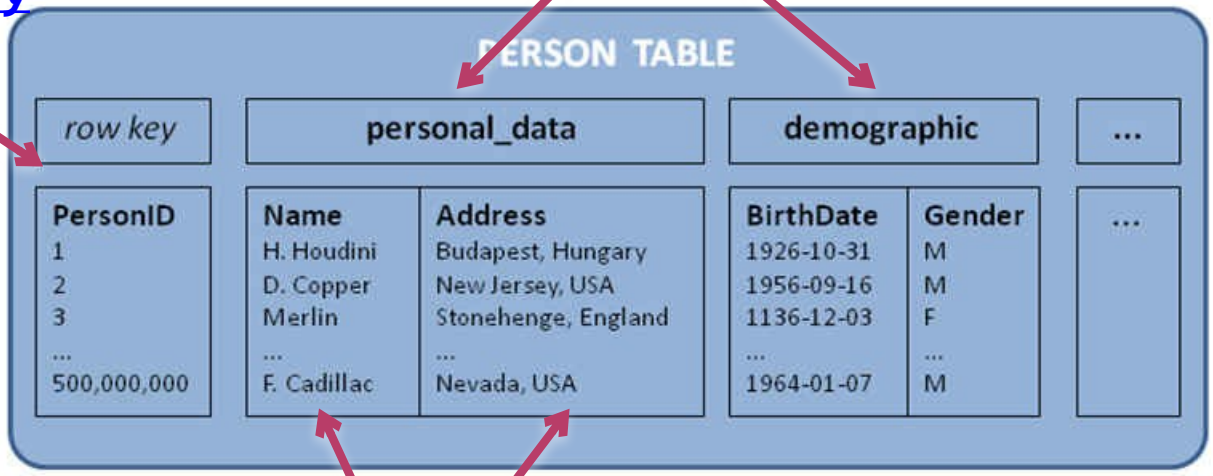
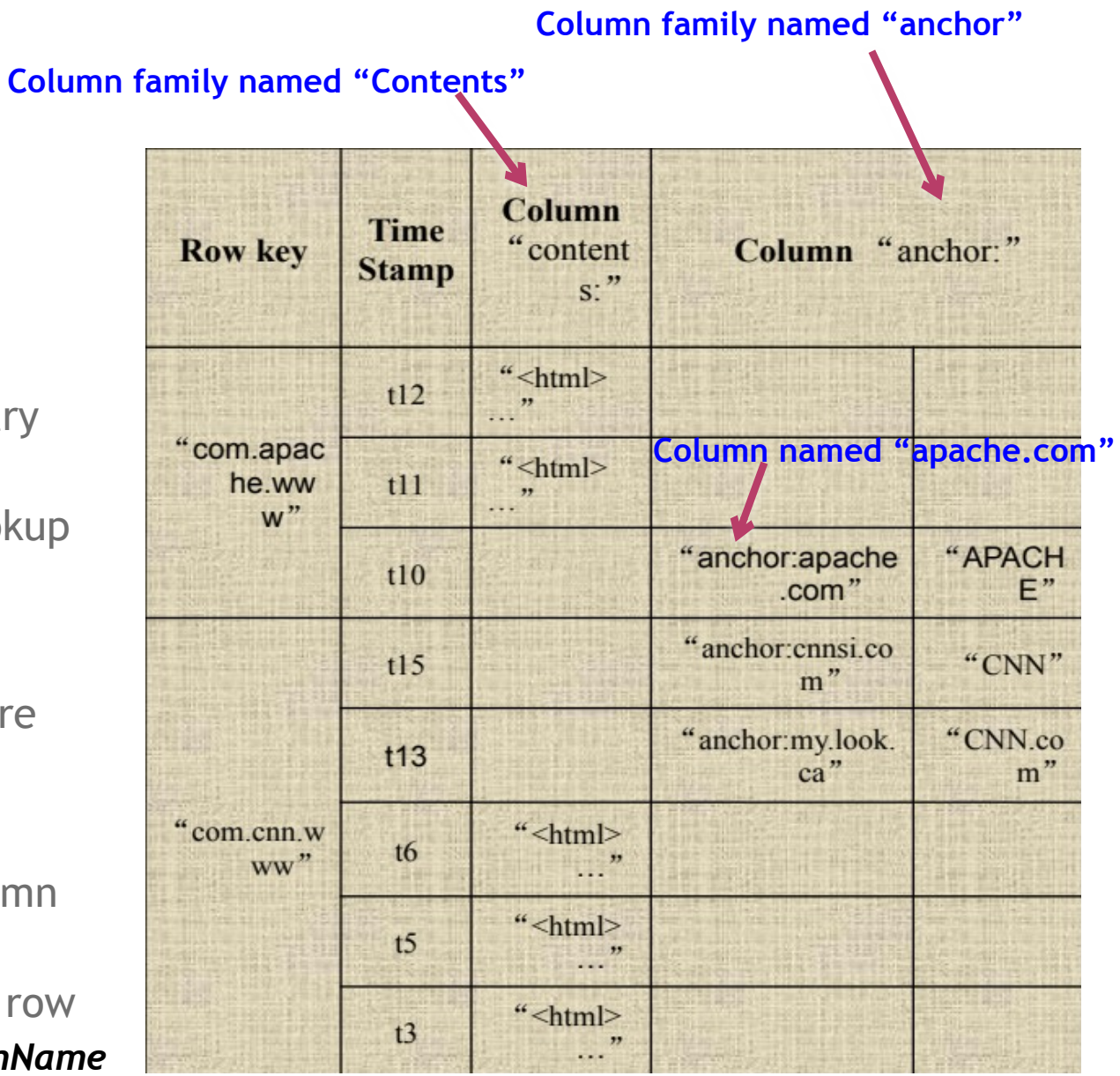


Figure 2 Census Data in Column Families

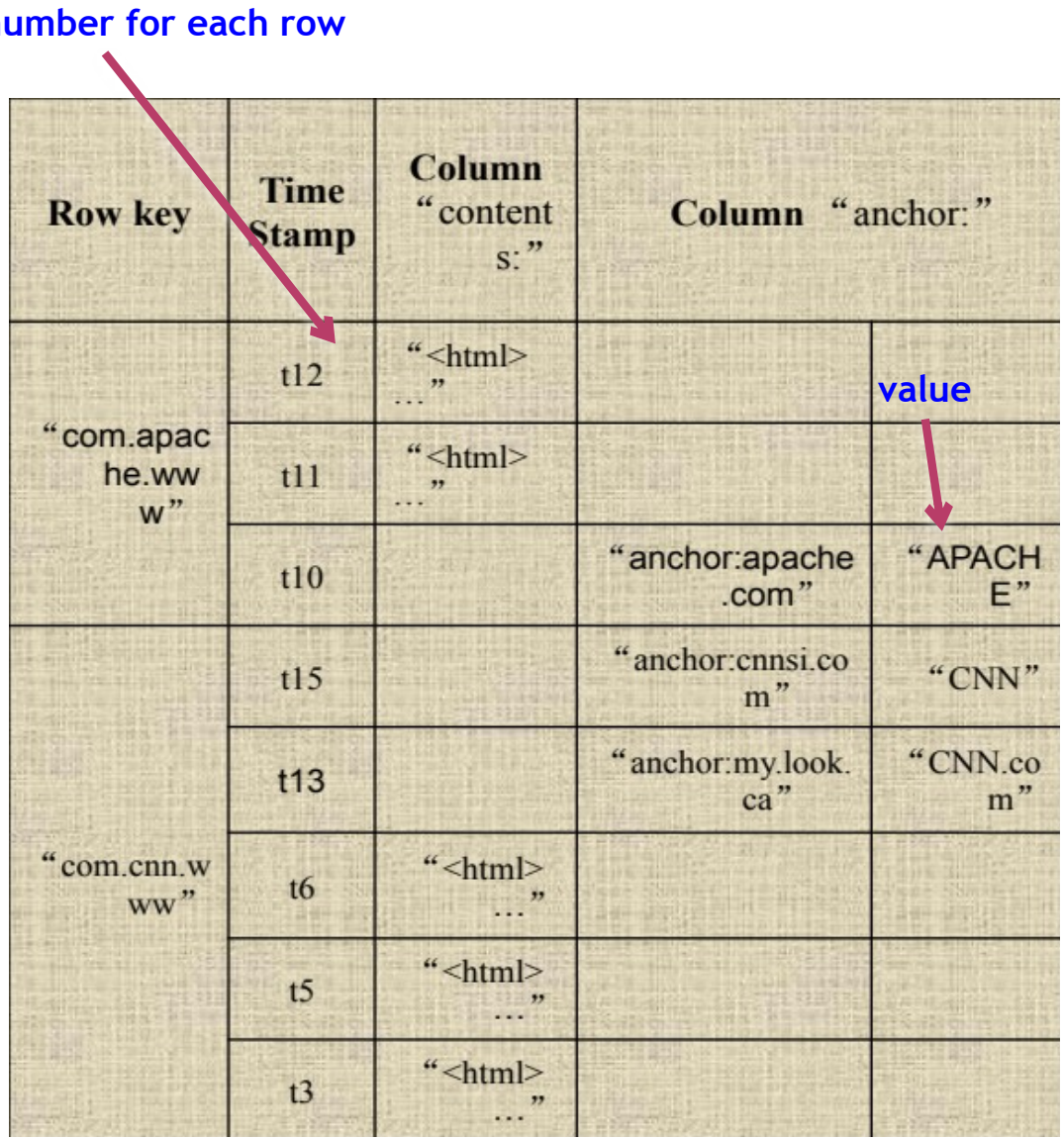
Each column family consists of one or more *Columns*

- **Key**
  - Byte array
  - Serves as the primary key for the table
  - Indexed far fast lookup
- **Column Family**
  - Has a name (string)
  - Contains one or more related columns
- **Column**
  - Belongs to one column family
  - Included inside the row
    - ***familyName:columnName***



Row key	Time Stamp	Column "contents:"	Column "anchor:"	
"com.apache.www"	t12	"<html>…"		
	t11	"<html>…"		
	t10		"anchor:apache.com"	"APACHE"
"com.cnn.www"	t15		"anchor:cnnsi.com"	"CNN"
	t13		"anchor:my.look.ca"	"CNN.com"
	t6	"<html>…"		
	t5	"<html>…"		
	t3	"<html>…"		

- **Version Number**
  - Unique within each key
  - By default→ System's timestamp
  - Data type is Long
- **Value (Cell)**
  - Byte array



Row key	Time Stamp	Column "contents:"	Column "anchor:"	
"com.apache.www"	t12	"<html>…"		
	t11	"<html>…"		
	t10		"anchor:apache.com"	"APACHE"
"com.cnn.www"	t15		"anchor:cnnsi.com"	"CNN"
	t13		"anchor:my.look.ca"	"CNN.com"
	t6	"<html>…"		
	t5	"<html>…"		
	t3	"<html>…"		

NOTES ON DATA MODEL

- HBase schema consists of several ***Tables***
- Each table consists of a set of ***Column Families***
  - Columns are not part of the schema
- HBase has ***Dynamic Columns***
  - Because column names are encoded inside the cells
  - Different cells can have different columns

"Roles" column family has different columns in different cells

Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7, 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

NOTES ON DATA MODEL (CONT'D)

- The ***version number*** can be user-supplied
  - Even does not have to be inserted in increasing order
  - Version number are unique within each key
- Table can be very sparse
  - Many cells are empty
- ***Keys*** are indexed as the primary key

Has two columns [cnnsi.com & my.look.ca]

Row Key	Time Stamp	ColumnFamily `contents`	ColumnFamily `anchor`
"com.cnn.www"	t9		`anchor:cnnsi.com` = "CNN"
"com.cnn.www"	t8		`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:html` = "<html>..."	

HBASE PHYSICAL MODEL

HBASE PHYSICAL MODEL

- Each column family is stored in a separate file (called ***HTables***)
- Key & Version numbers are replicated with each column family
- Empty cells are not stored

HBase maintains a multi-level index on values: *<key, column family, column name, timestamp>*

Table 5.3. ColumnFamily `contents`

Row Key	Time Stamp	ColumnFamily "contents:"
"com.cnn.www"	t6	`contents:html` = "<html>..."
"com.cnn.www"	t5	`contents:html` = "<html>..."
"com.cnn.www"	t3	`contents:html` = "<html>..."

Table 5.2. ColumnFamily `anchor`

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

EXAMPLE

Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7, 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

`info` Column Family

Row key	Column key	Timestamp	Cell value
cutting	info:height	1273516197868	9ft
cutting	info:state	1043871824184	CA
tlipcon	info:height	1273878447049	5ft7
tlipcon	info:state	1273616297446	CA

`roles` Column Family

Row key	Column key	Timestamp	Cell value
cutting	roles:ASF	1273871823022	Director
cutting	roles:Hadoop	1183746289103	Founder
tlipcon	roles:Hadoop	1300062064923	PMC
tlipcon	roles:Hadoop	1293388212294	Committer
tlipcon	roles:Hive	1273616297446	Contributor

Sorted on disk by Row key, Col key, descending timestamp

Milliseconds since unix epoch

cloudera

HBASE REGIONS

- Each HTable (column family) is partitioned horizontally into ***regions***
  - Regions are counterpart to HDFS blocks

Table 5.3. ColumnFamily `contents`

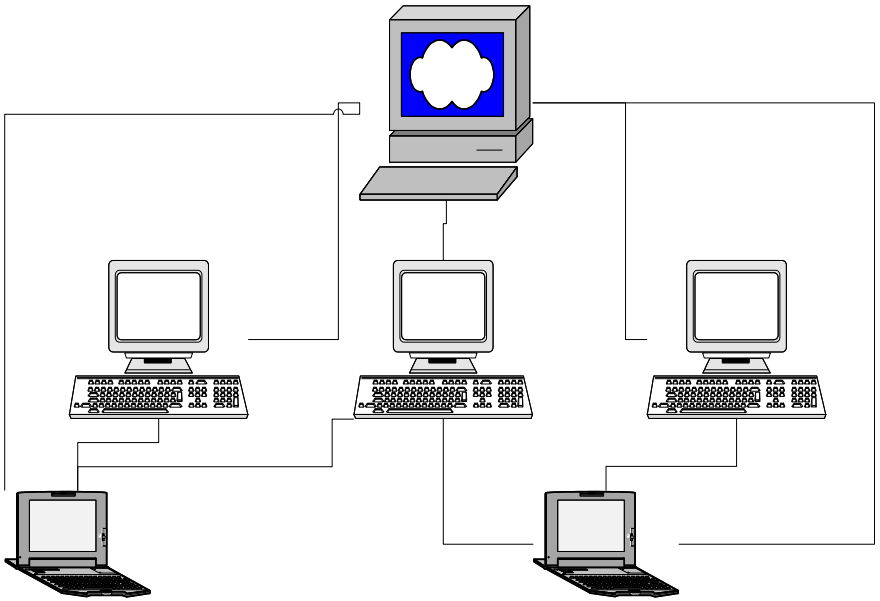
Row Key	Time Stamp	ColumnFamily "contents:"
"com.cnn.www"	t6	contents:html = "<html>..."
"com.cnn.www"	t5	contents:html = "<html>..."
"com.cnn.www"	t3	contents:html = "<html>..."

Each will be one region

HBASE ARCHITECTURE

THREE MAJOR COMPONENTS

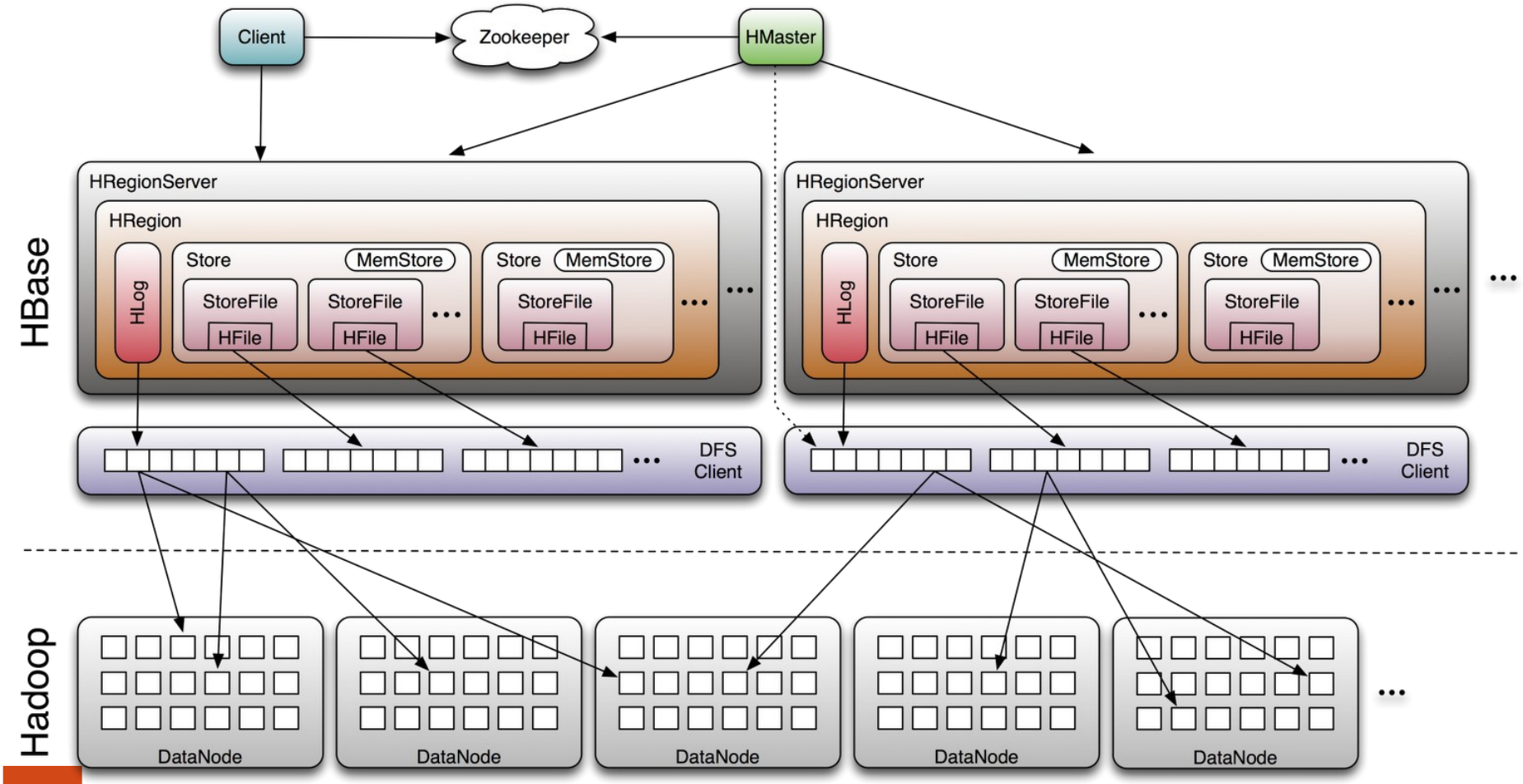
- The HBaseMaster
  - One master
- The HRegionServer
  - Many region servers
- The HBase client



HBASE COMPONENTS

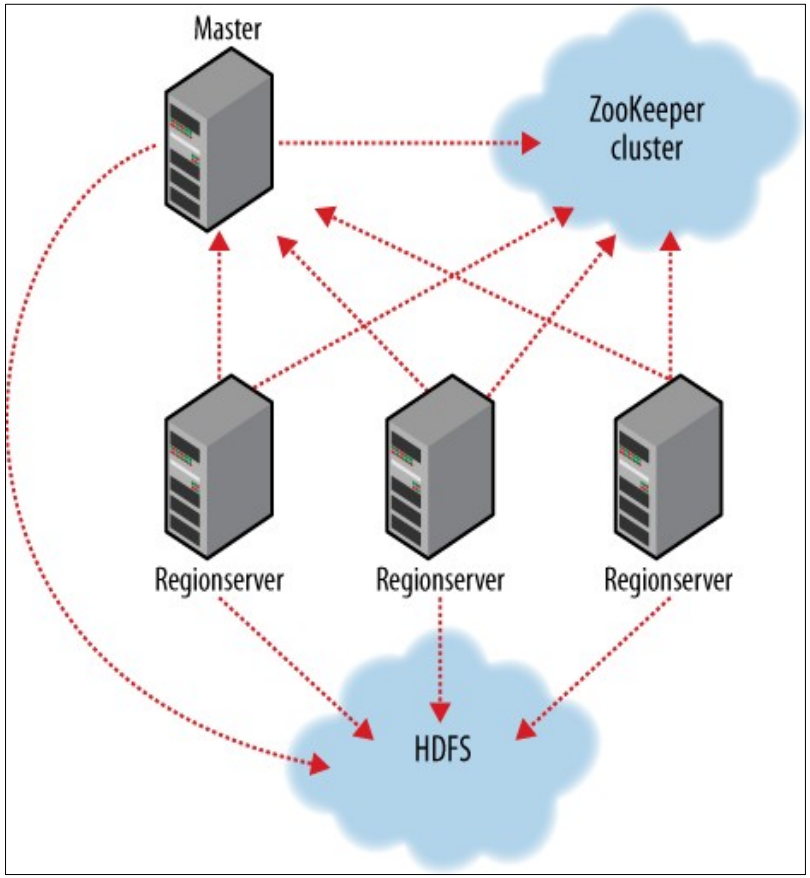
- **Region**
  - A subset of a table's rows, like horizontal range partitioning
  - Automatically done
- **RegionServer (many slaves)**
  - Manages data regions
  - Serves data for reads and writes (***using a log***)
- **Master**
  - Responsible for coordinating the slaves
  - Assigns regions, detects failures
  - Admin functions

BIG PICTURE


Client
Zookeeper
HMaster
HBase
HRegionServer
HRegion
HLog
Store
MemStore
StoreFile
HFile
DFS Client
Hadoop
DataNode

ZOOKEEPER

- HBase depends on ZooKeeper
- By default HBase manages the ZooKeeper instance
  - E.g., starts and stops ZooKeeper
- HMaster and HRegionServers register themselves with ZooKeeper



CREATING A TABLE

```
HBaseAdmin admin= new HBaseAdmin(config);
HColumnDescriptor []column;
column= new HColumnDescriptor[2];
column[0]=new HColumnDescriptor("columnFamily1:");
column[1]=new HColumnDescriptor("columnFamily2:");
HTableDescriptor desc= new
   HTableDescriptor(Bytes.toBytes("MyTable"));
desc.addFamily(column[0]);
desc.addFamily(column[1]);
admin.createTable(desc);
```

OPERATIONS ON REGIONS: GET()

- Given a key → return corresponding record
- For each value return the highest version

```
Get get = new Get(Bytes.toBytes("row1"));
Result r = htable.get(get);
5.8.1.2. Default Get Example ue(Bytes.toBytes("cf"), Bytes.toBytes("attr"));  // returns current version of value
```

- Can control the number of versions you want

```
Get get = new Get(Bytes.toBytes("row1"));
get.setMaxVersions(3);  // will return last 3 versions of row
Result r = htable.get(get);
byte[] b = r.getValue(Bytes.toBytes("cf"), Bytes.toBytes("attr"));  // returns current version of value
List<KeyValue> kv = r.getColumn(Bytes.toBytes("cf"), Bytes.toBytes("attr"));  // returns all versions of
```

OPERATIONS ON REGIONS: SCAN()

```
HTable htable = ...      // instantiate HTable

Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("cf"),Bytes.toBytes("attr"));
scan.setStartRow( Bytes.toBytes("row"));                   // start key is inclusive
scan.setStopRow( Bytes.toBytes("row" +  (char)0));  // stop key is exclusive
ResultScanner rs = htable.getScanner(scan);
try {
  for (Result r = rs.next(); r != null; r = rs.next()) {
  // process result...
} finally {
  rs.close();  // always close the ResultScanner!
}
```

GET()

Select value from table where key='com.apache.www' AND label='anchor:apache.com'

Row key	Time Stamp	Column "anchor:"	
"com.apache.www"	t12		
	t11		
	t10	"anchor:apache.com"	"APACHE"
"com.cnn.www"	t9	"anchor:cnnsi.com"	"CNN"
	t8	"anchor:my.look.ca"	"CNN.com"
	t6		
	t5		
	t3		

SCAN()

Select value from table where anchor='cnnsi.com'

Row key	Time Stamp	Column "anchor:"	
"com.apache.www"	t12		
	t11		
	t10	"anchor:apache.com"	"APACHE"
"com.cnn.www"	t9	"anchor:cnnsi.com"	"CNN"
	t8	"anchor:my.look.ca"	"CNN.com"
	t6		
	t5		
	t3		

OPERATIONS ON REGIONS: PUT()

- Insert a new record (with a new key), Or
- Insert a record for an existing key

Implicit version number (timestamp)

```
Put put = new Put(Bytes.toBytes(row));
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"), Bytes.toBytes( data));
htable.put(put);
```

Explicit version number

```
Put put = new Put( Bytes.toBytes(row));
long explicitTimeInMs = 555;  // just an example
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"), explicitTimeInMs, Bytes.toBytes(data));
htable.put(put);
```

OPERATIONS ON REGIONS: DELETE()

- Marking table cells as deleted
- **Multiple levels**
  - Can mark an entire column family as deleted
  - Can make all column families of a given row as deleted

- **All operations are logged by the RegionServers**
- **The log is flushed periodically**

HBASE: JOINS

- HBase does not support joins
- Can be done in the application layer
  - Using scan() and get() operations

ALTERING A TABLE

```
Configuration config = HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(conf);
String table = "myTable";

admin.disableTable(table);

HColumnDescriptor cf1 = ...;
admin.addColumn(table, cf1);      // adding new ColumnFamily
HColumnDescriptor cf2 = ...;
admin.modifyColumn(table, cf2);    // modifying existing ColumnFamily

admin.enableTable(table);
```

Disable the table before changing the schema

6.1. Schema Creation

Distributed Data Storage and Big Data

5.1 Distributed File Systems

5.1.1 Google File System (GFS):
Architecture, Operations, features and use cases

5.1.2 Hadoop Distributed File System (HDFS):
Architecture, Operations, features and use cases

5.2 NoSQL Databases:

5.2.1 Cassandra: Architecture, design principles,
Data Model and Query Language, features and use cases

5.2.2 MongoDB: Architecture, design principles,
Data Model and Query Language, features and use cases

5.3 Big Data Processing Frameworks

5.3.1 Hadoop: Key features and use cases

5.3.2 Spark : Key Features and use cases