HBase for Real-Time data
Session 1

Ronan Stokes(rstokes@cloudera.com)
Solutions Architect
Cloudera Inc.

Who am I

- Ronan Stokes, Solutions Architect, Professional services, Cloudera
  - Background in scalable data integration for unstructured and semi-structured data, semantic technologies, search …
- Previously
  - Development of distributed systems, data integration software, EAI tools, embedded systems …
  - First introduction to NOSQL – using to prepare semantic search engine corpuses for patent and news wire search

Real-time systems

- Real-time systems: Hardware and software subject to real time constraints
  - Operational deadlines from event to response
  - May be milliseconds or less, seconds, minutes, …
- Types
  - Hard: missing deadline means system failure
  - Firm: infrequent misses are tolerable but means degradation of quality of result. Results have zero value after deadline
  - Soft: usefulness of results degrades after deadline, degrading quality of service
  - Near real time : as soon as possible

Source : Wikipedia

Real-time systems - examples

- Hard
  - Anti-lock breaking
  - Drive by wire
  - X-ray dosing
- Firm
  - Trade execution
  - Level 2 quotes
- Soft
  - 15 minute quotes
- Near real time
  - User search for product availability

HBase and real-time

- Near real time data acquisition
- Near real time data access, retrieval or analytics
- Processing of real-time and near real-time data
  - Dashboards
  - Analytics
  - Soft or near real-time systems
- Exploratory analytics
  - Don't need to define schemas in advance

Real time / near real-time data

- Time series data
- Data from user driven events – web traffic logs, tweets
- SCADA data – data from industrial sensors or processes
  - Water level data – drives tsunami early warning system for US
  - Industrial production data
  - Oil and gas production
- Seismic data
- Weather data
- Just-in-time indexing
  - Only index items added in last time period X

Real-time and near real time data

- Often produced by time based processes
  - Data grows as time passes
  - Growth often occurs over time, not just dependent on business growth or production levels
- Often characterized by observation from location / site, system, subsystem, component at a point in time
  - Seismic data, weather information, user traffic …
  - Many of the same characteristics occur in log data
- Common challenges in time processing
  - Conversion to time slices
  - Computing time intervals

Common uses

- Producing time based aggregations
- Producing view across time slice of activity
- Acting as feed to dashboard
- Pattern discovery in activities

Hadoop and HBase

What is Apache Hadoop?

Apache Hadoop is an open source platform for data storage and processing that is…

- ✓ Scalable
- ✓ Fault tolerant
- ✓ Distributed

CORE HADOOP SYSTEM COMPONENTS

Brings Storage and Processing Together

- Scale-out architecture divides workloads across multiple nodes
- Flexible file system eliminates ETL bottlenecks

Flexibility - Store and Mine Any Type of Data

- Ask questions across structured and unstructured data that were previously impossible to ask or solve
- Schema "on read" vs "on write"
- Not bound by a single schema

Changes the Economics

- Can be deployed on industry standard hardware
- Open source
- Linear Scalability and Cost Model

Core Hadoop: HDFS

Self-healing, high bandwidth **clustered storage.**

HDFS breaks incoming files into blocks and stores them redundantly across the cluster.

How Files Are Stored

- Files are split into blocks
  - Each block is usually 64MB or 128MB
- Data is distributed across many machines at load time
  - Different blocks from the same file will be stored on different machines
  - This provides for efficient MapReduce processing (see later)
- Blocks are replicated across multiple machines, known as *DataNodes*
  - Default replication is three-fold
    - i.e., each block exists on three different machines
- A master node called the *NameNode* manages the metadata – block locations etc.

Core Hadoop: MapReduce

Distributed computing framework

Processes large jobs in parallel across many nodes and combines the results.

Some Hadoop database options

- Hive – associates metadata with data stored as files on HDFS
  - Supports SQL query and DML on underlying data
  - Allows leaving underlying file data unchanged or use of own file allocation
  - SQL queries executed as map reduce jobs
- Impala
  - Uses Hive metadata to execute queries over HDFS based data
  - Execution model is MPP providing much lower latency query execution
- HBase – NOSQL columnar key value store

HBase background

- Google produces “Google File System” – 2003
- Google - “MapReduce: Simplified Data Processing on Large Clusters”, 2003 - 2004
- Google – “Big Table: A Distributed Storage System for Structured Data”, 2006
- PowerSet building natural language indexer, 2006
- Hadoop and HBase, based on Google GFS and Big table papers, published 2007
- HBase used for many large high performance applications – twitter, yahoo, …

HBase overview

- Distributed, column-oriented data store built on top of HDFS
  - Designed to handle billions of rows and petabytes of data
  - Fast reads/writes
  - Columns may be added on the fly and store any kind of data
  - Strong consistency
  - Not a RDBMS: No joins, no indexes, no SQL
- Examples
  - **Search Engines**: Websites
  - **Oil and Gas**: Wells
  - **Retailers**: Customers
  - **Chemical companies**: Compounds

HBase basics

- Column oriented key-value store
- Allow store, retrieve and increment of value addressable by multi level key
  - Row-key, column family, 'column qualifier', timestamp => value
- Think of it as being one large table
- Columns do not have to be declared
- Each row can contain differing numbers of columns
- No advance schema required
- No relational language
- Writes are versioned
  - By default, versions are indicated by timestamps
  - But can specify explicit version or timestamp

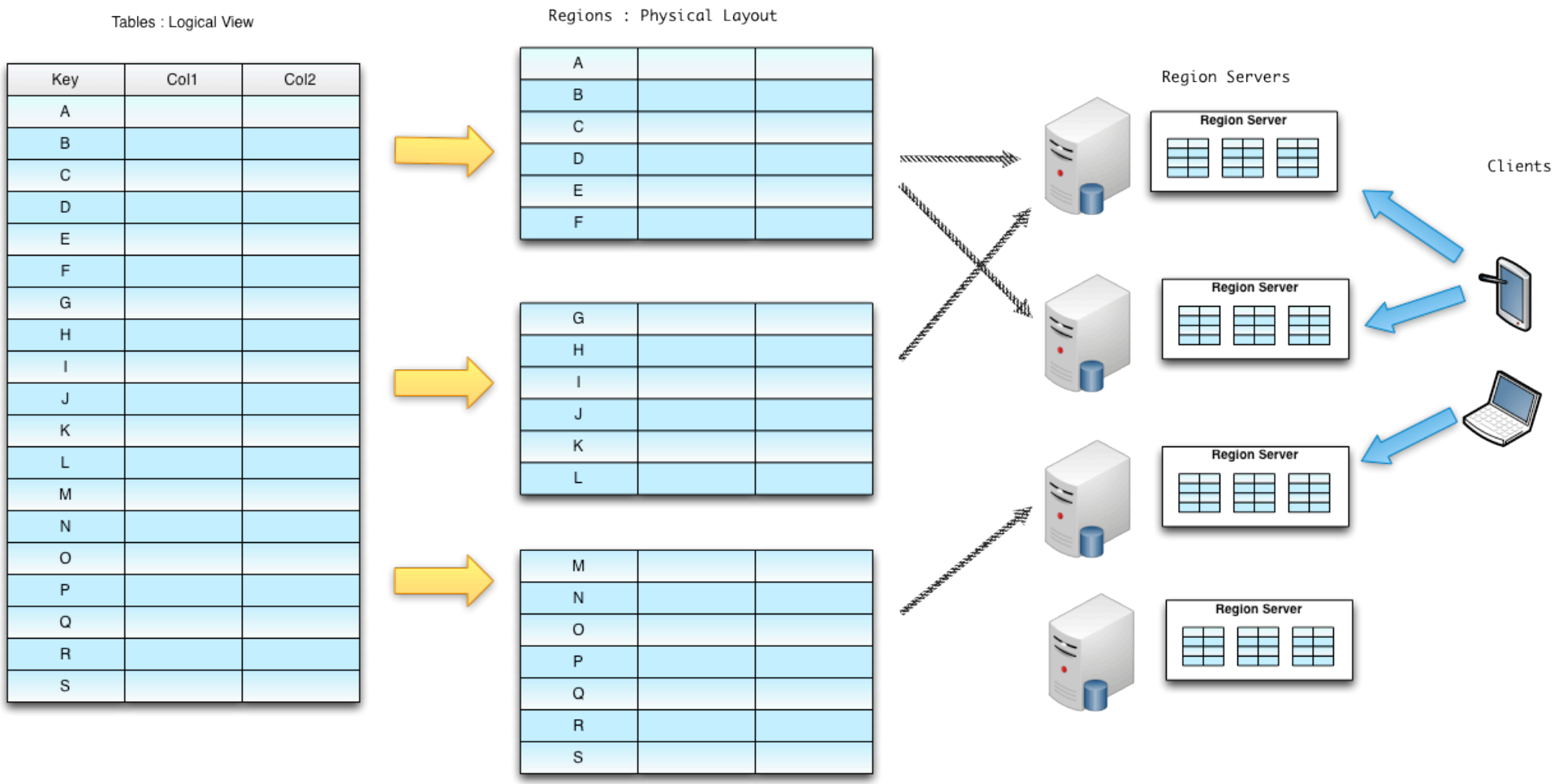
Storage API

- Java based API
- Create / delete tables and column families
- Get , Put, Increment column values
- Scan with filtering
- Single row transactions
- Atomic counters – increment is single atomic operation
- Run plugin 'coprocessor' code on server
  - Allows for implementation of 'triggers'
- No SQL
  - But gateways from Impala, Hive, other components

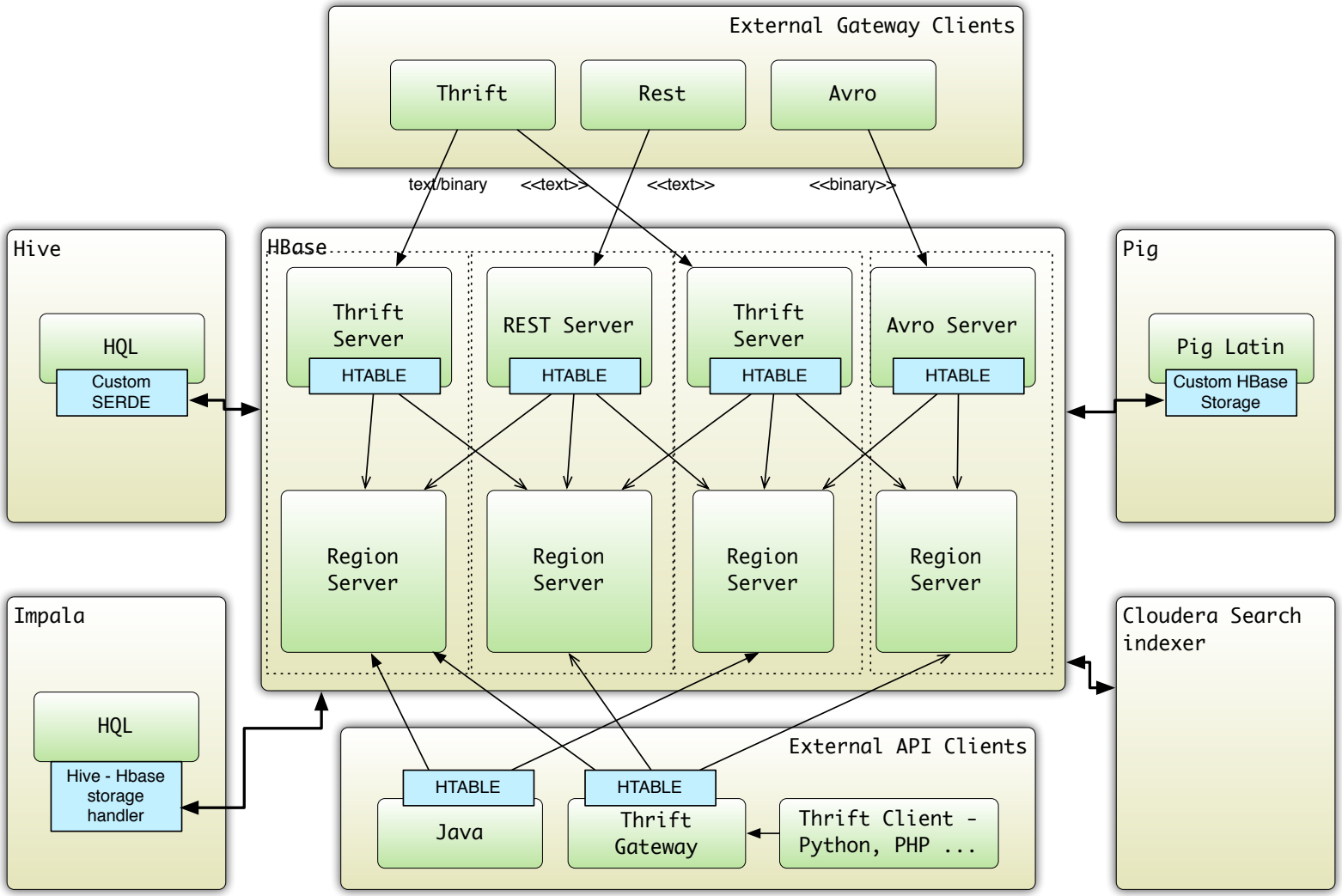
HBase API

- Simple operations
  - Put – write a value at (table, column family , location)
  - Get – read a value from (table, cf, location)
  - Scan – return rows / columns matching criteria
    - Start and end keys
    - Column names
    - Values
    - More advanced filtering
  - Incr : increment column value by 1 – null is considered zero
- Can partition columns by column family
- Column name aka ‘qualifiers’ can be arbitrary bytes stream
  - But Hue browser requires text qualifiers to display correctly
- Value is versioned
  - X versions retained where x is configurable
- Deletes deferred until compaction
  - ‘Tombstoned’ until deletion

Automatic Sharding

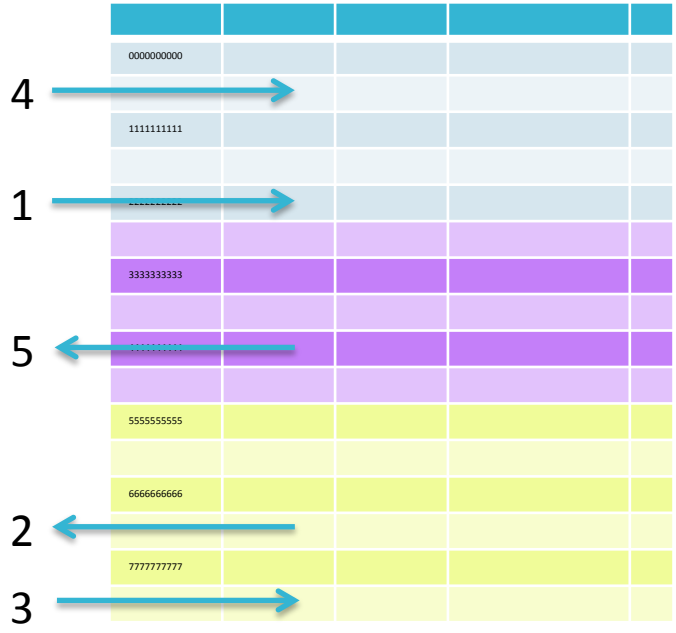


Integration options



HBase provides Low-latency Random Access

- **Writes**:
  - 1-3ms, 1k-20k writes/sec per node
- **Reads:**
  - 0-3ms cached, 10-30ms disk
  - 10k-40k reads / second / node from cache
- **Cell size:**
  - 0B-3MB
- Read, write, and insert data anywhere in the table



Hadoop and Databases

Databases
"Schema-on-Write"

- Schema must be created before any data can be loaded
- An explicit load operation has to take place which transforms data to DB internal structure
- New columns must be added explicitly before new data for such columns can be loaded into the database

Hadoop
"Schema-on-Read"

- Data is simply copied to the file store, no transformation is needed
- A SerDe (Serializer/Deserlizer) is applied during read time to extract the required columns (late binding)
- New data can start flowing anytime and will appear retroactively once the SerDe is updated to parse it

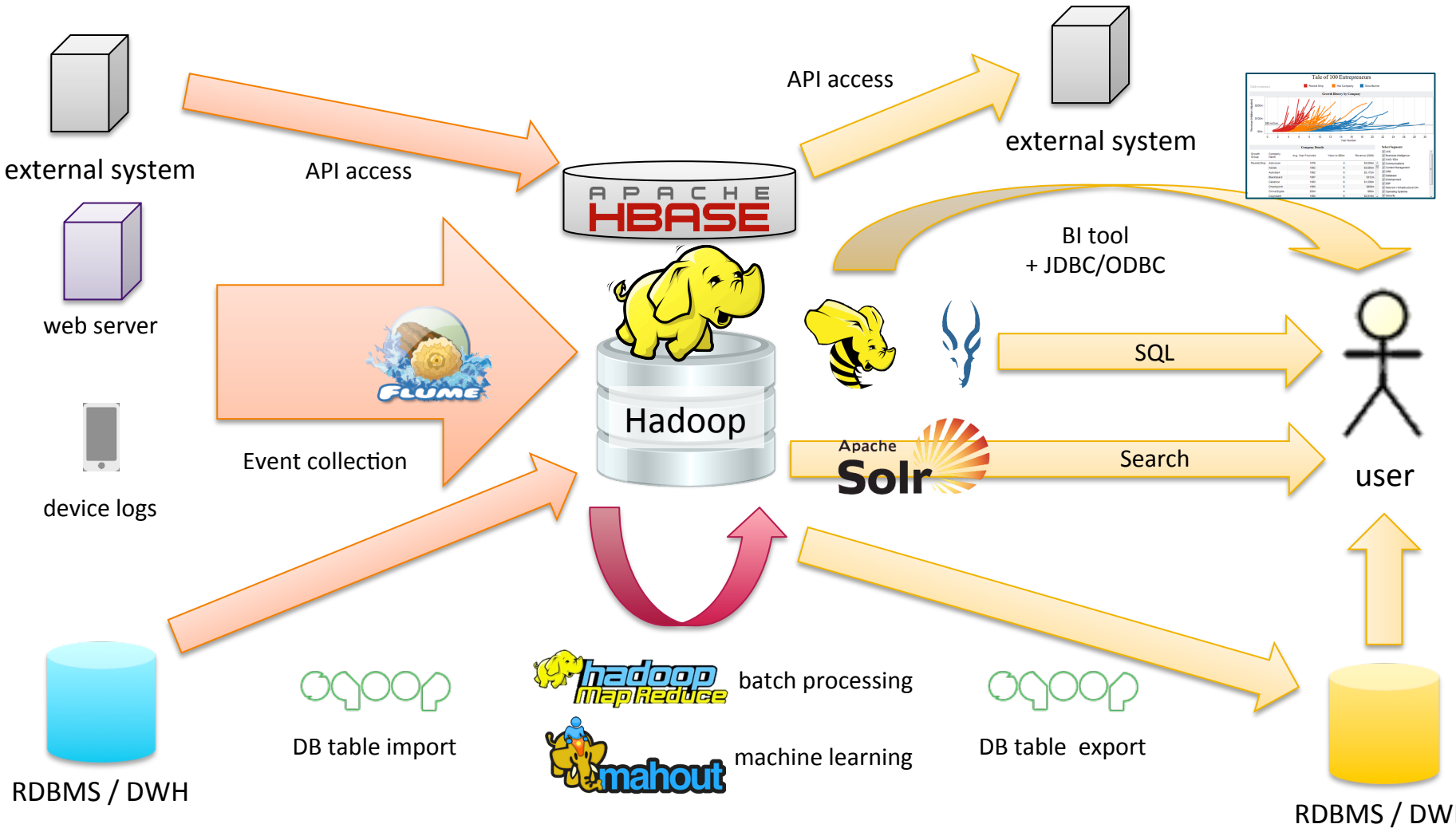
1) Reads are Fast
2) Standards and Governance

1) Loads are Fast
2) Flexibility and Agility

HBase vs. Relational Databases

	RDBMS	HBase
Data layout	Row or column-oriented	Column Family-oriented
Transactions	Yes	Single Row Only
Query Language	SQL	get/put/scan
Security	Yes	Yes, currently in beta
Indexes	Yes	Row Key Only
Max Data Size	TBs	PB+
Read/write throughput limits	1000s queries/second	Millions of queries/ second

Components of Hadoop system



HBase

HBase without HBase

- Want read/write access to data in HDFS
  - Can only write new data not update existing data
  - Implement versioning scheme so each row has version indicator that effectively updates existing values
  - Periodically compact to remove old values
    - Can't update existing rows to mark data as tombstoned
    - Would need to write to copy of table / store eliminating earlier versions of data
  - Exchange tables / partitions to remove old data
    - Would need to exchange partitions or table references so that old data is no longer retrieved in query
- HBase does all of this for you …
  - + in-memory cache and more

What is HBase

HBase is an
open source, **distributed**,
sorted map datastore
modeled after Google's BigTable

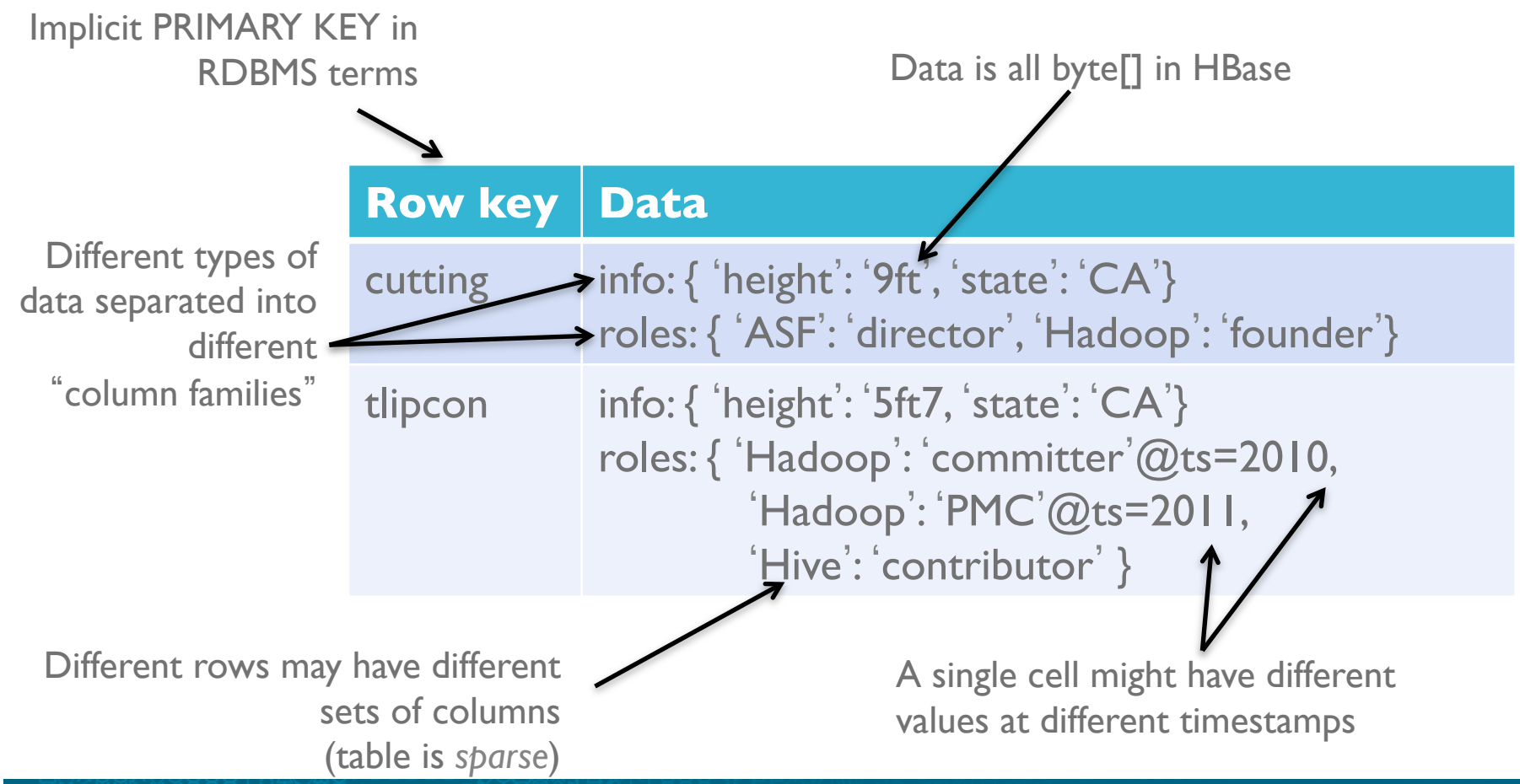
Open Source

- Apache 2.0 License
- Committers and contributors from diverse organizations
  - Cloudera, Facebook, StumbleUpon, Trend Micro, etc.

Sorted Map Datastore

- Not a relational database (very light "schema")
- Tables consist of rows, each of which has a primary key (row key)
- Each row may have any number of columns, like a `Map<byte[], byte[]>`
- Rows are stored in sorted order

Sorted Map Datastore
(logical view as "records")



Sorted Map Datastore
(physical view as "cells")

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

Row key	Column key	Timestamp	Cell value
cutting	info:height	1273516197868	9ft
cutting	info:state	1273871824184	CA
cutting	roles:ASF	1273871823022	director
cutting	roles:Hadoop	1273746289103	founder
tlipcon	info:height	1273878447049	5ft7
tlipcon	info:state	1273616297446	CA
tlipcon	roles:Hadoop	1300062064923	PMC
tlipcon	roles:Hadoop	1293388212294	Committer
tlipcon	roles:Hive	1273616297446	contributor

Milliseconds since unix epoch

Conceptual data model

- Tables comprised of column families of columns
- Column references : <column family name>:<column>
- Sparse storage – no need to define column before use

Column Families

- Different sets of columns may have different properties and access patterns
- Configurable by column family:
  - Compression (none, gzip, LZO)
  - Version retention policies
  - Cache priority
- CFs stored separately on disk: access one without wasting IO on the other.

HBase Shell

- Starting the shell
  - $ ./bin/hbase shell
- Running a script
  - ./bin/hbase shell PATH_TO_HBASE_SCRIPT
  - ./bin/hbase org.jruby.Main PATH_TO_SCRIPT
- Commands
  - Create, put, scan, disable, drop, count
  - Administration – compact, major_compact …

Physical Model

- Tables stored on column family basis
- Empty cells not stored
- Values stored as tuple – (row key, cf, column qualifier, timestamp, value)
- Tables must be declared but not columns
- Column families must be declared before use
- Row keys are un-interpreted bytes
  - Lexicographically ordered with lowest occuring first in table

Column families

- Column family prefixes must be printable characters
- Column names ('qualifiers') can be arbitrary stream of bytes
- All column family members stored together on file system
  - Tunings and storage specifications at column family level
- Compression may be specified at column family level
- Time to live may be specified on a column family level
  - Automatically removes data during next compaction after TTL expires

Cells

- Logically {rowid, version_indicator, value}
- Content is uninterpreted bytes
  - Can have unbounded set of cells differing only by version
- FAQ
  - What about multiple writes to same version?
  - What about writing cells in non-increasing version order?
- Gets and Scans
  - Get latest version of cell (largest version)
  - Can retrieve specific versions
- Put
  - Auto increments version
  - Can specify explicit version even replacing existing one
  - Beware of deletes – will shadow older versions even if written later
  - If put and delete happen in same millisecond, major compaction will remove (as they have same timestamp)

Operations

- Main operations
  - Get
  - Put
  - Scan – iterates over group of rows
  - Delete
    - Does not physically delete
    - Adds tombstone markers
    - Removed on major compactions
    - Delete will remove all cells with timestamp <= tombstone timestamp
  - Sorting – implicit
  - Joins – via application or MR job only

Sort order

- Data stored in order of increasing rowkey
- All operations return data in sorted order
  - Sort order is row, column family, column qualifier, timestamp (in descending order) so newest are returned first
  - Data is stored in increasing lexical order of row key and column qualifier
  - Can skip rows and columns with filtering, start and stop row keys

Metadata

- HBase does not store metadata for table
  - Need to scan entire table to get all column names
- Using Hive or Impala can define schema for tables when used in Hive / Impala queries
  - But metadata is only imposed when reading via Hive or Impala
  - Access directly through HBase APIs does not use Hive / Impala metadata

Acid transactions

- Simple Acid model
  - Puts to same row occur in same transaction
  - HBase retrieves next highest txn no (“WriteNumber”)
  - Each written kv pair tagged with write number
  - Reads use last commited txn number (“Readpoint”)
- Write transaction
  - Writes lock row to prevent concurrent modification
  - Retrieve write number
  - Apply changes to write ahead log
  - Apply changes to memstore
  - Commit txn (roll readpoint forward to new write number)
  - Unlock rows

Acid transactions - 2

- Read txn
  - Readpoint is called “memstore timestamp”
  - Separate from application visible timestamp of cell
- Read transaction
  - Open scanner
  - Get current readpoint
  - filter all scanned KeyValues with memstore timestamp > the readpoint
  - close the scanner (this is initiated by the client)
- HBase commits all transactions serially
- Does not guarantee consistency between regions
- http://hadoop-hbase.blogspot.com/2012/03/acid-in-hbase.html

HBase API

- get(row)
- put(row, Map<column, value>)
- scan(key range, filter)
- increment(row, columns)
- … (checkAndPut, delete, etc…)
- MapReduce/Hive

HBase API – more details

- Types – all types read and written are bytes
  - Application interprets meaning
- checkAndPut
  - Checks for value and only writes if tested value matches
- getRowOrBefore
  - Find row with key less than provided key
- compareAndDelete
- Batched puts, deletes, gets
- Locks on rows
  - Use sparingly – has deadlock potential

HBase API - Filters

- Filters run on region servers when performing scans
  - Specify rows , columns to skip or retrieve
  - Filter by column value, row key prefix, row key value, regex of value ... Many more
  - Filter by time range of timestamps allows for easy isolation of recent updates
  - Combine with AND, OR, …
- Closest thing to query language
  - Filter by
    - "(PrefixFilter('user1') AND PageSize(5))"
  - Returns 5 rows having key beginning with 'user1'

HBase coprocessors

- CoProcessors
  - Observer – Allow for executing of user defined code to be done in a number of areas while doing a put/get/delete

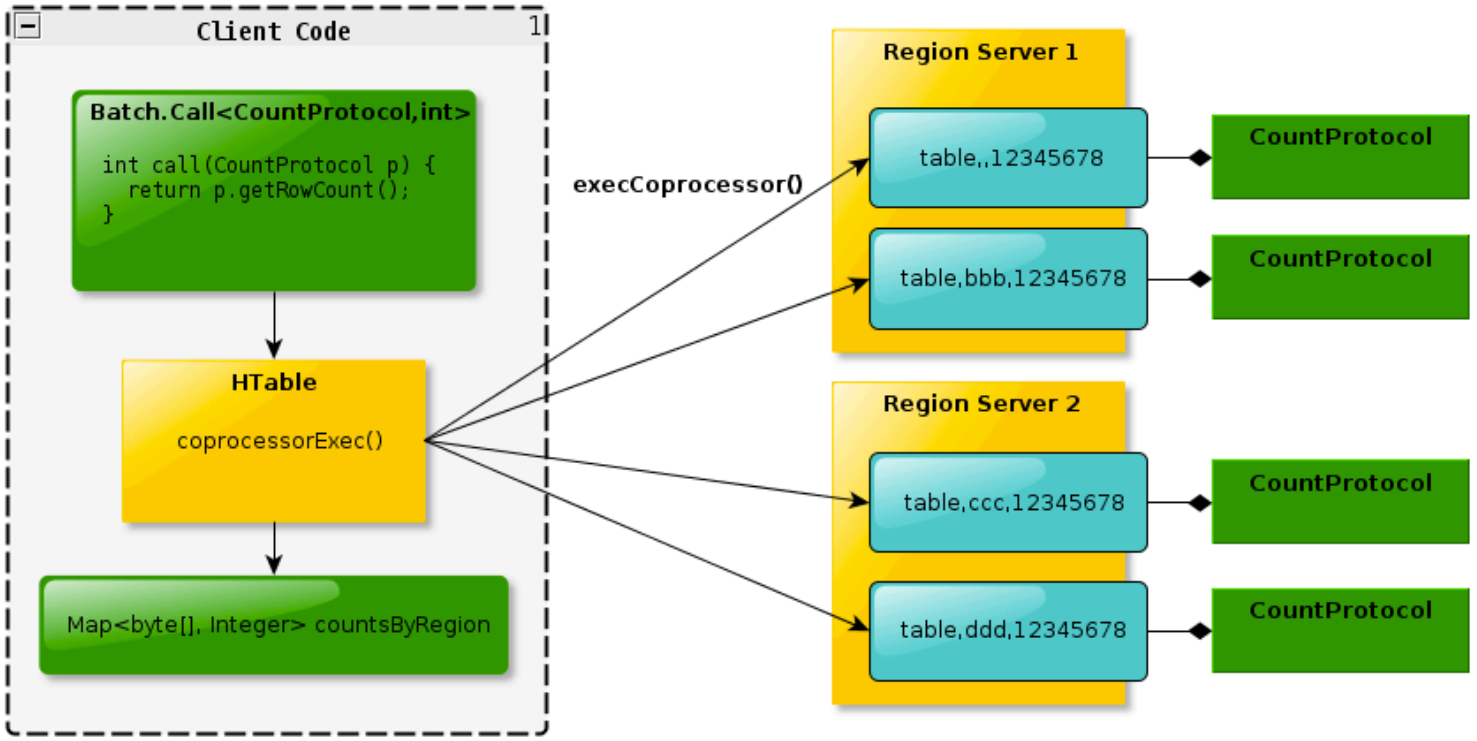
```
// Sample access-control coprocessor. It utilizes RegionObserver
// and intercept preXXX() method to check user privilege for the given table
// and column family.
public class AccessControlCoprocessor extends BaseRegionObserver {
  @Override
  public void preGet(final ObserverContext<RegionCoprocessorEnvironment> c,
final Get get, final List<KeyValue> result) throws IOException
      throws IOException {

    // check permissions..
    if (!permissionGranted())  {
        throw new AccessDeniedException("User is not allowed to access.");
    }
  }

  // override prePut(), preDelete(), etc.
}
```

HBase coprocessors - 2

- CoProcessors
  - Endpoint – Allow for the aggregation of a specific set of rows and for other calculations to be done without MapReduce



Accessing HBase

- Java API (thick client)
- REST/HTTP
- Apache Thrift (any language – Python, PHP etc.)
- Impala/Hive/Pig for analytics
  - Impala gives best performance – near real-time
  - Hive & Pig for batch
- Client apps can access running within MR or as standalone client

HBase architecture

HBase Terms

Region

- A subset of a table's rows, like a range partition
- Automatically sharded

RegionServer (slave)

Serves data for reads and writes

Master

Responsible for coordinating the slaves

Assigns regions, detects failures of Region Servers, controls some admin functions

Write Ahead Log

Records writes before writing to memstore

Memstore

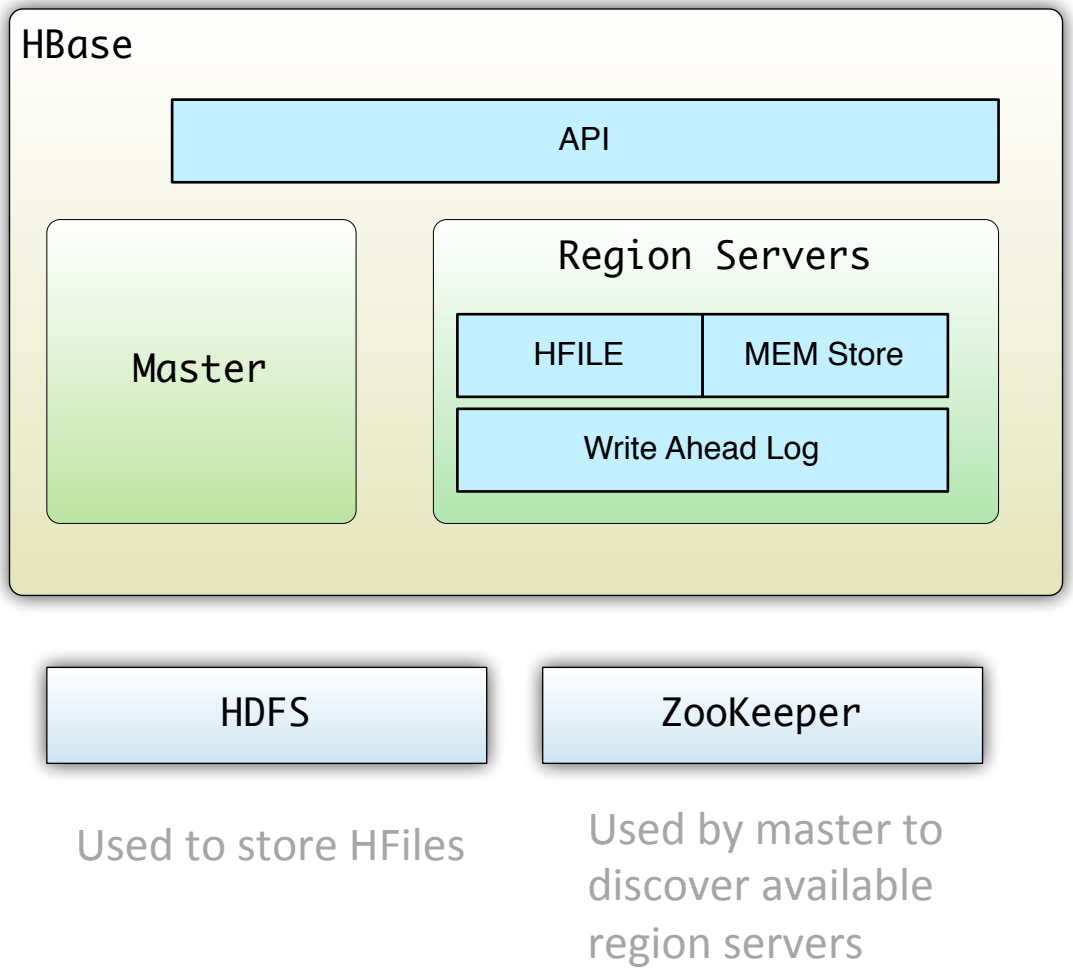
In memory cache of edits

Storefile

Persisted edits

HBase Architecture

Master monitors HBase regions and rebalances regions as necessary



Used to store HFiles

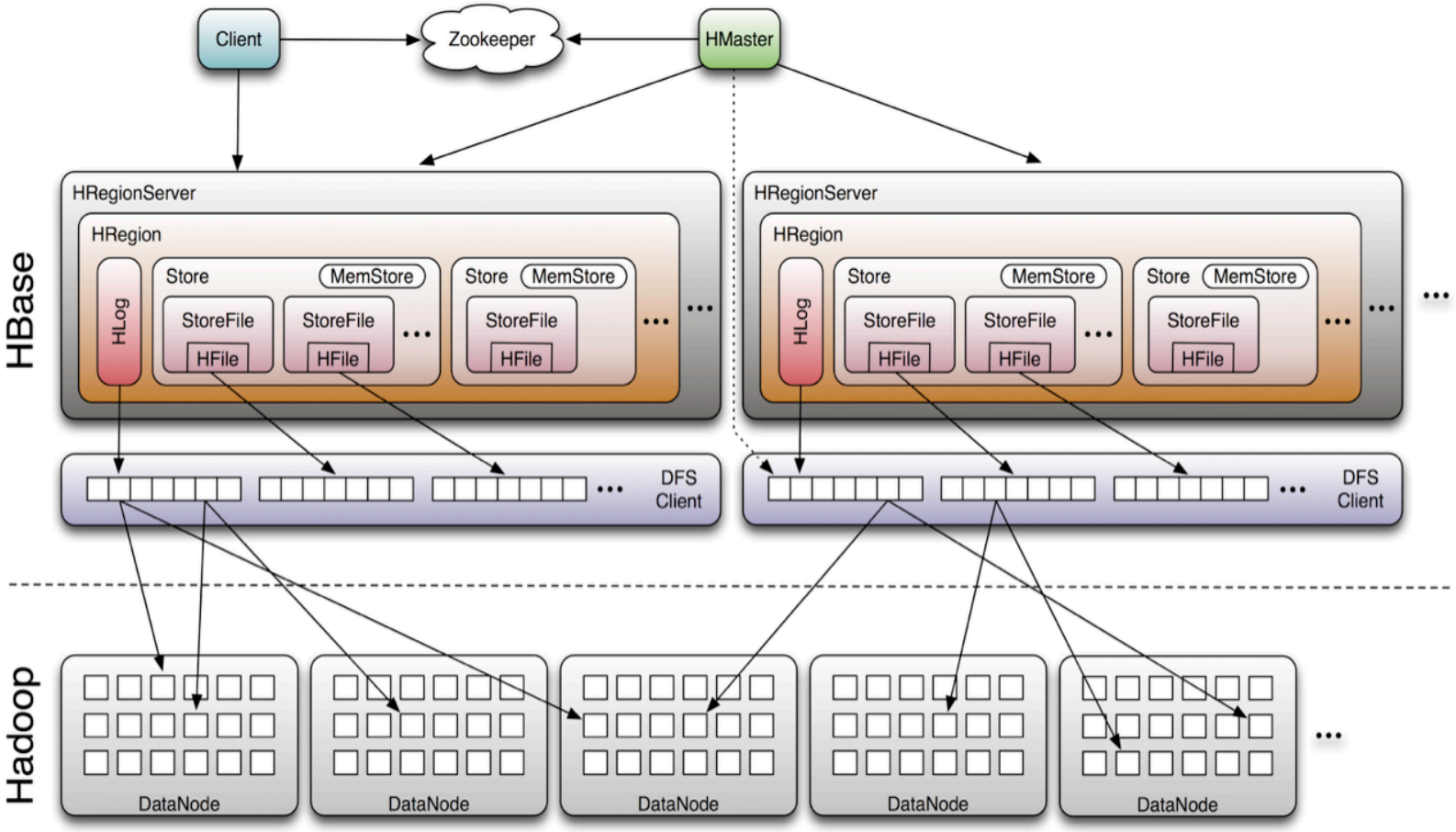
Used by master to discover available region servers

Memstore is in-memory cache of edits

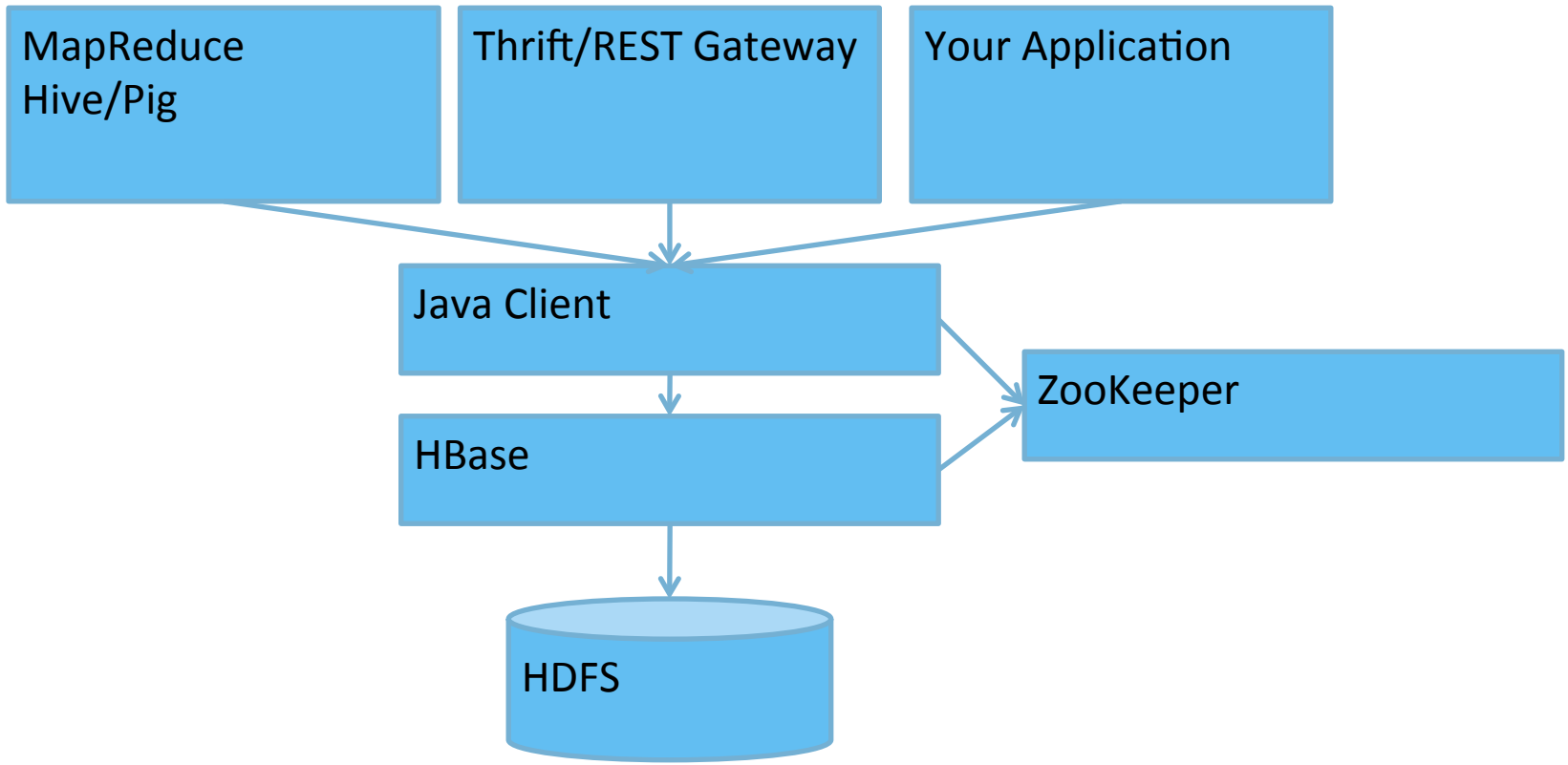
HFile is persistent store of data aka Store File

WAL – write ahead log is log of edits for recovery purposes

HBase architecture



High Level Architecture



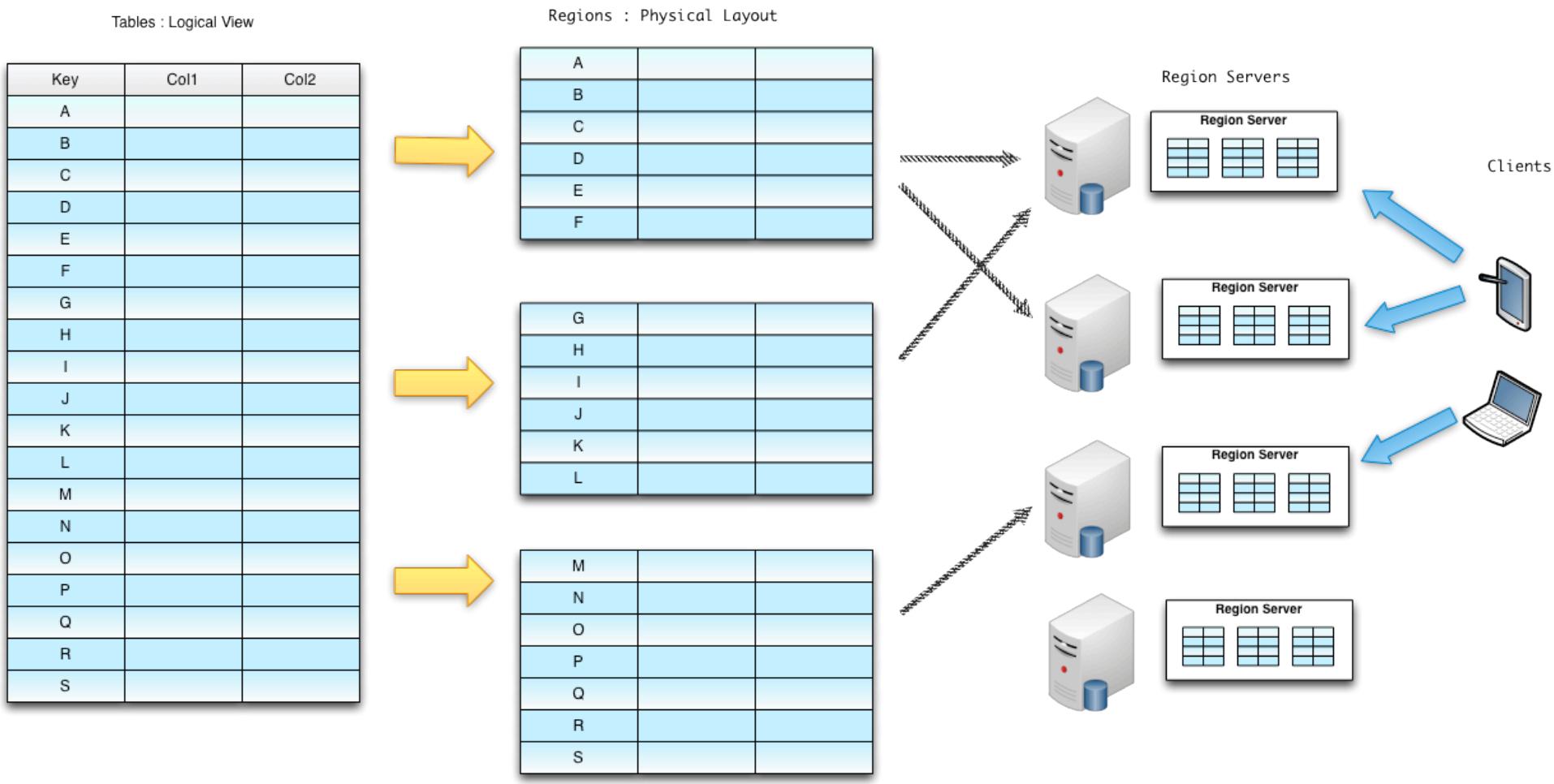
Logical storage model

- HBase Logical Data Storage Structures
  - **Table** – A grouping of column families
  - **Column Families** – A grouping of columns which will be kept locally to each other
  - **Column** – Made up on column name along with a list of values which are stored by timestamp

Distribution model

- Unit of scalability and load balancing is Region
  - Contiguous groups of rows stored together
  - Region is dynamically split when becomes too large
  - Alternatively merged to reduce number and storage requirements
- Initially one region for table
  - Split when table size grows beyond configurable limit
  - Split occurs at middle key
- Splitting is fast
  - Read from original storage file until compaction rewrites store
- We can think of this as automatic sharding

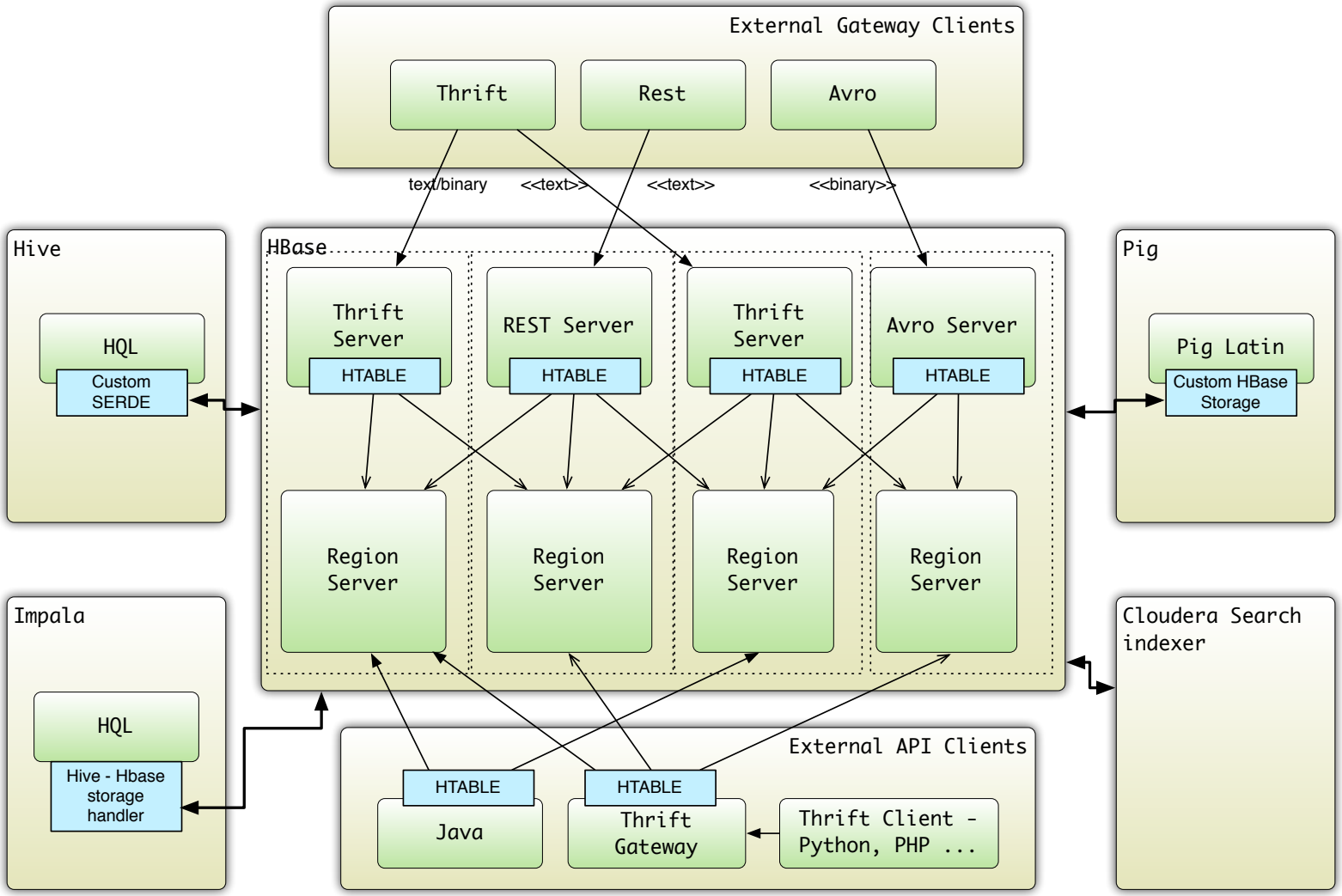
Automatic Sharding



Fold, store, shift

- Physical representation does not match logical representation
- Each cell is stored as separate row
  - Stored as row key, column family name, column qualifier, version timestamp, value
  - Folds columns into row per column
  - Multiple versions stored as separate rows
  - Stored in order of row key, column qualifier

Integration options



Implementation

- Data stored in HFILEs on HDFS – persistent immutable map from key to value
  - Internally sequences of blocks with block index at end
  - Index is loaded and kept in memory
  - Block index enables lookup with single seek
  - Default block size Is 64k but can change
  - Block is unit read during scan
- Each memstore flush creates new HFile

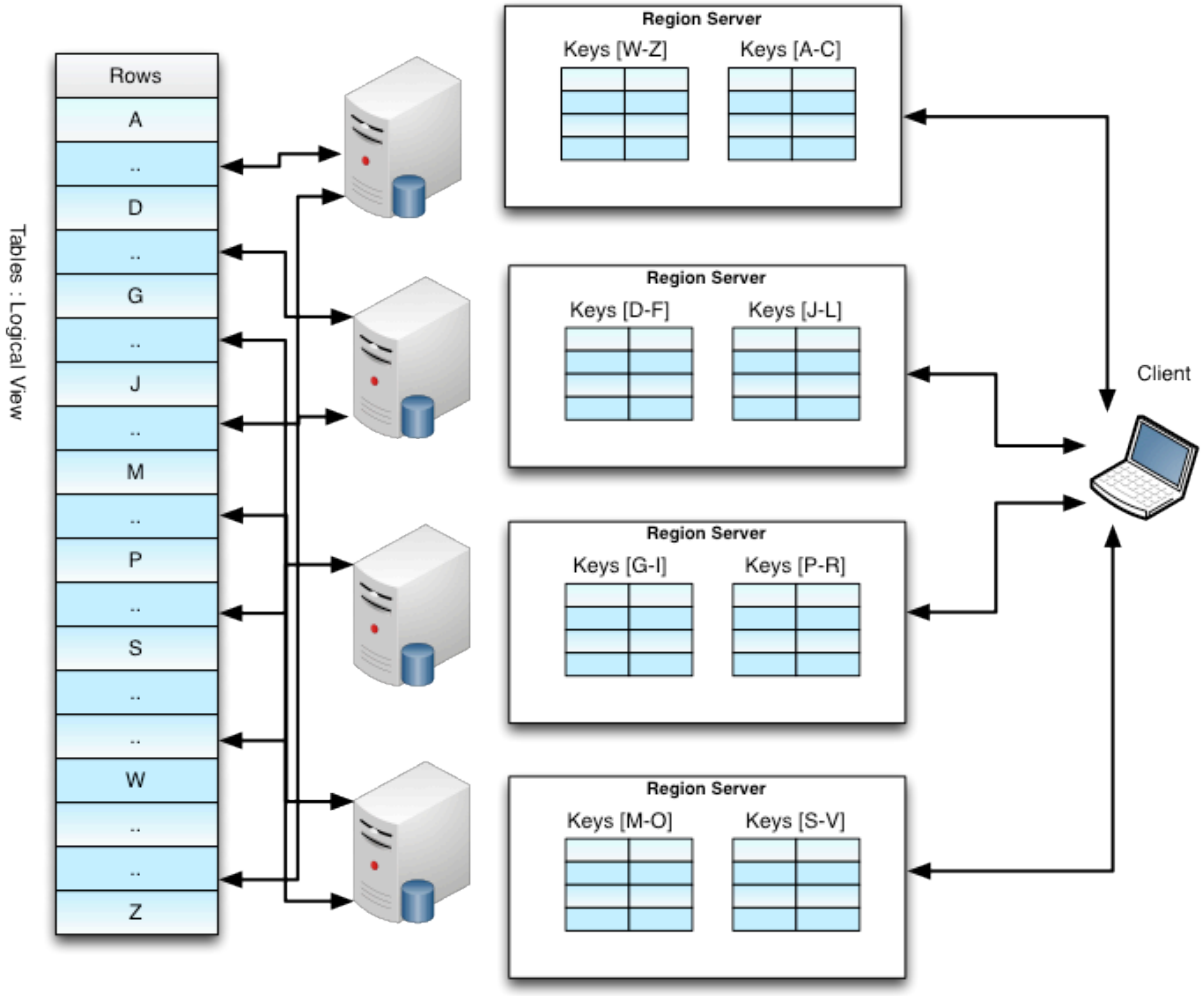
Splits

- When regions exceeds configured maximum size
  - System will split at middle key of range
- Alternatively may be merged if small regions can be combined
- Can define splits in advance:
  - create ’t1', 'cf1', {SPLITS => [’A',’G',’M',’S']}
  - Defines splits for key ranges [null – ‘A’], [‘A’-’F’], [‘G’-’L’], [’M’-’R’], [‘S’ to highest key]
- Other options to specify

Other options

- Specify use of block cache to cache blocks in memory
- Can specify that table should have priority for memory retention in block cache
- Can specify that column family should use Bloom filters
  - Optimizes access through use of bloom filter
  - Can quickly determine if row-key is not in table
- Can customize the splitting algorithm
- Can manually split, move regions

Key distribution



Write operations

- Put writes to commit log, AKA the "write ahead log" (WAL)
  - Then stored in memstore
  - When memstore exceeds configured size, flush writes memstore as new Hfile to HDFS
  - After flush, commit logs discarded up to last unflushed modification
  - While flushing, region still serves requests via multiple memstores
- Deletes are written as new tombstone marker
- Minor compactions merge Hfiles into smaller number of files
  - Relatively low cost operation
- Major compactions merge Hfiles into single Hfile removing any deleted items
  - Costly operation

Read operations

- Will read from merge of memstores and on disk store files
- WAL is never used during retrieval
  - Only used for recovery
  - Can optimize operations where write failures may be tolerated by disabling write to WAL programatically

End of session 1