

# **APACHE HBASE**

## **HADOOP NoSQL DATABASE**

Bin Jiang

04/01/2017

# Objective

Participants will learn about

- HBase distributed database system and its features
- Client interface supported by HBase
- Hbase table design
- Hbase Interface Shell

# What's HBase

- HBase is a column-oriented, sorted map database management system written in Java that runs on top of HDFS
- Designed for hosting large tables of sizes in the petabyte range, consisting of billions of rows and millions of columns
- Main use cases are an efficient handling of Big Data stored in the form of sparse data sets
- HBase supports fast random reads and writes in real time meeting low-latency requirements of online systems
- Integrates with MapReduce
- Support direct data querying using Java and other programming languages

# HBase Design

- HBase is modeled after Google's Bigtable system
- Just as Bigtable leverages the distributed data storage provided by the Google File System, Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS.
- Hbase's architecture is based on the master/salve design pattern
- Hbase system has a Master node that manages the cluster and Region Servers (slaves) to store portions of the databases (data is horizontally partitioned by rows)

## Notes:

The single master node design of HBase makes it a system with a single point of failure similar to HDFS's NameNode. Such "weak" points of open-source systems are usually "hardened" in enterprise deployments by consulting companies specializing in such systems or by large vendors that use open-source projects in their offerings

# HBase Features

- Linear scalability with automatic and configurable sharding (partitioning) of tables
- Strictly consistent reads and writes
  - Hbase is not an “eventually consistent” datastore as most NoSQL System – it is a strongly consistent data store
  - Strong (high) data consistency is ensured by routing all write operations for a key through the RegionServer made responsible for that key (it simplifies the system design and avoids last-write-wins scenarios)
    - ✓ If that RegionServer is lost, the entire key range owned by that RegionServer is made unavailable pending data recovery
- High Availability properties with automatic fail-over between Region Servers are reviewed a bit later
- HBase achieves operational efficiencies through data compression, in-memory data processing and some other features originally published in Google’s Bigtable paper in 2006
- Front cache for real-time queries



# HBase Concepts

- Namespace
- Table
- Row
- Column Family
- Cells (Row+Column+Version)
- Data Model Operation (Get, Put, Scan, Delete)
- Versions
- Sort Order
- Column Metadata
- Join
- ACID

# HBase Concepts – Data Model

- Table
- Row
- Column Family
- Column
- Column Qualifier
- Cell
- Timestamp

# HBase Concepts - KeyValue

The KeyValue format inside a byte array is:

- keylength
- valuelength
- key
- value



# HBase Concepts - KeyValue

The Key is further decomposed as:

- rowlength
- row (i.e., the rowkey)
- columnfamilylength
- columnfamily
- columnqualifier
- timestamp
- keytype (e.g., Put, Delete, DeleteColumn, DeleteFamily)

# HBase Concepts – Sort Order

All data model operations HBase return data in sorted order. First by row, then by ColumnFamily, followed by column qualifier, and finally timestamp (sorted in reverse, so newest records are returned first)

# HBase Concepts – Join

- There is no join in HBase similar to RDBS
- Two Strategies:
  - denormalizing the data upon writing to HBase
  - have lookup tables and do the join between HBase tables in your application or MapReduce code

# HBase Concepts – Secondary Index

- Filter Query
- Periodic-Update Secondary Index
- Dual-Write Secondary Index
- Summary Tables
- Coprocessor Secondary Index
- Search Engine

# Table Schema Rules Of Thumb

- Aim to have regions sized between 10 and 50 GB
- Aim to have cells no larger than 10 MB, or 50 MB if you use mob
- Consider storing your cell data in HDFS and store a pointer to the data in HBase if have cells larger than 10 MB, or 50 MB if you use mob
- A typical schema has between 1 and 3 column families per table
- HBase tables should not be designed to mimic RDBMS tables
- Around 50-100 regions is a good number for a table with 1 or 2 column families
- Keep your column family names as short as possible

# HBase Concepts – Data Type

- Anything that can be converted to an array of bytes can be stored as a value
- There is size limitation



# HBase Architecture

- Master
- Region Server
- Region
- Zookeeper
- HDFS

# HBase Architecture – Data Storage

- HLog - The write-ahead log file, also known as WAL
- HFile - The real data storage file
- hbase:meta - System-defined catalog table and keeps the list of all the regions for user-defined tables
- -ROOT- and .META. - Two old catalog tables

# Blocks

StoreFiles are composed of blocks. The blocksize is configured on a per-ColumnFamily basis.

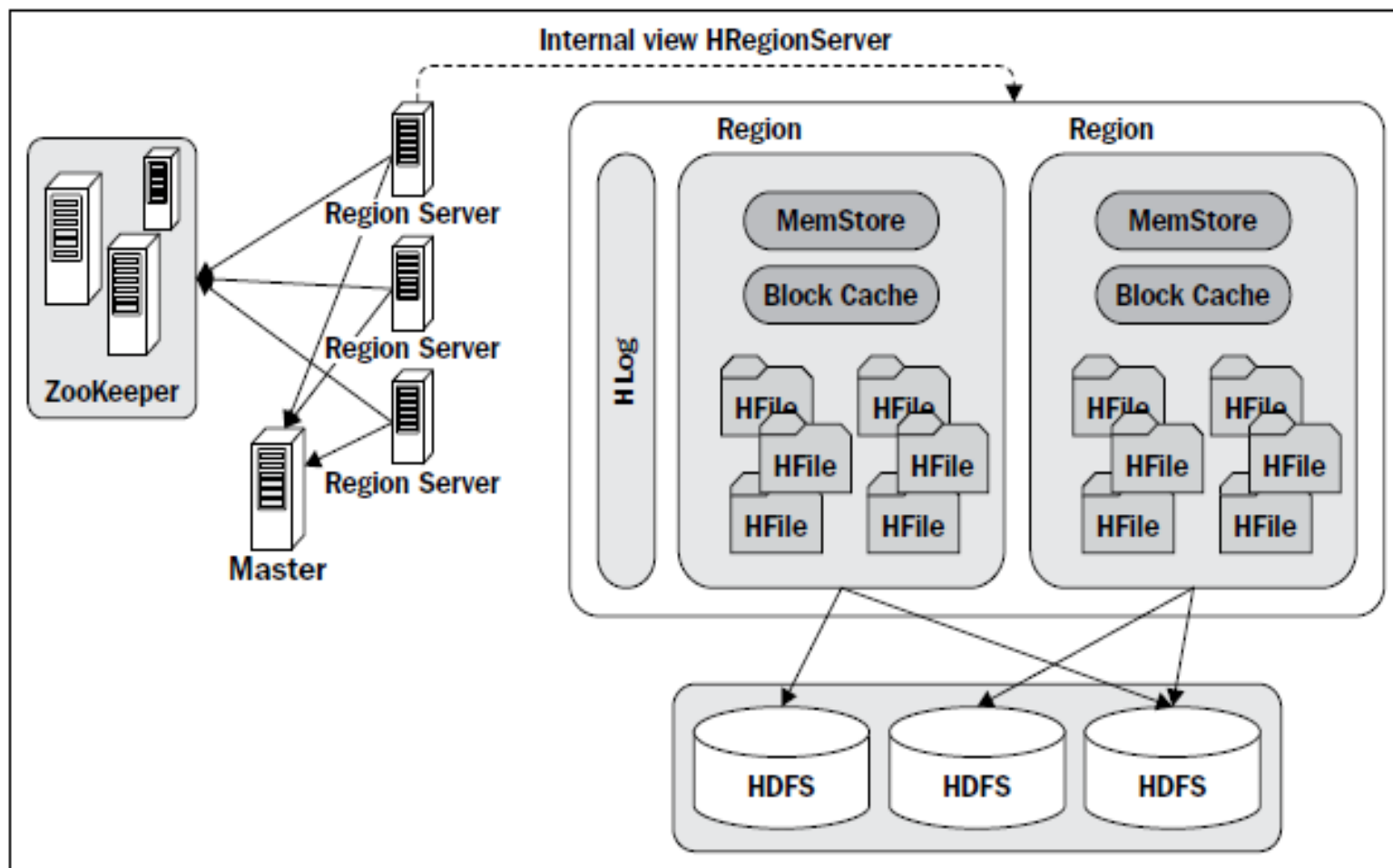
Compression happens at the block level within StoreFiles. For more information on compression

# Catalog Tables

## hbase:meta

- Key
  - Region key of the format ([table],[region start key],[region id])
- Values
  - info:regioninfo (serialized HRegionInfo instance for this region)
  - info:server (server:port of the RegionServer containing this region)
  - info:serverstartcode (start-time of the RegionServer process containing this region)

# HBase Architecture – Data Storage



# HBase Architecture – Block Cache

- Created at the time of the region server startup
- LruBlockCache, SlabCache, or BucketCache
- Multilevel Caching - LruBlockCache L1, Others are L2
- LruBlockCache - JVM Heap
- SlabCache, or BucketCache - Non JVM Heap



# HBase Architecture – HLog

- Write-ahead log
- HLogKey - Actual data and sequence numbers
- Also written in MemStore
- Writing data directly into MemStore is dangerous
- Flush command can be used to write the in-memory (memstore) data to the store files
- KeyValue instance
- log writer's sync() - Also writing the WAL to the replication servers
- LogRoller - `hbase.regionserver.logroll.multiplier`

# The WAL and MemStore

- When HBase writes data to a table (the write path), it takes extra effort to ensure data durability by writing data into two places:
  - ❑ The Write-Ahead Log (WAL), a.k.a the HLog (one per server), and
  - ❑ The MemStore (a Java heap-based write buffer, one per column family)
- The write operation to the table is considered complete when writes to both places (WAL and MemStore) succeed

## Notes:

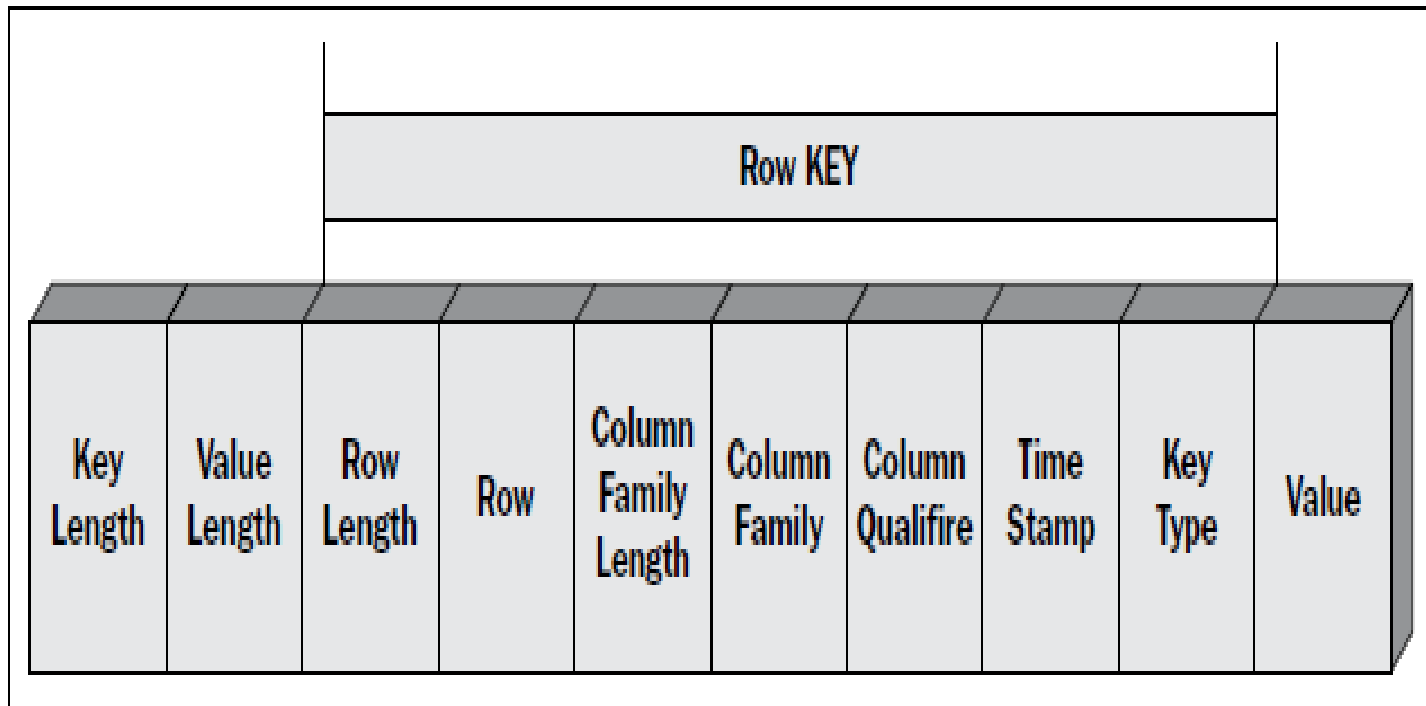
The MemStore is an area on the Java heap which acts as a write buffer where HBase accumulates data before flushing it to the disk.

Each Hbase server in the cluster has its own WAL file (physical file) which is shared by all tables housed on the server

You can speed up the write operation (it is a blocked/synchronous operation) at the expense of persistence reliability by switching off writes to the WAL using Java API as follows:

```
Put p = new Put();  
p.setWriteToWAL(false);
```

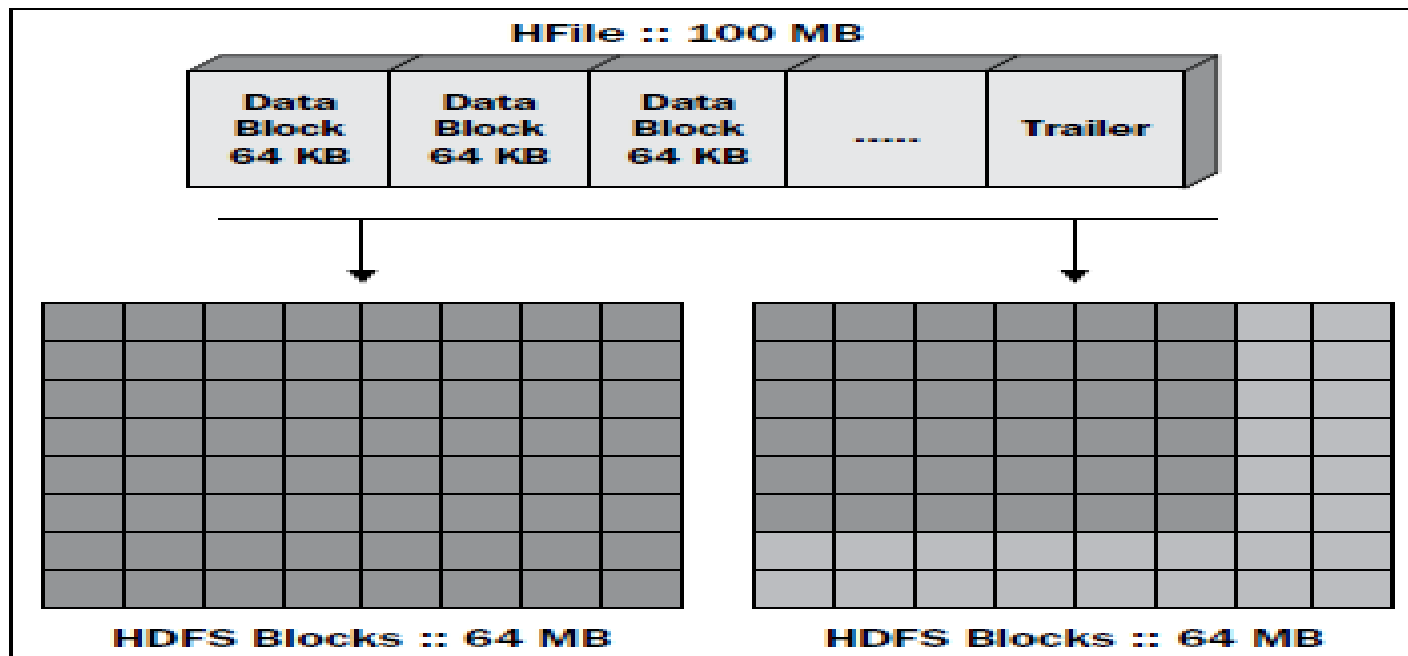
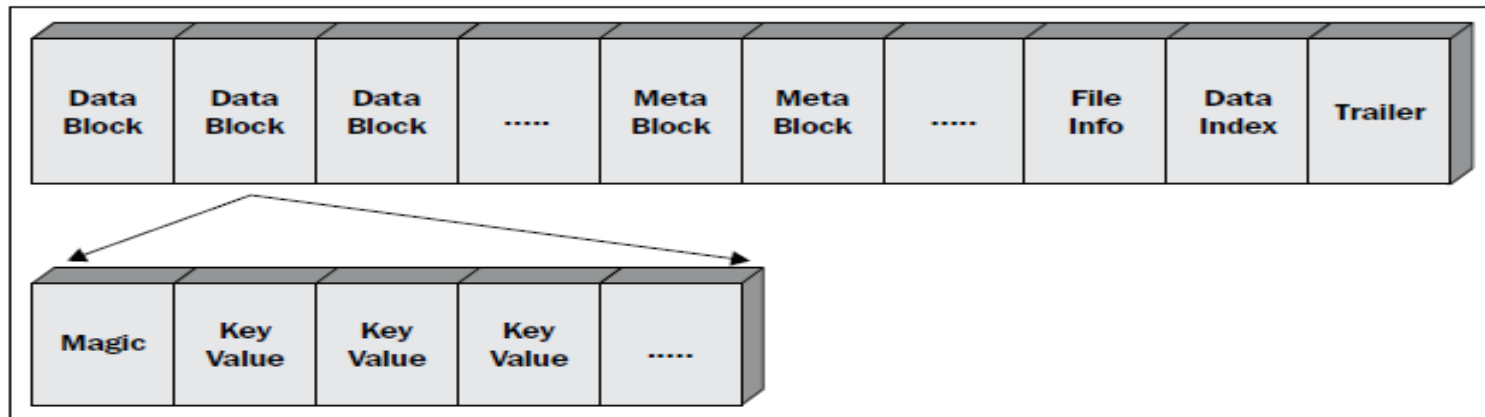
# HBase Architecture – KeyValue



# HBase Architecture – HFile

- Real data storage file
- Variable number of data blocks - Header and a number of serialized KeyValue instances.
- Fixed number of file info blocks and trailer blocks
- Index blocks
- Meta blocks
- Default size of the block is 64 KB
- Region Split - `hbase.hregion.max.filesize`
- Compaction process

# HBase Architecture – HFile



# HBase Architecture – hbase:meta

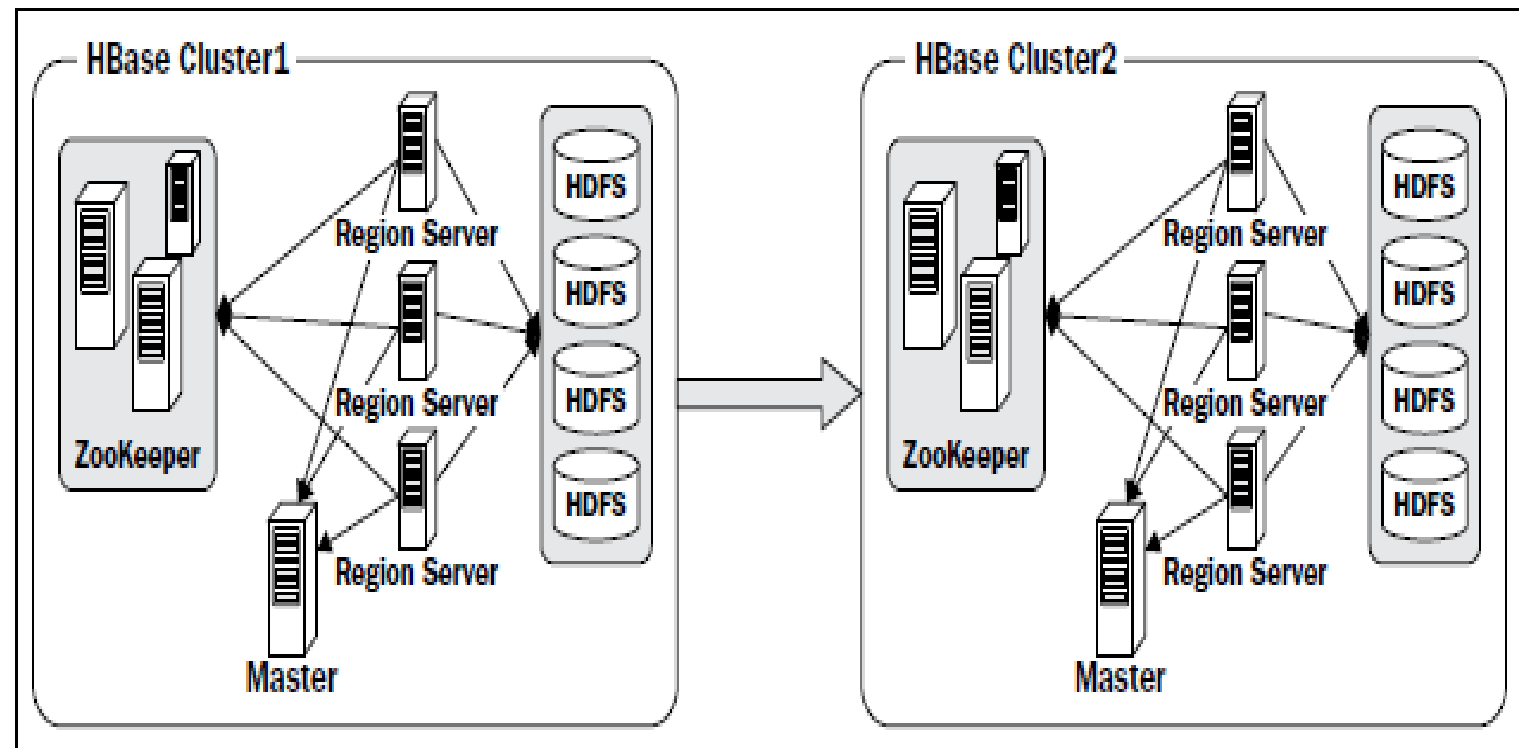
- List of all the regions for user-defined tables
- Region key - ([table],[region start key],[region id])
- First region is with an empty start key
- Value - info:regioninfo (serialized the HRegionInfo instance for this region)
- Value - info:server (server:port of the RegionServer containing this region)
- Value - info:serverstartcode (start time of the RegionServer process that contains this region)
- Table is split - Two new columns (info:splitA and info:splitB)
- Data reading- Connects to ZooKeeper and looks up the location of the hbase:meta table



# HBase Architecture – Data Replication

- Copying data from one cluster to another cluster
- Achieved by log shipping asynchronously
- Serves as a disaster recovery solution
- Master-push pattern - One master cluster can replicate any number of slave clusters
- Each region server will participate to replicate its own batch
- Every HBase cluster has a unique cluster ID stored on the file system
- The master connects to the slave's ZooKeeper ensemble using the provided cluster ID
- Can be done at a column-family level

# HBase Architecture – Data Replication



# HBase High Availability

- An area of active design and development work
  - ☐ From the very beginning, HBase has been leveraging HDFS-based data replication
  - ☐ HBase supports its own data partitioning across nodes in the cluster
  - ☐ The two above features combined give a three 9s data availability
- The next iteration in delivering HBase HA is called Hbase Read HA, where a failed read from one (Primary) RegionServer can be failed over to another (Secondary) RegionServer
  - ☐ Number of secondary RegionServers is configurable via the region replication factor which is usually set to 2
  - ☐ This is an automatic recovery process during which Hbase loses write availability and data for reads may be stale (as only the Primary RegionServer accept writes and keeps the latest data)
  - ☐ With a region replication factor of 2, HBase Read HA delivers four 9s data availability

# HBase vs RDBS

- HBase is not a relational database system and it is not positioned as a direct replacement for relational databases
  - ❖ Hbase is a “data store” rather than a “database”
- HBase does not support a declarative query language like SQL
  - ❖ Data access is performed imperatively through the client-side API
  - ❖ It lacks column types (byte arrays are used instead), triggers, and other common features of a RDBMS
- Traditional relational databases are not inherently distributed and don’t scale out easily
  - ❖ HBase is a distributed persistence store by design that uses the storage, memory and CPU resources of nodes in a “cluster” scaling out as the data size increases
  - ❖ HBase leverages HDFS for data durability (data is transparently replicated, 3-way by default)
- HBase does not support the full set of ACID properties

# HBase vs RDBMS

<u>HBase</u>	RDBMS
Schema-less in database	Having fixed schema in database
Column-oriented databases	Row oriented data store
Designed to store De-normalized data	Designed to store Normalized data
Wide and sparsely populated tables present in <u>HBase</u>	Contains thin tables in database
Supports automatic partitioning	Has no built in support for partitioning
Well suited for OLAP systems	Well suited for OLTP systems
Read only relevant data from database	Retrieve one row at a time and hence could read unnecessary data if only some of the data in a row is required
Structured and semi-structure data can be stored and processed using <u>HBase</u>	Structured data can be stored and processed using RDBMS
Enables aggregation over many rows and columns	Aggregation is an expensive operation

# HBase and Hive - Work Together

- Apache HBase - the NoSQL database for Hadoop and is great at fast updates and low latency data access
- Apache Phoenix - a SQL skin for data in Hbase
- Apache Hive is - SQL engine for Hadoop providing the deepest SQL analytics and supporting both batch and interactive query patterns

## Common Use Cases:

- Using HBase as the online operational data store for fast updates on hot data such as current partition for the hour, day etc
- Executing operational queries directly against HBase using Apache Phoenix
- Aging data in HBase to Hive tables using standard ETL patterns
- Performing deep SQL analytics using Hive



# NoSQL

- A scale-out, shared-nothing architecture, capable of running on a large number of nodes
- A non-locking concurrency control mechanism so that real-time reads will not conflict writes
- Scalable replication and distribution – thousands of machines with distributed data
- An architecture providing higher performance per node than RDBMS
- Schema-less data model

# CAP Theorem

In the parlance of Eric Brewer's CAP (**C**onsistency, **A**vailability and **P**artition Tolerance) theorem, HBase is a **CP** type system. The above means that in order to support the distributed (Partitioned)persistence and programming model and strict consistency (**C**), HBase has to forfeit Availability (**A**) or a guarantee that every request receives a response about whether it was successful or failed, therefore not making the data available (for example, the row may be locked for an update by one process and the read operation against the same row performed by another process may fail to get the data).

# HBase vs Cassandra vs MongoDB

- **Key characteristics:**

- Distributed and scalable big data store
- Strong consistency
- Built on top of Hadoop HDFS
- CP on CAP

- **Good for:**

- Optimized for read
- Well suited for range based scan
- Strict consistency
- Fast read and write with scalability

- **Not good for:**

- Classic transactional applications or even relational analytics
- Applications need full table scan
- Data to be aggregated, rolled up, analyzed cross rows

- **Usage Case:**

- Facebook message

# HBase vs Cassandra vs MongoDB

- **Key characteristics:**
  - High availability
  - Incremental scalability
  - Eventually consistent
  - Trade-offs between consistency and latency
  - Minimal administration
  - No SPF (Single point of failure) – all nodes are the same in Cassandra
  - AP on CAP
- **Good for:**
  - Simple setup, maintenance code
  - Fast random read/write
  - Flexible parsing/wide column requirement
  - No multiple secondary index needed
- **Not good for:**
  - Secondary index
  - Relational data
  - Transactional operations (Rollback, Commit)
  - Primary & Financial record
  - Stringent and authorization needed on data
  - Dynamic queries/searching on column data
  - Low latency
- **Usage Case:**
  - Twitter, Travel portal

# HBase vs Cassandra vs MongoDB

- Key characteristics:
  - Schemas to change as applications evolve (Schema-free)
  - Full index support for high performance
  - Replication and failover for high availability
  - Auto Sharding for easy Scalability
  - Rich document based queries for easy readability
  - Master-slave model
  - CP on CAP
- Good for:
  - RDBMS replacement for web applications
  - Semi-structured content management
  - Real-time analytics and high-speed logging, caching and high scalability
  - Web 2.0, Media, SAAS, Gaming
- Not good for:
  - Highly transactional system
  - Applications with traditional database requirements such as foreign key constraints
- Usage Case:
  - Craigslist, Foursquare

# Not Good Use Cases for HBase

- Relatively small number of rows (under a few million) where a RDBMS is a better choice
  - The size threshold would be in the region of hundreds of millions or billions of records
- Situations where you need to have the “bona fide” features of a RDBMS: ACID properties, type-safety of columns, triggers, etc.
- Limited number of physical nodes, e.g. 5: limitation imposed by HDFS that requires the default 3-way replication and a dedicated NameNode machine



# Interfacing with HBase

- HBase client applications are mostly written in Java using HBase Java client API
  - **Note:** There is no JDBC driver for HBase, you have to operate on HBase Java classes
- The HBase Shell
- In addition to Java API, HBase supports the Avro serialization, the Thrift and RESTful API (Stargate) gateways
- Apache Phoenix takes your SQL query, compiles it into a series of HBase scans, and orchestrates the running of those scans to produce regular JDBC result sets

## Notes:

Thrift is a software framework that allows you to create cross-language bindings for RPC-like communications with the other party. Thrift supports generating language bindings (IDLs) for more than 14 languages: Java, C#, C++, Python, Ruby etc.



# HBase Data Modeling

- Each row can store different numbers of columns and data types
- Ideal for storing semi-structured data
- Tables are composed of rows and these rows are composed of columns
- Rows are identified by a unique rowkey and are compared with each other at the byte level
- Columns are organized into column families
- No restriction on the number of columns that can be grouped together in a single column family
- At the storage level, all columns in a column family are stored in a single file, called HFile

# HBase Data Modeling

- Placeholder for the column value is called cell
- Each cell stores the most recent value and the historical values for the column
- Regions groups the continuous range of rows and stores them together at lower levels in region servers
- Records are stored in HFiles as key-value pairs
- Records from a single column family might be split across multiple HFiles, but a single HFile cannot contain data for multiple column families

# HBase Table Design

- HBase uses its own HFile storage format
- HBase is a quasi-schemaless system
  - The system designer must define the table schema and column families upfront
- An HBase system consists of a set of tables made of rows and columns that hold objects
  - A column holds an entity's attribute
- The first column must be allocated for storing a unique row identifier defined as the Primary Key (this unique identifier is called the rowkey or rowid)
- Table cells (the intersection of rows and columns) are versioned (by default, three versions are kept) with a timestamp assigned by HBase, or programmatically by the client, written alongside the value
- A cell's content is stored as an uninterrupted array of bytes

# HBase Table Design

- Table rows are sorted by the table's rowkey
  - So, the rowkey can also be viewed as the table's index
- All table accesses are via the table's rowkey
- Row updates are atomic

# HBase Table Design

- Number of column families and which data goes to which column family
- Maximum number of columns in each column family
- Type of data to be stored in the column
- Number of historical values that need to be maintained for each column
- Structure of a rowkey

# Column Families

- Following the Google's Bigtable design, HBase allows for grouping entity's attributes (columns) into column families
- All column family members have a common prefix
  - For example, the column family engine may include these two columns (a column in HBase is referred to as the "column qualifier"): engine:hp and engine:cyl
- Column families must be specified up front as part of the table schema definition
- New column family members can be added to predefined column families at run-time
  - For example, a new column engine:weight can be added to the existing engine column family
  - This feature of HBase helps accommodate agile business requirements
- Column families help with entity storage and retrieval optimization



# A Cell's Value Versioning

- Changes of a cell's value are versioned by a timestamp
- BY default, HBase stores the last three versions of a cell's value
  - This is parameter is configurable per column family
- You have access to the version history and can fetch element from previous versions by supplying the version's timestamp



# Timestamps

- By default, HBase uses the current time in milliseconds since Unix epoch time
- The current time is captured on the RegionServer that performs an operation that change the cell value's version
- For the timestamp facility to work properly, system clocks on all machines in the HBase cluster must be in sync

# Accessing Cells

- HBase uses a cell coordinate (cell path) to locate a cell in a table
- A cell coordinate is made up as follows:
  - The table name, followed by the primary key (rowkey), followed by the column family, followed by the column qualifier (column family member) and, optionally, the timestamp, or
  - Table\_name/rowkey/column/timestamp

# HBase Table Design Digest

- Rows are sorted by the primary key (rowkey)
- Cell values are accessed using the “table\_name/rowkey/column/(and optionally)timestamp” positioning coordinates
- Columns can be added to existing column families at run-time
- Cell values are typeless and are always treated as byte arrays (leaving the type casting exercise to client applications)

# Table Horizontal Partitioning with Regions

- HBase tables are automatically partitioned horizontally (by rows) when a configurable size threshold is passed
- A table partition (containing a chunk of table rows) is referred to as a region
- A table's initial storage consists of a single region with new regions added as the number of rows crosses a configurable size threshold
- Regions get distributed over an HBase cluster

## **Notes:**

Assignment of regions is mediated via ZooKeeper, Hadoop's distributed coordination service

# HBase Compaction

- HBase provide optimal read performance by having only one file per column family which minimizes the number of disk seek reads
- Using some internal heuristics (regulated by compaction policies), Hbase finds the good candidates (HFile) and combines them into one contiguous file on disk
- This “merging” process is called compaction
- This is a “work-in-process” and the compaction policies are subject to change

# Loading Data in HBase

- The recommended way to load data from HDFS into HBase is with a bulk loader
- The bulk loading process prepares and loads data in the HFile storage format (Hbase's own file format) directly into the RegionServers
  - The process bypasses the standard write path involving the WAL and MemStore which makes it very efficient
- Hbase comes with its own MapReduce-based bulk loader called `importtsv` that can read CSV-delimited files stored in HDFS and import them into Hbase
- After this, you can use Hbase against the imported data



# Column Families Notes

- In Most of situations, you should have only once column family in your table
- More than three column families would degrade performance of your system
- If you have more than once column family (two or three), try to limit data access patterns to one column family at a time
  - That will minimize the number of disk seek reads and make an efficient use of the MemStore (a Java heap-based write buffer, allocated one per column family)



# Rowkey Notes

- Usually, the row key ids are sorted lexicographically to improve the scan (sequential access) of the table
- This arrangement can result in hotspotting where a single physical machine takes all the user traffic for reads/writes while other machines in the cluster that host other data regions may sit idle
- If you foresee random access to your data, you can use the following techniques/tricks to ensure a fair distribution of key values across multiple regions on writes (and subsequent reads):
  - **Salting**: randomly (kind of round-robin way of) adding specially designed prefix to the key
  - **Hashing**: delegating the random key allocation to a one-way hash function
  - **Resersing** the fixed-width key: making the least significant digit appear first
  - **Note**: These techniques require deep understanding of effective management of data regions in Hbase and the topic is beyond the scope of this module

# Region Splitting – Pre-splitting

- Create a table with many regions by supplying the split points at the table creation time
- If no split points at hand, use RegionSplitter utility

**hbase org.apache.Hadoop.hbase.util.RegionSplitter test\_table HexStringSplit -c 10 -f f1**

- SplitAlgorithm and HexStringSplit

The former can be used if the row keys have a prefix for hexadecimal strings

The latter divides up the key space evenly assuming they are random byte arrays

# Region Splitting – Pre-splitting

- Custom SplitAlgorithm
- If split points at hand

**create 'test\_table', 'f1', SPLITS=> ['a', 'b', 'c']**

# Region Splitting – Auto-splitting

- Once a region gets to a certain limit, it is automatically split into two regions
- Pluggable region split policy to calculate the split points
  - ConstantSizeRegionSplitPolicy
    - ✓ Default split policy before 0.94 to check if the total data size for one of stores in the region bigger than **hbase.hregion.max.filesize**

# Region Splitting – Auto-splitting

- Pluggable region split policy to calculate the split points
  - IncreasingToUpperBoundRegionSplitPolicy
  - ✓ Default split policy after 0.94 based on the number of regions hosted in the same region server  $\text{Min}(R^2 * \text{"hbase.hregion.memstore.flush.size"}, \text{"hbase.hregion.max.filesize"})$ , R is the number of regions of the same table hosted on the same region server


# Region Splitting – Auto-splitting

- Pluggable region split policy to calculate the split points
  - `KeyPrefixRegionSplitPolicy`
    - ✓ Configure the length of the prefix for your row keys for grouping them
    - ✓ This split policy ensures that the regions are not split in the middle of a group of rows having the same prefix

# Region Splitting – Forced-splitting

**split 'b07d0034cbe72', 'b'**





# How region splits are implemented



# Region Merges

# HBase Shell

- Hbase comes with the Jruby-based shell which acts as a command-line interface to the underlying Hbase client Java API
- You start the shell with this command:

## **Hbase shell**

- You can run it in either interactive or unattended (batch) mode
  - Hbase scripts are Jruby files (that interface with Hbase classes) that you execute as follows:

```
Hbase org.jruby.Main <script.rb> args
```

- **Notes:**

Hbase Jruby scripts import Java API via the require 'java' instruction and then import the needed statement, e.g.

```
import org.apache.Hadoop.hbase.HBaseConfiguration
```

```
Import org.apache.Hadoop.hbase.HConstants
```

# HBase Shell Command Groups

- HBase shell commands are organized into groups some of which are shown in the table below

Command Group Name	Commands in the Group
general	status,table_help,version,whoami
ddl	alter,alter_async,alter_status,create,describe,disable,disable_all,drop,drop_all,enable,enable_all,exists,get_table,is_disabled,is_enabled,list,show_filters
dml	count,delete,deleteall,get_counter,incr,put,scan,truncate
security	grant,revoke,user_permission

# Accessing HBase

- Establishing Connection

```
Configuration newConfig = new Configuration(defaultConfig);  
HConnection connection =  
HConnectionManager.createConnection(newConfig);  
HTableInterface table = connection.getTable("Costumers");
```

# Accessing HBase

- A connection needs to be established with the help of the HConnection class
- HTable class is not thread-safe as concurrent modifications are not safe
- Multiple HTable instances with the same configuration reference, the same underlying HConnection instance can be used
- Creating an HTable instance is a slow process
- Consider using the HTablePool class - HTableInterface usersTable = pool.getTable("Costumers");



# Accessing HBase - CRUD operations

- Put
- Get
- Delete
- Scan
- Increment

# Accessing HBase – Writing Data

- The data is synchronously written into HLog
- The data is synchronously written to the memstore
- Every time the memstore flushes the data to the disk, a new HFile is created
- A Put class instance is used to store data in an HBase table
- Example:

```
HTable table = new HTable(conf, "tab1");  
Put put = new Put(Bytes.toBytes("row-1"));  
put.add(Bytes.toBytes("cf1"),  
Bytes.toBytes("greet"),Bytes.toBytes("Hello"));  
put.add(Bytes.toBytes("cf1"),  
Bytes.toBytes("person"),Bytes.toBytes("John"));  
table.put(put);  
table.close();
```

# Accessing HBase – Reading Data

- LRU cache for reads
- Every column family has its own block cache
- Get class instance is used to read the data back from the HBase table
- Data reading in HBase can take place in the form of a batch representing multiple rows - `get(List<Get> gets)`
- Example:

```
List<Get> gets = new ArrayList<Get>();  
Get get1 = new Get(Bytes.toBytes("row-1"));  
get1.add(Bytes.toBytes("cf1"), Bytes.toBytes("greet"));  
gets.add(get1);
```

# Accessing HBase – Updating Data

- Data updation in HBase is done in a manner that is similar to writing it
- Example:

```
HTable table = new HTable(conf, "tab1");  
Put put = new Put(Bytes.toBytes("row-1"));  
put.add(Bytes.toBytes("cf1"),  
Bytes.toBytes("greet"),Bytes.toBytes("GoodMorning"));  
put.add(Bytes.toBytes("cf1"),  
Bytes.toBytes("person"),Bytes.toBytes("David"));  
table.put(put);  
table.close();
```

# Accessing HBase – Deleting Data

- Delete command only marks the cell for deletion rather than deleting the data immediately
- Actual deletion is performed when the compaction of HFiles is done
- Delete class instance is used to delete the data from the HBase table
- Data deletion in HBase can happen in the form of a batch representing multiple rows - `delete(List<Delete> deletes)`
- Example:

```
List<Delete> deletes = new ArrayList<Delete>();
```

```
Delete delete1 = new Delete(Bytes.toBytes("row-1"));
```

```
delete1.deleteColumn(Bytes.toBytes("cf1"), Bytes.toBytes("greet"));
```

```
deletes.add(delete1);
```

# Creating and Populating a Table

- **Note:** All names in Hbase shell must be used in quotes

```
create 'myTable', 'myColFam'
```

- The above command creates the myTable table with the myColFam column family

```
put 'myTable', 'row1', 'myColFam:col1', 'Some value'
```

```
Put 'myTable', 'row2', 'myColFam:col2', 'Some other value'
```

- The above commands insert 'Some value' in a cell located at myTable/row1/myColFam:col1 and 'Some other value' in a cell located at myTable/row2/myColFam:col2



# Creating and Populating a Table

- **Note:** You can optionally put your own timestamp when inserting values with the put command:

```
put '<tableName>', 'rowid', 'colid', 'cell value', <your timestamp>
```

- Since Hbase is a schemaless storage system, you never need to predefine the column qualifiers upfront or assign them types – just provide the name at write time

# Getting a Cell's Value

- The following command will retrieve a row identified with the row1 rowkey from the myTable table

```
get 'myTable', 'row1'
```

COLUMN	CELL
myColFam:col1	timestamp=<...>, value=Some value

- If the row identified with the rowkey does not exist, you will get back an empty response

# Counting Rows in an HBase Table

- The count command (the dml group) returns the number of rows in a table  
count '<tablename>', CACHE=>1024  
➤ **Note:** => is the assignment operator (=)
- The above count command fetches 1024 rows at a time to speed up row counting

# Summary

- Hbase is designed for hosting tables of sizes in the petabyte range
- It offers linear scalability with automatic and configurable partitioning of tables with strictly consistent reads and writes
- Hbase rows are identified and accessed by their rowkey ids which act as a primary key
- Hbase offers a rich Java client API; access from other languages is also supported via the Thrift or RESTful gateways
- Hbase also offers an interactive shell as a command-line interface

# Objective

In this section, participants will learn about some of the elements of Java API for Hbase:

- The ResultScanner Interface
- The Scan Class
- The KeyValue class
- The Result Class
- The Cell Interface
- The Bytes Utility Class

# HBase Java Client

- HBase is written in Java which is also its primary application client
- An HBase Java client is a regular Java app with the main() method
- You use HBase-specific classes to read the HBase configuration and set up HBase operations (put, get, scan, etc.)



# HBase Scanners

- HBase scanners work like Java iterators that orderly go over all qualifying rows in a table
  - You may think of scanners are RDBMS cursors
- Scanners' functionality is offered through the `org.apache.Hadoop.hbase.client.ResultScanner` interface
- The `ResultScanner` works in tandem with the `org.apache.Hadoop.hbase.client.Scan` Java class that helps limit the number of rows to scan:

```
Scan scan = new Scan();  
ResultScanner scanner = table.getScanner(scan);
```

- After that you can scan through the table:

```
For (Result scannerResult: scanner){  
    System.out.println("Scaned: " + scannerResult);  
}
```

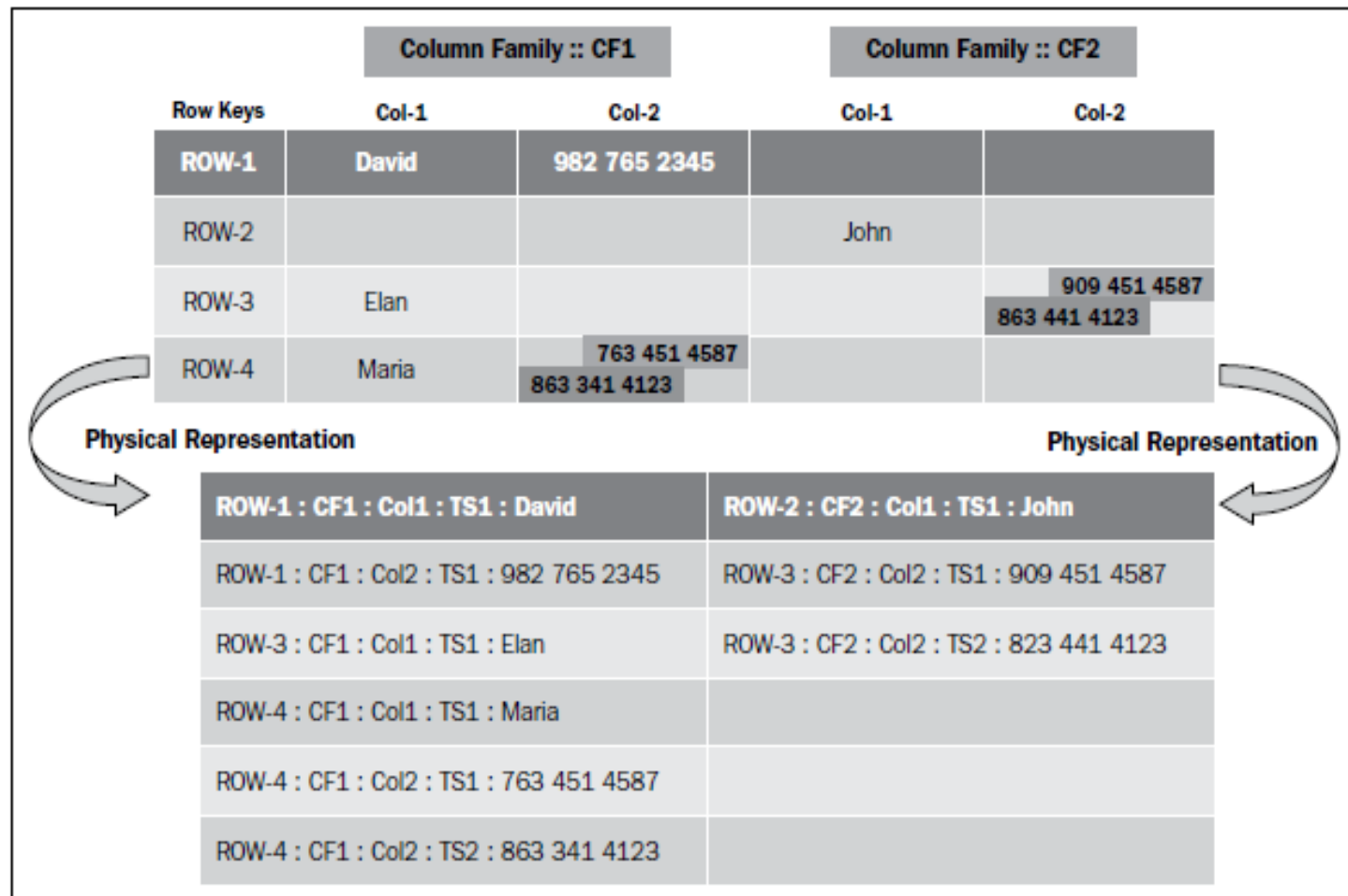
# Securing HBase

- User authentication with Kerberos
- User authorization via access an controller coprocessor
- Access controller coprocessor is only implemented at the RPC level
- Based on the Simple Authentication and Security Layer
- Set `hbase.security.authentication` to true
- Set `hadoop.security.authentication` to true
- Set `hbase.security.authorization` to true
- Set `hbase.coprocessor.master.classes` to `AccessController`
- Encrypted communication - set `hbase.rpc.protection` to `privacy`
- Authorization with Apache Ranger

# Advanced Data Modeling - Keys

- Row key - logical representation of an entire row
- Column key - Combining the column family and the column qualifier
- Fewer rows with many columns (flat and wide tables)
- Fewer columns with many rows (tall and narrow tables)
- Use case 1 - Tweets (Many users vs the users with tons of tweets)
- Use case 2 - Time series data (the event's time as a row key) and hotspotting issue
- Use case 2 Solution - prefix the row key or salting
- HBase does not provide direct support for secondary indexes

# Advanced Data Modeling - Keys



# Advanced Data Modeling - Keys

- Secondary indexes use cases - A cell lookup using coordinates other than the row key
- Secondary indexes use cases - Scanning a range of rows from the table ordered by the secondary index
- Approaches in HBase to create secondary indexes - Application-managed approach
- Approaches in HBase to create secondary indexes - Index support in HBase, such as Lily HBase indexer
- Approaches in HBase to create secondary indexes - Coprocessor

# Advanced Data Modeling - Scans

- Table scans are similar to iterators in Java or nonscrollable cursors in the RDBMS world
- Querying the data to access the complete set of records for a specific value by applying filters
- Get() operation - Need to define the row key
- Scan - Define the optional startRow parameter
- Partial key scan by using the start and stop keys
- Defines one more optional parameter called filter
- Avoid too many calls - row caching via `hbase.client.scanner.caching`
- Set the caching limit for individual scan calls - `setScannerCaching`
- Limiting the columns returned - `setBatch`



# Advanced Data Modeling - Filters

- Scan - setFilter(Filter filter)
- Utility filters - TimeStampFilter, SingleColumnValueFilter, SingleColumnValueExcludedFilter, PageFilter, ColumnCountGetFilter, RandomRowFilter
- Comparison filters - RowFilter, ValueFilter, FamilyFilter
- Custom filters - Wrapper filters and Wrapper filters

# Advanced HBase API - Counter

- Collect statistics such as user clicks, likes, views, and so on
- Allow us to increment a column value
- Potential of real-time accounting
- Normal way for incrementing column values - locking the row, reading, incrementing, writing the value, and finally releasing the row
- Drawback of normal way for incrementing column values - lot of I/O overheads
- Counters work only with a single row - Single row lock
- Single counters - a single RPC call is made to increment the value for a single counter
- Multiple counters - a single RPC call is made to increment the value of multiple counters
- Example:

```
incr '<table>', '<row>', '<column>', [<increment-value>]
```

# Advanced HBase API - Admin

- The data definition API
- The HBaseAdmin API

# HBase Clients

- HBase Shell
- Object Mapper – Kundera
- REST API – JSON, XML
- THRIFT Client

# HBase Administration – ETL

- The state of HBase data recovery to date
  - Table exports within a cluster, which could be copied remotely
  - Table-level snapshots
  - Real-time cross-cluster replication

# HBase Administration – ETL

- The Export job takes the source table name and the output directory name as inputs
- The Import job reads the records from the source sequential file by creating Put instances from the persisted Result instances. It then uses the HTable API to write these puts to the target table
- The CopyTable MapReduce job is used to scan through an HBase table and directly write to another table
- Advanced import with importtsv
- The LoadIncrementalHFiles utility, also called completebulkload, handles the messy business of installing and activating new HFiles in a table in Hbase
- The snapshots utility
- The backup utility
- HBase replication
- Spark and MapReduce



# Advanced HBase Table Design

## How to approach schema design

- 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?
- 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?

# Advanced HBase Table Design

## De-normalization

- Data is repeated and stored at multiple locations
- Querying the data much easier and faster because you no longer need expensive JOIN clauses

# Advanced HBase Table Design

- Heterogeneous data in the same table
- Rowkey design strategies
- I/O considerations
- From relational to non-relational
- Advanced column family configurations
- Filtering data

# HBase and MapReduce

- Running MapReduce over Hbase
  - Mapper class should extend the TableMapper class
  - The map method of the Mapper class takes the rowkey of the Hbase table as an input key
  - The define input key is the ImmutableBytesWritable object
  - Another parameter, the org.apache.hadoop.hbase.client.Result object contains the input values as column/column-families from the HBase table
  - Reducer class should extend the TableReducer class
  - The output key is defined as NULL
  - The output value is defined as the org.apache.hadoop.hbase.client.Put object

# HBase and MapReduce

- Running MapReduce over Hbase

- Configure the `org.apache.hadoop.hbase.client.Scan` object and optionally define parameters such as start row, stop row, row filter, columns, and the column-families for the scan object.
- Set the record caching size
- Set the block cache for scan object as false
- `TableInputFormatBase`. It iterates over the splits and creates a new `TableRecordReader`
- Each `TableRecordReader` instance handles exactly one region
- `TableOutputFormat` class is used to allow output to HBase tables

# HBase and MapReduce

- HBase as a data source

```
static class HBaseTestMapper extends TableMapper<Text, IntWritable>
Scan scan = new Scan();
scan.setCaching(250);
scan.setCacheBlocks(false);
Job job = new Job(conf, "Read data from " + table);
job.setJarByClass(HBaseMRTest.class);
TableMapReduceUtil.initTableMapperJob(table, scan,
HBaseSourceTestMapper.class, Text.class, IntWritable.class, job);
```



# HBase and MapReduce

- HBase as a data sink

```
Job job = new Job(conf, "Writing data to the " + table);  
job.setOutputFormatClass(TableOutputFormat.class);  
job.getConfiguration().set(TableOutputFormat.OUTPUT_TABLE, table);
```

```
static class HBaseSourceTestReduce extends TableReducer<.,.>  
TableMapReduceUtil.initTableReducerJob("customers",  
HBaseTestReduce.class, job);
```

# Extending HBase

- The two kinds of coprocessors (observer and endpoint)
- Observers allow the cluster to behave differently during normal client operations. Endpoints allow you to extend the cluster's capabilities, exposing new operations to client applications.
- Observers sit between the client and HBase, modifying data access as it happen
- Observer coprocessors as analogous to triggers from a relational database
- Multiple observers can be registered simultaneously
- The CoprocessorHost class manages observer registration and execution on behalf of the region

# Extending HBase

- Three kinds of observers (RegionObserver, WALObserver and MasterObserver)
- Endpoints are a generic extension to HBase
- Endpoints execute on the RegionServers
- Endpoint coprocessors are similar to stored procedures in other database engines
- Endpoint coprocessors functionality is based on the custom code that defines the coprocessor
- Endpoint that computes simple aggregates like sum and average

# Extending HBase

## Coprocessor Use Cases:

- Calculation huge dataset on the server side
- Secondary Indexes
- Security Check
- Logging
- Referential Integrity

# Extending HBase

## Coprocessor Analogies:

- Observer -> Trigger in RDBMS
- Endpoint -> Stored Procedure in RDBMS
- MapReduce -> Data Locality
- AOP -> Advice by intercepting the request

# Extending HBase

## Coprocessor Implementation:

- Implement one of the interface – e.g. Coprocessor, RegionObserver, CoprocessorService
- Load the coprocessor from the configuration or using HBase Shell
- Call the coprocessor from client-side code



# Extending HBase

## Static Loading Coprocessors

```
<property>
```

```
<name>hbase.coprocessor.region.classes</name>
```

```
<value>org.mynamespace.hbase.coprocessor.endpoint.SumEndPoint</value>
```

```
</property>
```

# Extending HBase

## Dynamic Loading Coprocessors

- Per-table basis
- Table must be taken offline to load the coprocessor
- Using Java API
- Using HBase Shell

```
disable 'airdelaydate'
alter 'airdelaydate', METHOD => 'table_att',
'Coprocessor'=>'/root/TrainingOnHDP/HBaseCookBook/target/HBaseCookB
ook-1.0-SNAPSHOT-jar-with-
dependencies.jar|ca.training.bigdata.hbase.coprocessor.AirDelayDataObser
ver|1073741823'
describe 'airdelaydate'
```

# HBase and Hive

```
CREATE TABLE IF NOT EXISTS pagecounts (projectcode STRING, pagename STRING, pageviews STRING,
bytes STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ' '
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION '/user/root/pagecounts';
```

```
select INPUT__FILE__NAME from pagecounts limit 10;
```

```
CREATE VIEW IF NOT EXISTS pgc (rowkey, pageviews, bytes) AS
SELECT concat_ws('/',
               projectcode,
               concat_ws('/',
               pagename,
               regexp_extract(INPUT__FILE__NAME, 'pagecounts-(\\d{8}-\\d{6}}', 1))),
               pageviews, bytes
FROM pagecounts;
```

# HBase and Hive

```
CREATE TABLE IF NOT EXISTS pagecounts_hbase (rowkey STRING, pageviews STRING, bytes STRING)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ('hbase.columns.mapping' = ':key,0:PAGEVIEWS,0:BYTES')
TBLPROPERTIES ('hbase.table.name' = 'PAGECOUNTS');
```

```
FROM pgc INSERT INTO TABLE pagecounts_hbase SELECT pgc.* WHERE rowkey LIKE 'en/q%' LIMIT 10;
```

Go to HBASE SHELL, run the following command

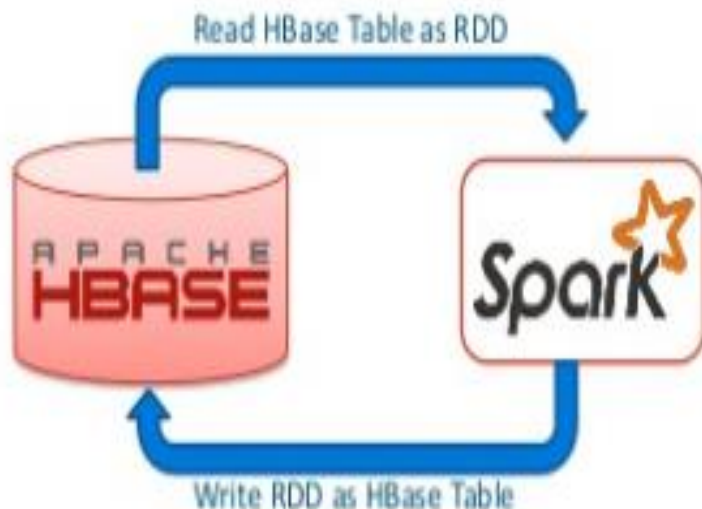
```
scan 'PAGECOUNTS'
```

Go to Phoenix Zeepline, and create new note:

```
@phoenix
CREATE VIEW "PAGECOUNTS" (pk VARCHAR PRIMARY KEY,
"0".PAGEVIEWS VARCHAR,
"0".BYTES VARCHAR)
```

# HBase and Spark

## HBase + Spark: Batch processing patterns

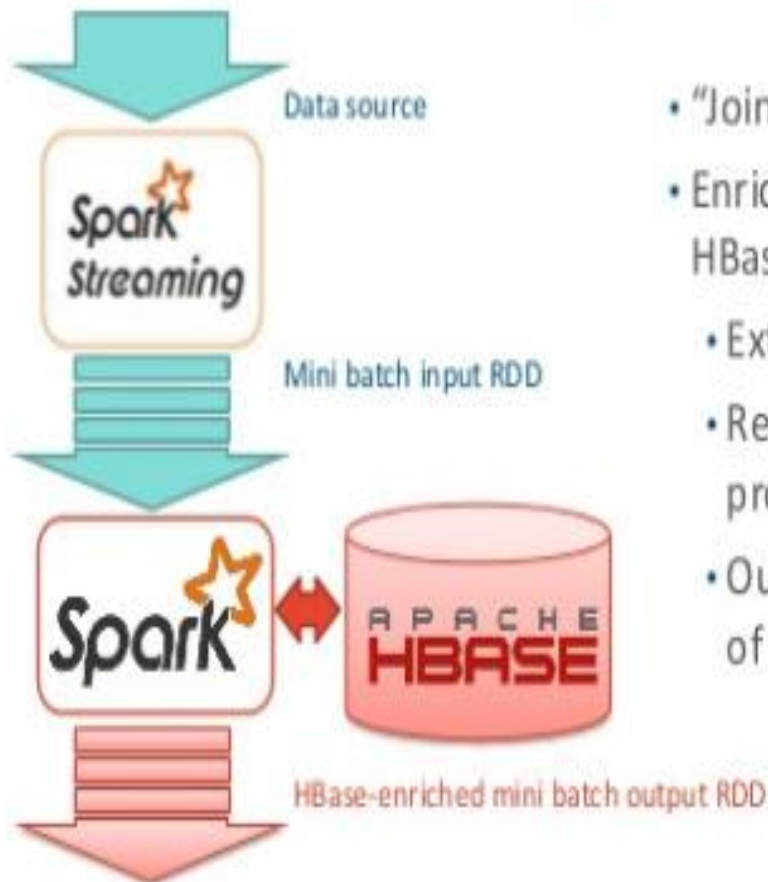


- Read dataset(RDD) from HBase Table
  - Use HBase's MR InputFormats
    - TableInputFormat
    - MultiTableInputFormat
    - TableSnapshotInputFormat
- Write dataset(RDD) to HBase Table
  - Use HBase's MR OutputFormat
    - TableOutputFormat
    - MultiTableOutputFormat
    - HFileOutputFormat



# HBase and Spark

## HBase + Spark Streaming – Enriching With HBase Data

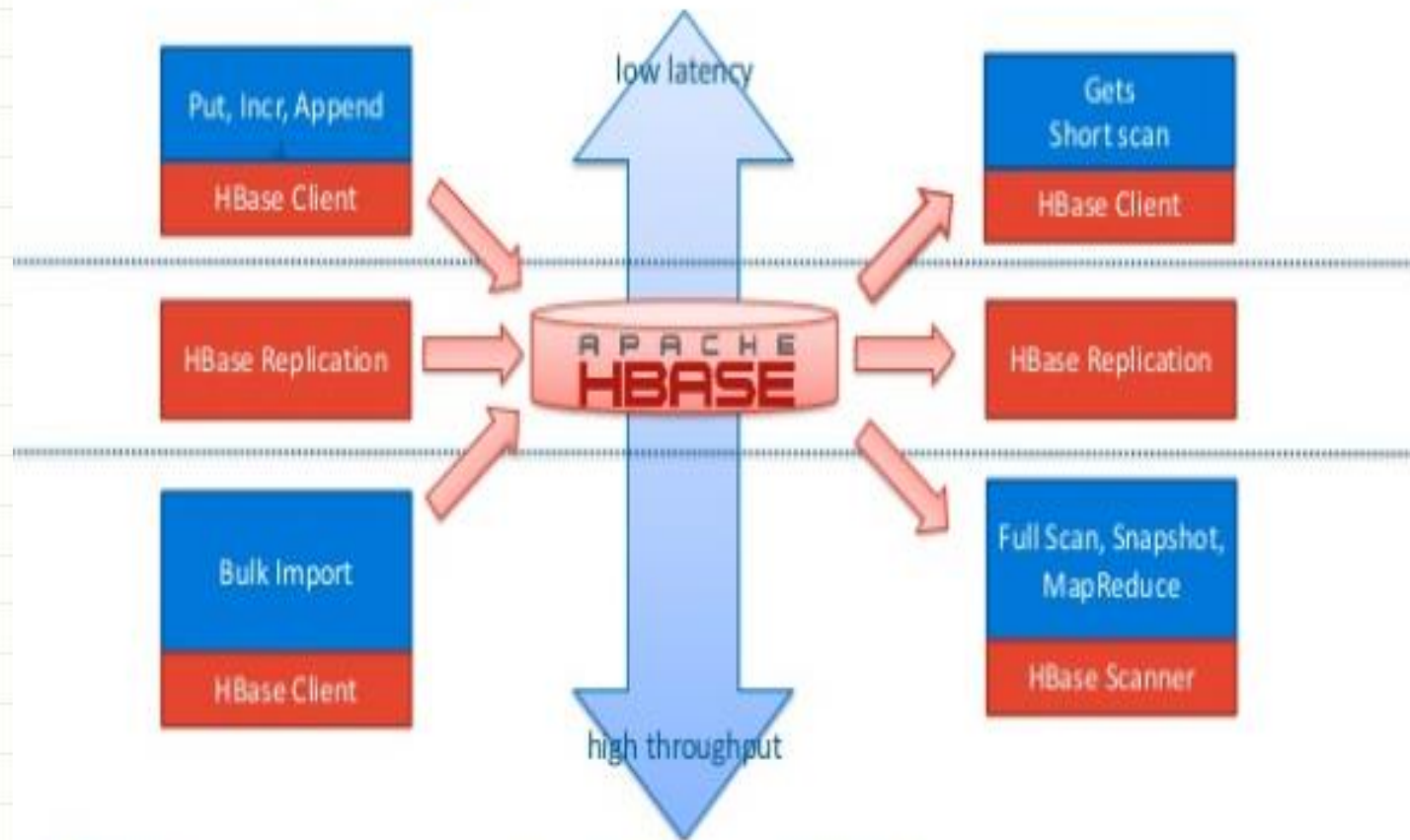


- “Join” a dataset with HBase data
- Enrich Streaming data source with HBase data
  - Extract information from minibatch
  - Read/write/update HBase data in processing
- Output HBase-data enriched stream of output RDDs



# HBase and Spark

How does Spark get data in and out of HBase?



# HBase Ecosystem - Monitoring

- Cloudera Manager
- Apache Ambari
- Hannibal

# HBase Ecosystem - SQL

- Apache Phoenix - SQL skin over HBase.
- Apache Trafodion - enterprise-class SQL-on-HBase solution targeting big data transactional or operational workloads.
- Honorable Mentions (Kylin, Themis, Tephra, Hive and Impala)

# HBase Ecosystem - Frameworks

- Tephra - Transactions for HBase.
- OpenTSDB - distributed time series database on top of HBase.
- Hortonworks HOYA - application that can deploy HBase cluster on YARN.
- Lily HBase Indexer - quickly and easily search for any content stored in HBase.
- Kite - Kite is a whole SDK designed to codify Hadoop best practices
- HappyBase - Allow the end user to leverage the HBase client through the use of Python
- AsyncHBase - Client replacement that offers better throughput and performance

# HBase Sizing and Tuning - Hardware

- HBase currently can use about 16–24 GB of memory for the heap
- G1GC collector has shown very promising results with heaps over 100 GB
- HBase cluster does not need a high core count

# HBase Sizing and Tuning - Storage

- HBase takes advantage of a JBOD disk configuration (Better performance and control hardware costs)
- Disk count is not currently a major factor
- The HBase writepath is limited due to HBase's choice to favor consistency over availability
- SSDs are currently overkill and not necessary



# HBase Sizing and Tuning - Networking

- Important consideration when designing an HBase cluster
- standard Gigabit Ethernet (1 GbE) or 10 Gigabit Ethernet(10 GbE)
- Topofrack(TOR) switches
- VLANs

# HBase Sizing and Tuning – OS Tuning

- Linux based
- EXT4 for the local filesystem
- XFS is an acceptable filesystem
- Swapping and swap space
- Swapping is not used by the kernel of the processes

# HBase Sizing and Tuning – Hadoop Tuning

- Number of CPU cores that can be allocated for containers
- Amount of physical memory can be allocated for containers
- The minimum allocation for every container request at the RM

# HBase Sizing and Tuning – HBase Tuning

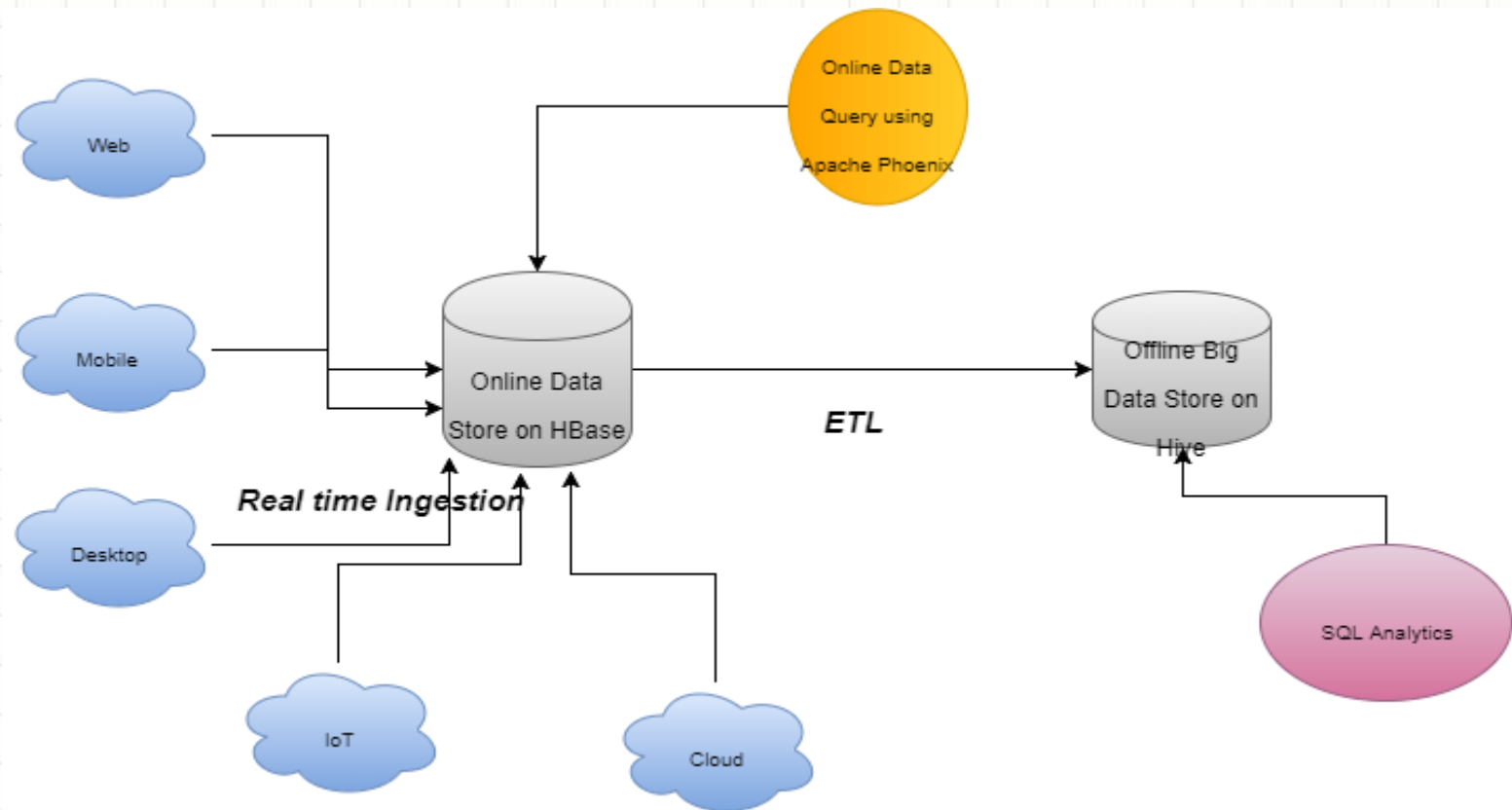
- Write-heavy workload (API or Bulk Load)
- Primary bottleneck for HBase is the WAL followed by the memstore

# HBase Sizing and Tuning – HBase Tuning

- Compression
- Load balancing
- Splitting regions
- Merging regions
- MemStore-local allocation buffers
- JVM tuning
- Other recommendations

# HBase Use Cases – Common Case

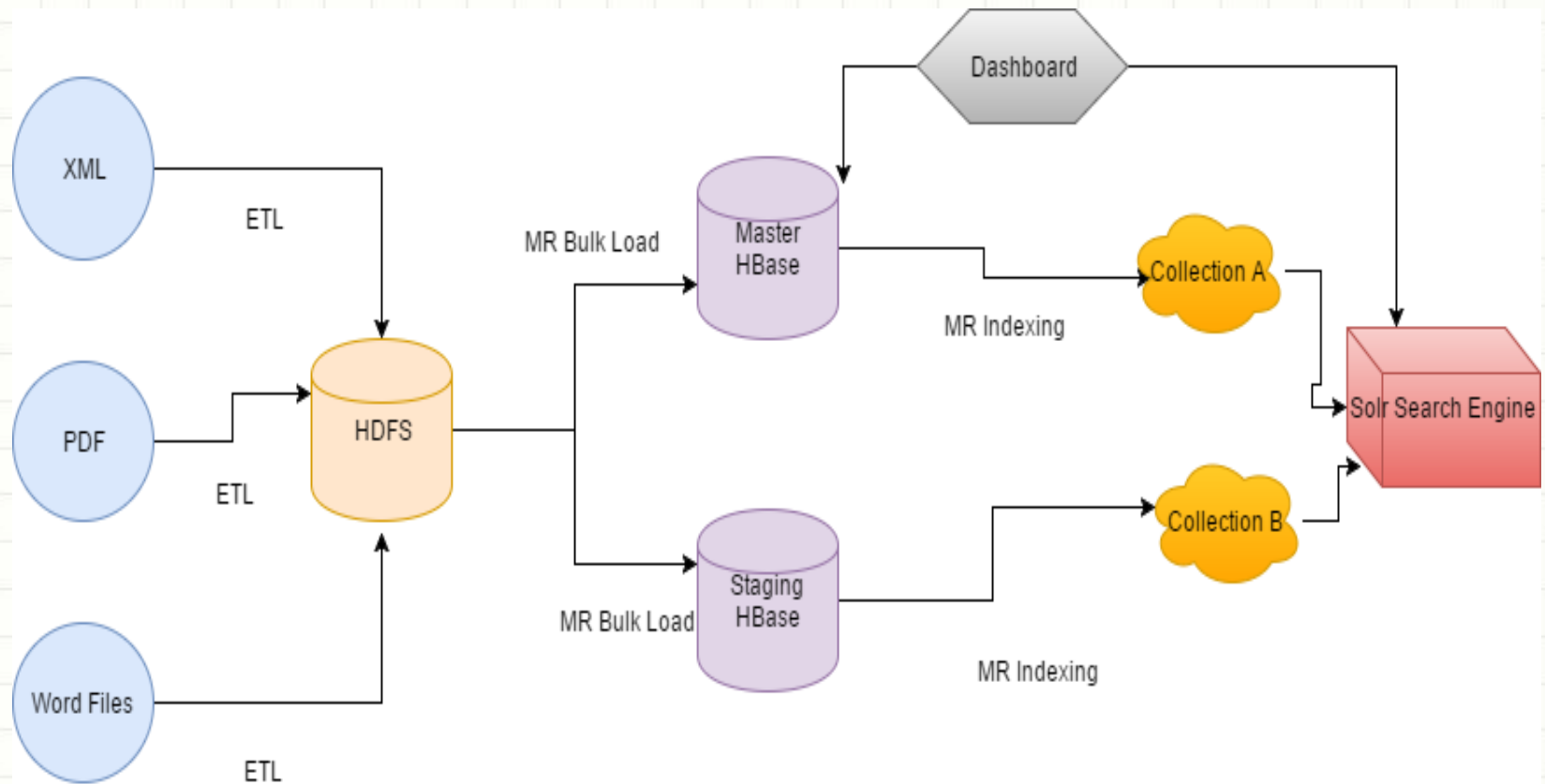
## Online Data Store





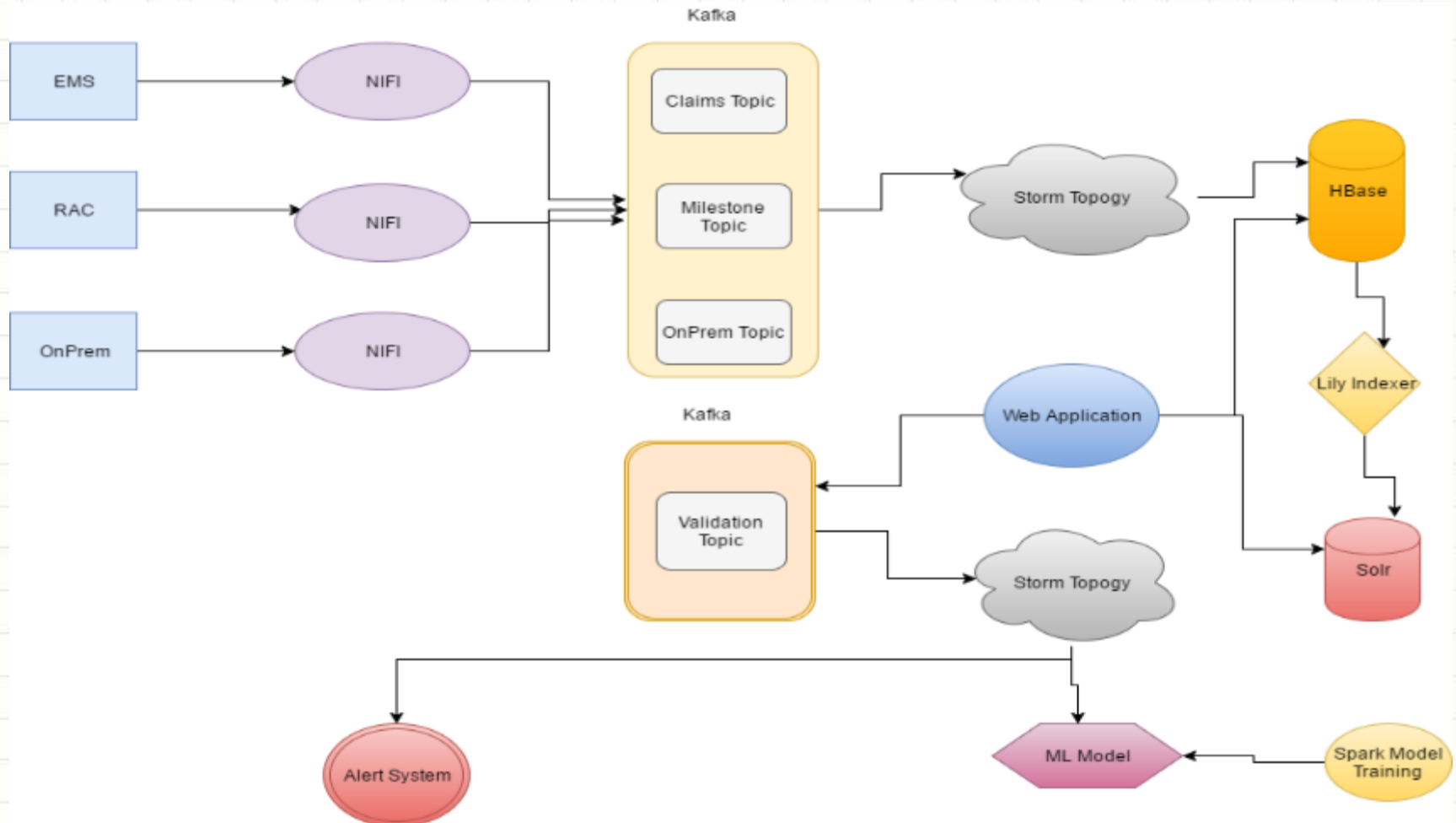
# HBase Use Cases – Document Search

## Storage Engine



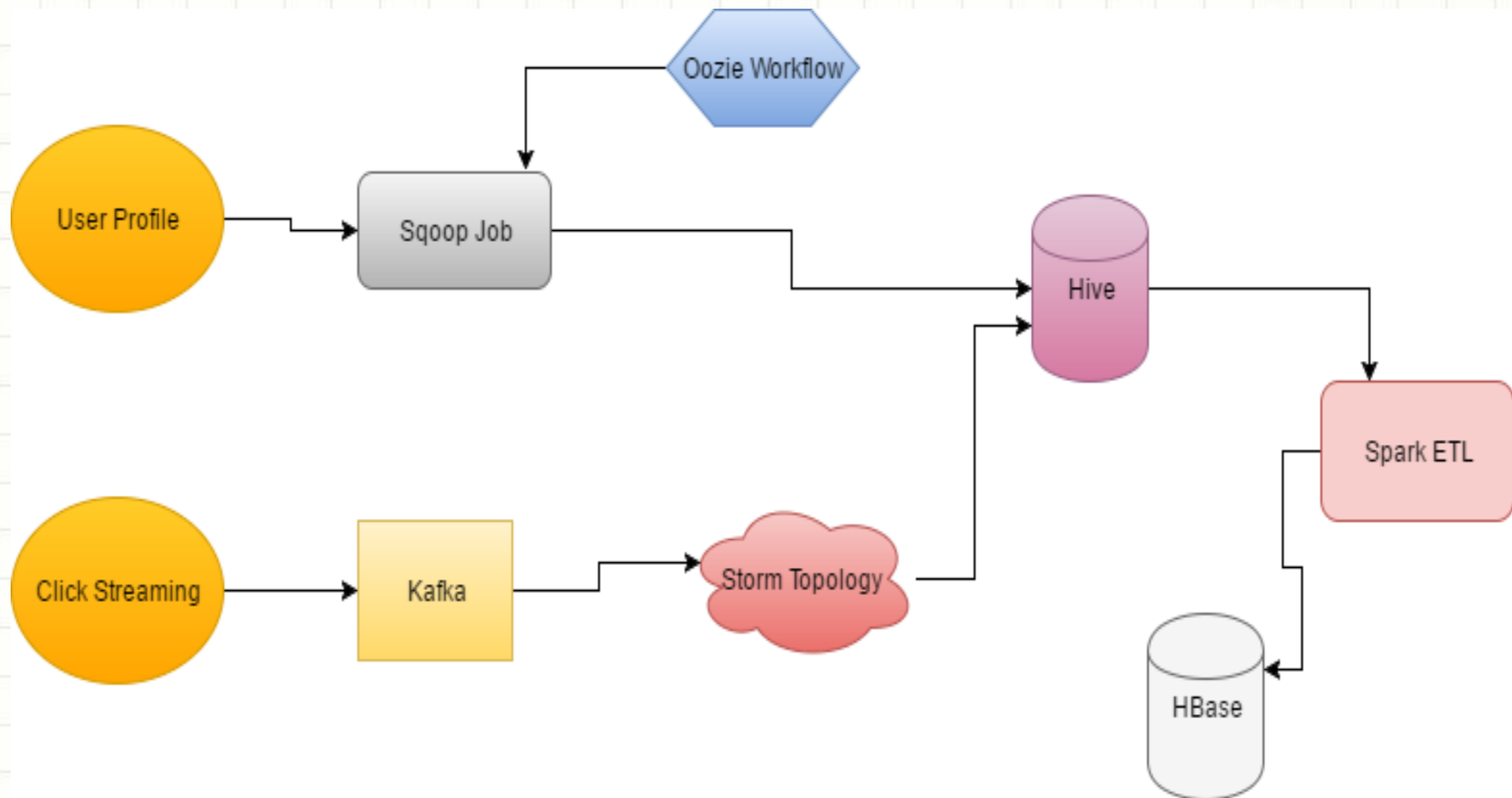
# HBase Use Cases – Insurance Claim

## Near Real-Time Event Processing



# HBase Use Cases – Behavior Tracking

## Master Data Management



# HBase Troubleshooting

## Too Many Regions

- **Problem:**

- Regions are sharing the memstore memory area. Therefore, the more regions there are, the smaller the memstore flushes will be.
- The smaller the generated HFiles will be
- More compaction operations
- Excessive churn on the cluster, affecting performance
- Compaction storm
- Snapshots timeout
- Client operations can timeout (like flush)
- Bulk load timeouts (might be reported as RegionTooBusyException)

# HBase Troubleshooting

## Too Many Regions

- **Causes:**
  - Maximum region size set too low (1G is default)
  - Configuration settings not updated following an HBase upgrade
  - Accidental configuration settings
  - Over-splitting (misuse of the split feature)
  - Improper presplitting

# HBase Troubleshooting

## Too Many Regions

- **Solutions:**
  - Merging some of them together using HBase shell - `merge_region`
  - Merging some of them together using the Java API



# HBase Troubleshooting

## Too Many Regions

- **Prevention:**

- Maximum of regions Size has to be at least 1G. could be as 10G
- Monitor region size using tools
- Key and table design - not abuse the number of column families
- Presplitting

# HBase Troubleshooting

## Too Many Column Families

- **Problem:**

- Impact your application's performance
- Memstore size will be small
- More compaction operations
- Some column families are pretty big while others are pretty small, create unnecessary pressure on those resources

# HBase Troubleshooting

## Too Many Column Families

- **Causes, Solutions and Prevention:**
  - Delete a column family
  - Merge a column family
  - Separate a column family into a new table

# HBase Troubleshooting

## Hotspotting

- **Problems:**

- Some RegionServers become overwhelmed while the other RegionServers are mostly idle

# HBase Troubleshooting

## Hotspotting

- **Causes:**
  - An issue in the key design
  - Monotonically incrementing keys
  - Poorly distributed keys
  - Small reference tables - RegionServers hosting the reference regions will be overwhelmed
  - Applications issues
  - Meta region hotspotting

# HBase Troubleshooting

## Hotspotting

- **Solutions and Prevention:**
  - Key design
  - Presplitting the reference table
  - Distribute this table to all the nodes



# HBase Troubleshooting

## Timeouts

- **Problems:**
  - Full GCs
  - False RegionServer failures
  - Miss the heartbeat

# HBase Troubleshooting

## Timeouts

- **Causes:**
  - Memory fragmentation
  - Improper hardware behaviors or failures

# HBase Troubleshooting

## Timeouts

- **Solutions:**
  - Restart the server

# HBase Troubleshooting

## Timeouts

- **Prevention:**
  - Reduce heap size
  - Off-heap blockCache
  - Using the G1GC algorithm
  - Configure swappiness to 0 or 1
  - Disable environment-friendly features
  - Hardware duplication

# HBase Design Pattern

- Storing information in the table that reflects single entities
  - Forming row keys
  - Why you should not generate keys with the database
  - How and when to use collections
  - Using Phoenix
- Dealing with Large Files
  - Storing files using keys
  - Using UUID
  - What to do when files grow larger
- Time Series Data
  - Using time-based keys to store time series data
  - Avoiding region hotspotting
  - Tall and narrow rows versus wide rows
  - OpenTSDB principles

# HBase Design Pattern

- Denormalization
  - Storing all the objects for a user
  - Dealing with lost usernames and passwords
  - Generating data for performance testing
  - The section tag index
- Advanced Patterns for Data Modeling
  - Many-to-many relationships in HBase
  - Event time data – keeping track of what is going on
  - Dealing with transactions
- Performance Optimization
  - Loading bulk data into HBase
  - Importing data into HBase using MapReduce
  - Importing data from HDFS into HBase
  - Profiling HBase applications
  - Benchmarking or load testing HBase
  - Monitoring HBase



# **APACHE PHOENIX SQL SKIN OVER HBASE**

Bin Jiang

04/01/2017



# Overview

Apache Phoenix is an open source, massively parallel, relational database engine supporting OLTP for Hadoop using Apache HBase as its backing store. It enables developers to access large dataset in real-time with familiar SQL interface.

- Standard SQL and JDBC APIs with full ACID transaction capabilities
- Support for late-bound, schema-on-read with existing data in HBase
- Access data stored and produced in other Hadoop products such as Spark, Hive, Pig, Flume, and MapReduce

# What Phoenix does

- Apache HBase provides random, real time access to data in Hadoop. It's well adopted in the Hadoop ecosystem. Apache Phoenix abstract away the underlying data store by enable you to query the data with standard SQL via JDBC driver. Apache Phoenix provides features such as secondary indexes to help you speed up the queries without relying on specific row key designs
- Apache Phoenix is also massively parallel where aggregation queries are executed on the nodes where data is stored, greatly reduce the need to send data over the network

# Phoenix Features

- Query data with a SQL-based language
- Real-time queries
- Built on top of proven data store Hbase
- Phoenix provides ODBC connector drivers, allowing you to connect to your dataset using familiar BI tools

# How Phoenix works

- Phoenix provides fast access to large amount of data. Full table scan of 100M rows usually completes in 20 seconds (narrow table on a medium sized cluster). This time come down to few milliseconds if query contains filter on key columns. For filters on non-key columns or non-leading key columns, you can add secondary indexes on these columns which leads to performance equivalent to filtering on key column by making copy of table with indexed column(s) part of key
- Phoenix chunks up your query using the region boundaries and runs them in parallel on the client using a configurable number of threads
- The aggregation will be done in a coprocessor on the server-side, collapsing the amount of data that gets returned back to the client rather than returning it all.