

COMP9313: Big Data Management



Lecturer: Xin Cao

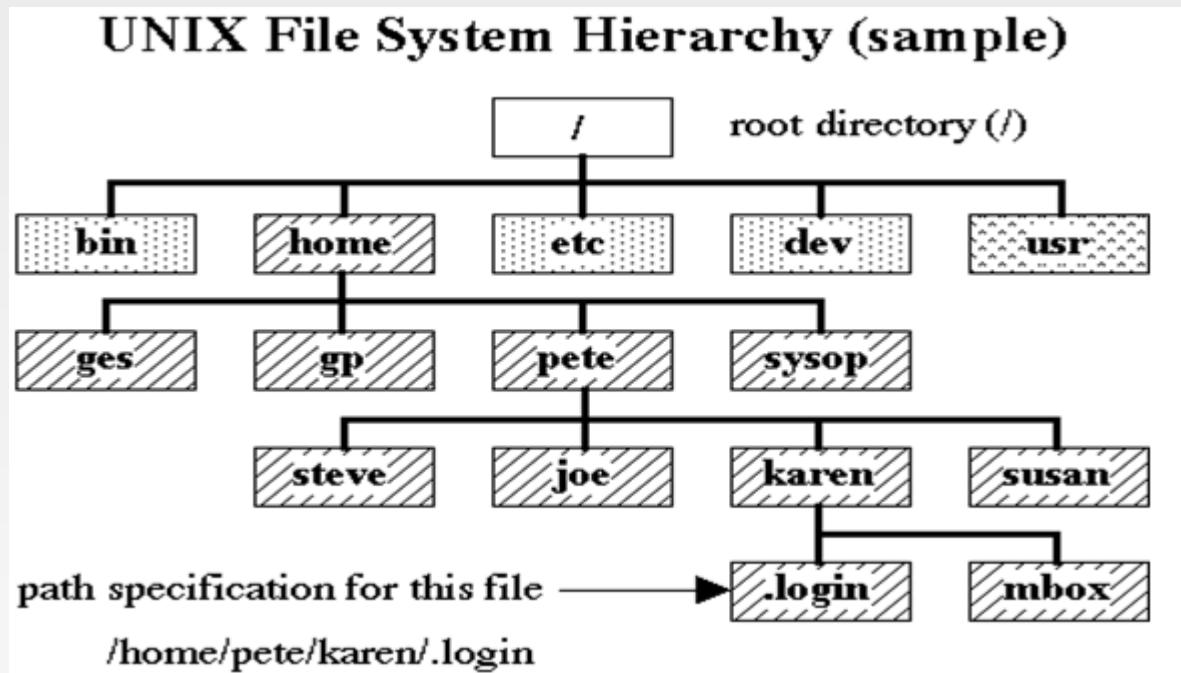
Course web site: <http://www.cse.unsw.edu.au/~cs9313/>

Chapter 11: HDFS, NoSQL, HBase, and Hive

Part 1: HDFS Introduction

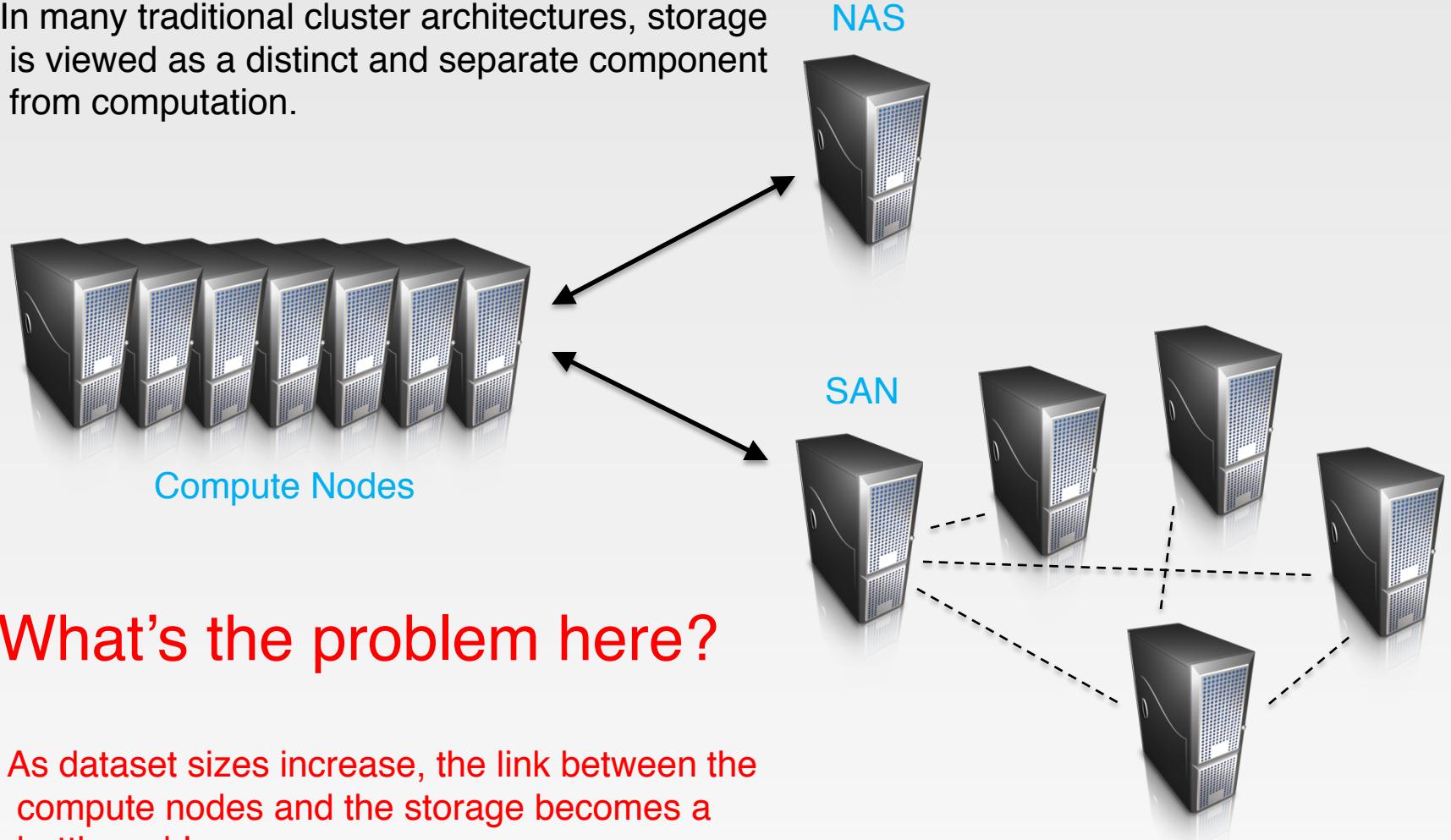
File System

- A filesystem is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk.



How to Move Data to Workers?

In many traditional cluster architectures, storage is viewed as a distinct and separate component from computation.



What's the problem here?

As dataset sizes increase, the link between the compute nodes and the storage becomes a bottleneck!

Distributed File System

- Don't move data to workers... move workers to the data!
 - Store data on the local disks of nodes in the cluster
 - Start up the workers on the node that has the data local
- Why?
 - Not enough RAM to hold all the data in memory
 - Disk access is slow (low-latency), but disk throughput is reasonable (high throughput)
- A distributed file system is the answer
 - A distributed file system is a client/server-based application that allows clients to access and process data stored on the server as if it were on their own computer
 - GFS (Google File System) for Google's MapReduce
 - HDFS (Hadoop Distributed File System) for Hadoop

Latency and Throughput

- **Latency** is the time required to perform some action or to produce some result.
 - Measured in units of time -- hours, minutes, seconds, nanoseconds or clock periods.
 - I/O latency: the time that it takes to complete a single I/O.

- **Throughput** is the number of such actions executed or results produced per unit of time.
 - Measured in units of whatever is being produced (e.g., data) per unit of time.
 - Disk throughput: the maximum rate of sequential data transfer, measured by Mb/sec etc.

Assumptions and Goals of HDFS

- Very large datasets
 - 10K nodes, 100 million files, 10PB
- Streaming data access
 - Designed more for batch processing rather than interactive use by users
 - The emphasis is on high throughput of data access rather than low latency of data access.
- Simple coherency model
 - Built around the idea that the most efficient data processing pattern is a write-once read-many-times pattern
 - A file once created, written, and closed need not be changed except for appends and truncates
- “Moving computation is cheaper than moving data”
 - Data locations exposed so that computations can move to where data resides

Assumptions and Goals of HDFS (Cont')

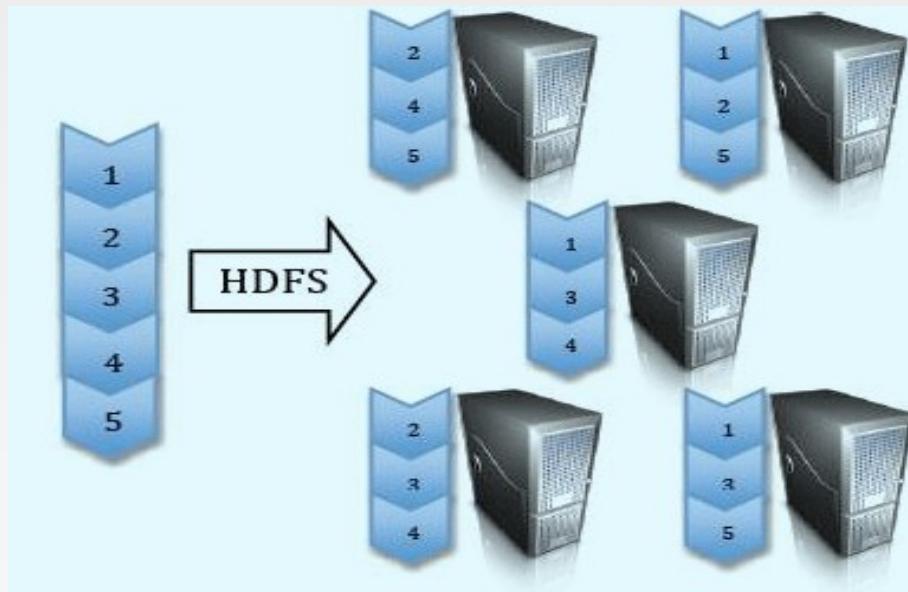
- Assumes Commodity Hardware
 - Files are replicated to handle hardware failure
 - Hardware failure is normal rather than exception. Detect failures and recover from them
- Portability across heterogeneous hardware and software platforms
 - designed to be easily portable from one platform to another
- HDFS is not suited for:
 - Low-latency data access (HBase is a better option)
 - Lots of small files (NameNodes hold metadata in memory)

HDFS Features

- The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware.
- Basic Features:
 - Suitable for applications with large data sets
 - Streaming access to file system data
 - High throughput
 - Can be built out of commodity hardware
 - Highly fault-tolerant

HDFS Architecture

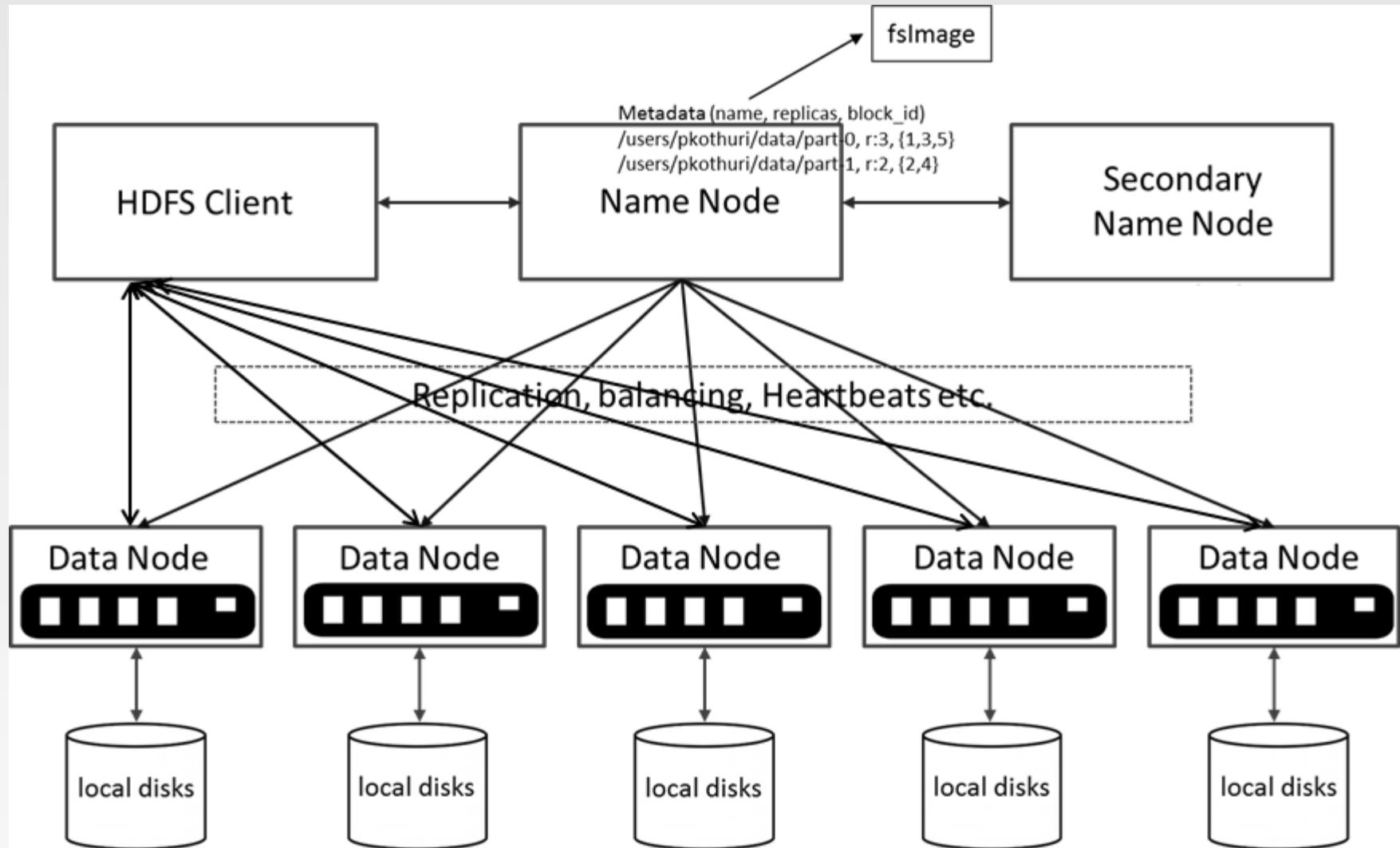
- HDFS is a block-structured file system: Files broken into blocks of 64MB or 128MB
- A file can be made of several blocks, and they are stored across a cluster of one or more machines with data storage capacity.
- Each block of a file is replicated across a number of machines, To prevent loss of data.



HDFS Architecture

- HDFS has a master/slave architecture.
- There are two types (and a half) of machines in a HDFS cluster
 - NameNode: the heart of an HDFS filesystem, it maintains and manages the file system metadata. E.g., what blocks make up a file, and on which datanodes those blocks are stored.
 - ▶ Only one in an HDFS cluster
 - DataNode: where HDFS stores the actual data. Serves read, write requests, performs block creation, deletion, and replication upon instruction from Namenode
 - ▶ A number of DataNodes usually one per node in a cluster.
 - ▶ A file is split into one or more blocks and set of blocks are stored in DataNodes.
 - Secondary NameNode: **NOT** a backup of NameNode!!
 - ▶ Checkpoint node. Periodic merge of Transaction log
 - ▶ Help NameNode start up faster next time

HDFS Architecture



Functions of a NameNode

- Managing the file system namespace:
 - Maintain the namespace tree operations like opening, closing, and renaming files and directories.
 - Determine the mapping of file blocks to DataNodes (the physical location of file data).
 - Store file metadata.
- Coordinating file operations:
 - Directs clients to DataNodes for reads and writes
 - No data is moved through the NameNode
- Maintaining overall health:
 - Collect block reports and heartbeats from DataNodes
 - Block re-replication and rebalancing
 - Garbage collection

NameNode Metadata

- HDFS keeps the entire namespace in RAM, allowing fast access to the metadata.
 - 4GB of local RAM is sufficient
- 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 (EditLog)
 - Records file creations, file deletions etc

Functions of DataNodes

- Responsible for serving read and write requests from the file system's clients.
- Perform block creation, deletion, and replication upon instruction from the NameNode.
- Periodically sends a report of all existing blocks to the NameNode (Blockreport)
- Facilitates Pipelining of Data
 - Forwards data to other specified DataNodes

Communication between NameNode and DataNode

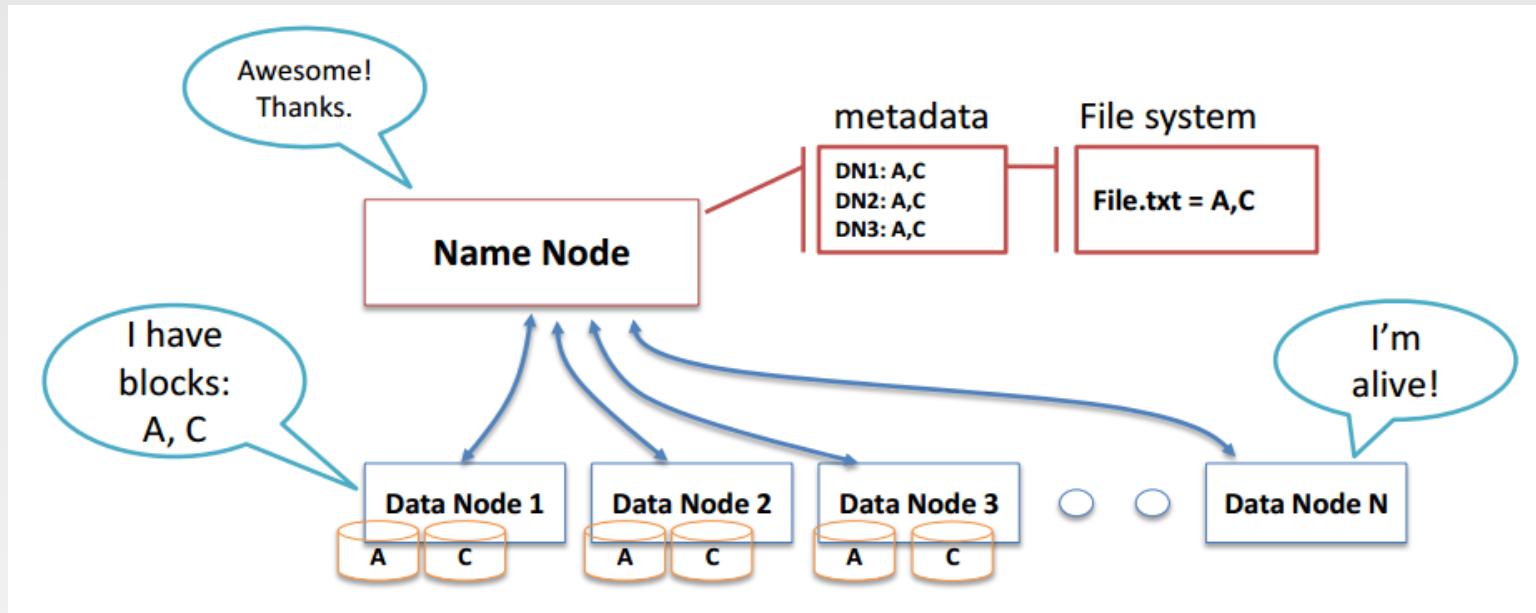
■ Heartbeats

- DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available.
 - ▶ Once every 3 seconds
- The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them

■ Blockreports

- A Blockreport contains a list of all blocks on a DataNode
- The Namenode receives a Heartbeat and a BlockReport from each DataNode in the cluster periodically

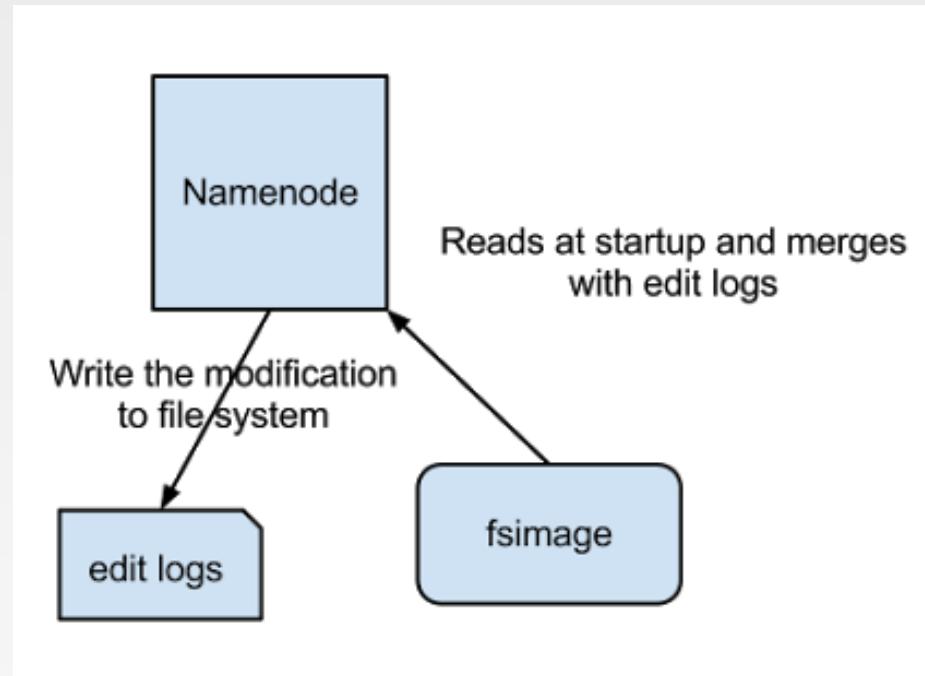
Communication between NameNode and DataDode



- TCP – every 3 seconds a Heartbeat
- Every 10th heartbeat is a Blockreport
- Name Node builds metadata from Blockreports
- If Name Node is down, HDFS is down

Inside NameNode

- FsImage - the snapshot of the filesystem when NameNode started
 - A master copy of the metadata for the file system
- EditLogs - the sequence of changes made to the filesystem after NameNode started

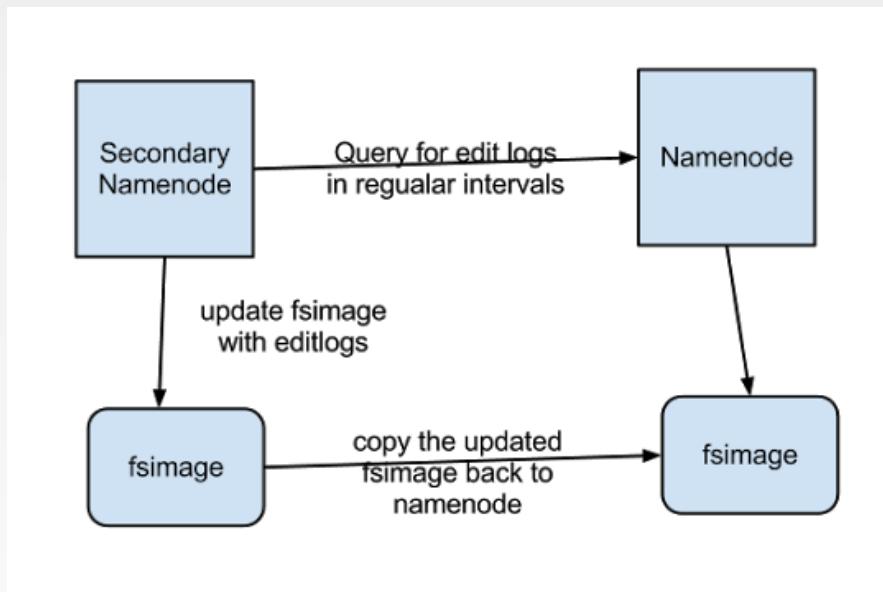


Inside NameNode

- Only in the restart of NameNode, EditLogs are applied to FsImage to get the latest snapshot of the file system.
- But NameNode restart are rare in production clusters which means EditLogs can grow very large for the clusters where NameNode runs for a long period of time.
 - EditLog become very large , which will be challenging to manage it
 - NameNode restart takes long time because lot of changes has to be merged
 - In the case of crash, we will lost huge amount of metadata since FsImage is very old
- How to overcome this issue?

Secondary NameNode

- Secondary NameNode helps to overcome the above issues by taking over responsibility of merging EditLogs with FsImage from the NameNode.
 - It gets the EditLogs from the NameNode periodically and applies to FsImage
 - Once it has new FsImage, it copies back to NameNode
 - NameNode will use this FsImage for the next restart, which will reduce the startup time



File System Namespace

- Hierarchical file system with directories and files
 - /user/comp9313
- Create, remove, move, rename etc.
- NameNode maintains the file system
- Any meta information changes to the file system recorded by the NameNode (EditLog).
- An application can specify the number of replicas of the file needed: replication factor of the file.

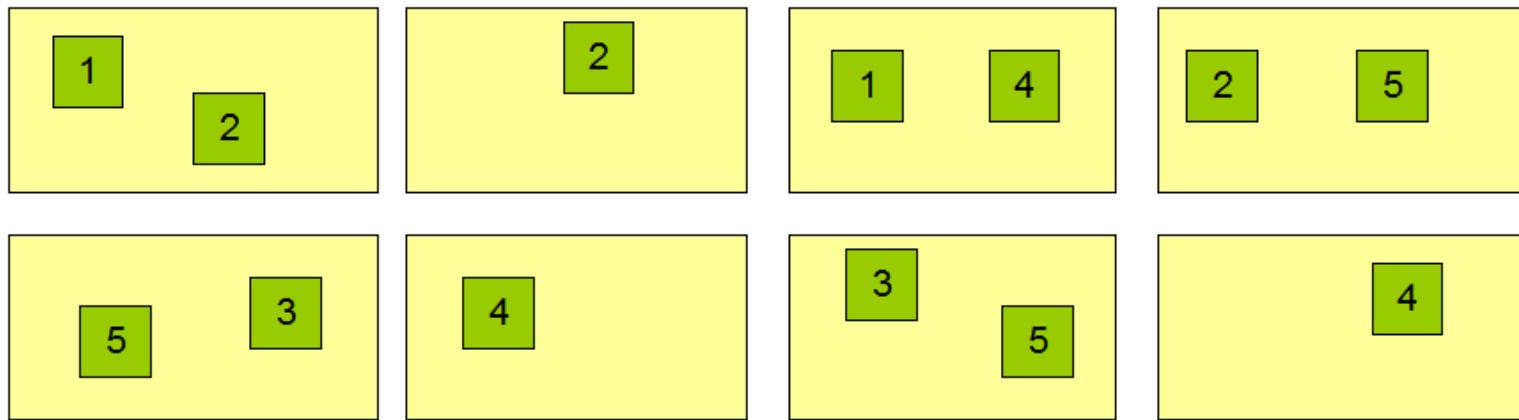
Data Replication

- The NameNode makes all decisions regarding replication of blocks.

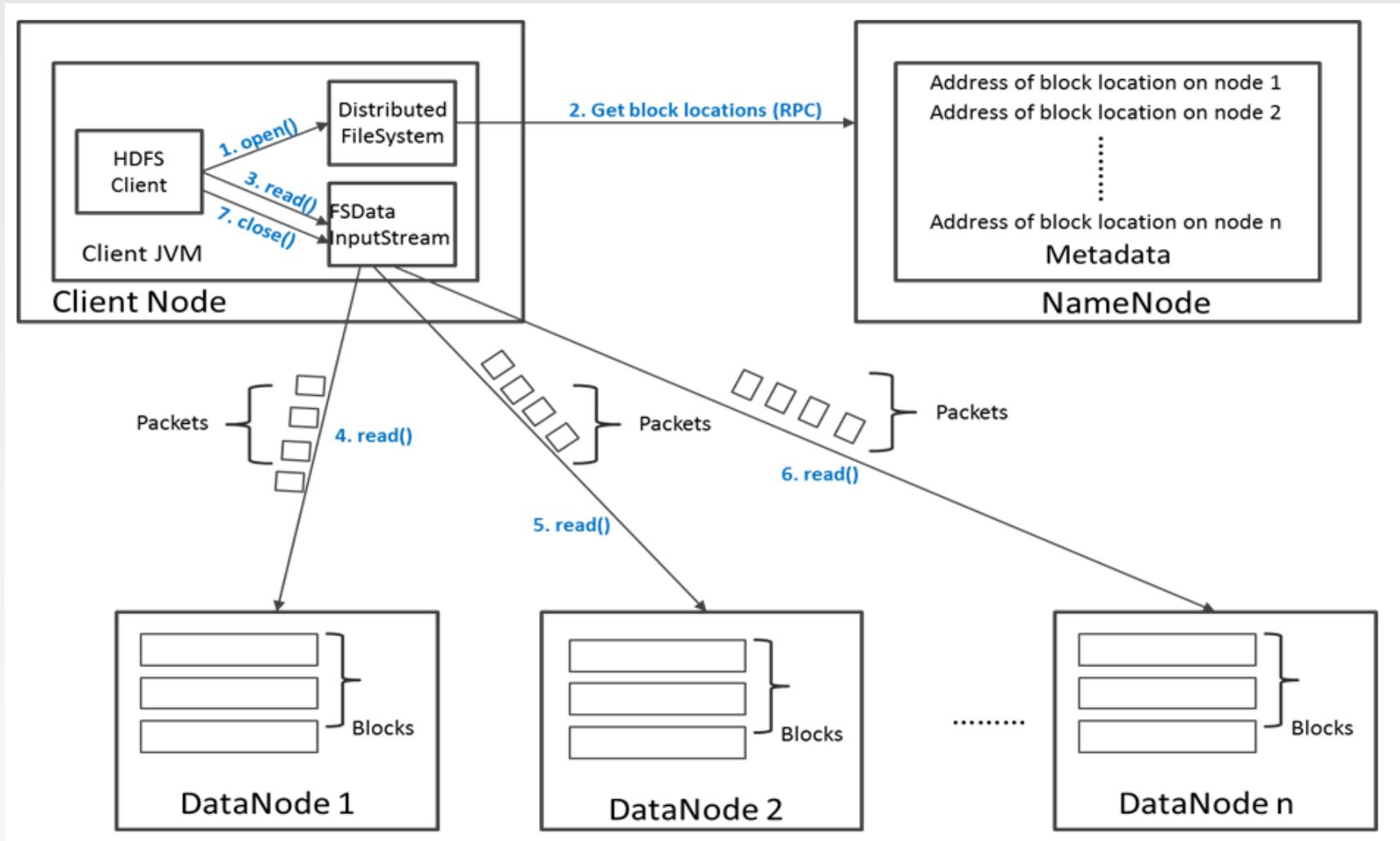
Block Replication

```
Namenode (Filename, numReplicas, block-ids, ...)  
/users/sameerp/data/part-0, r:2, {1,3}, ...  
/users/sameerp/data/part-1, r:3, {2,4,5}, ...
```

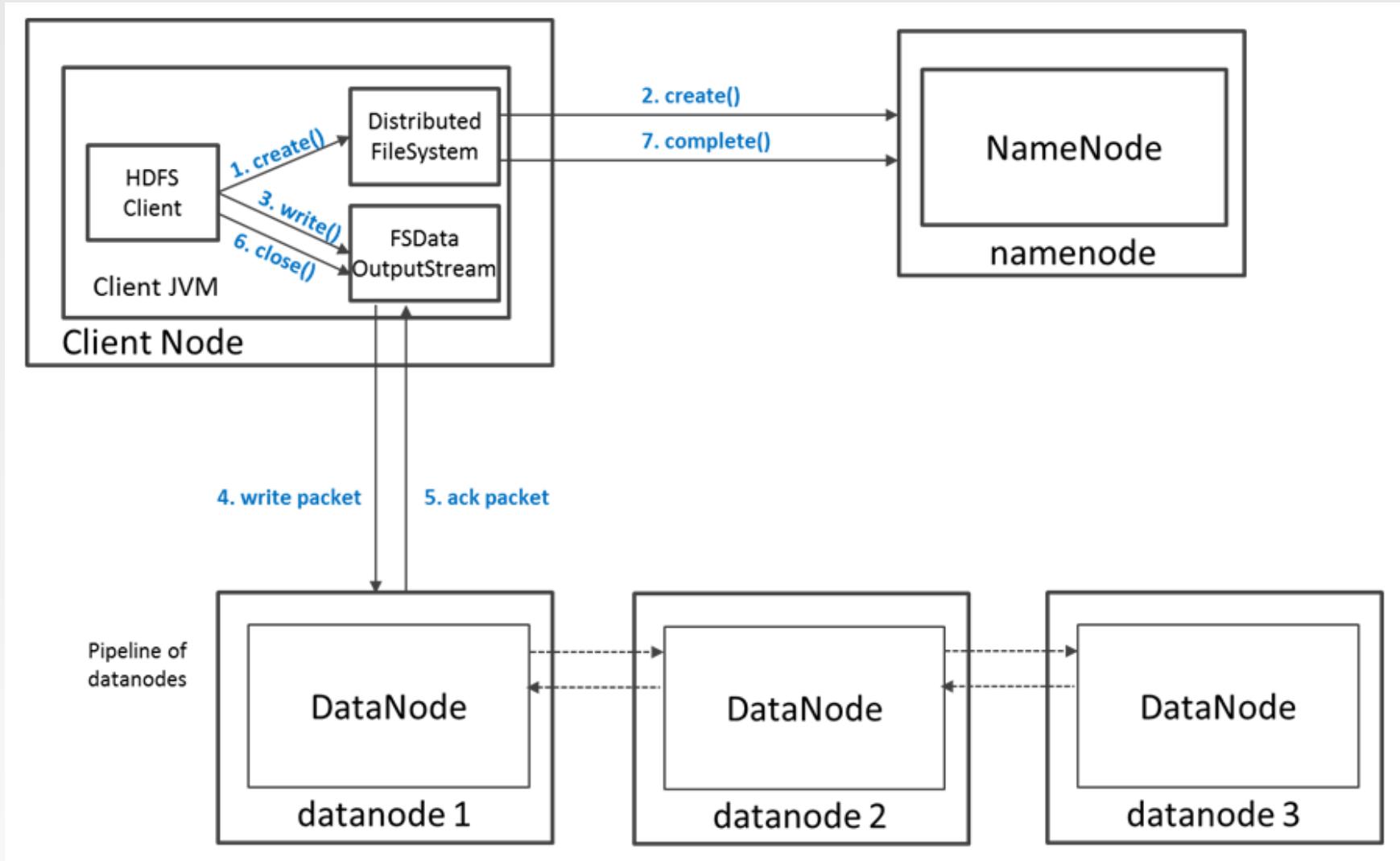
Datanodes



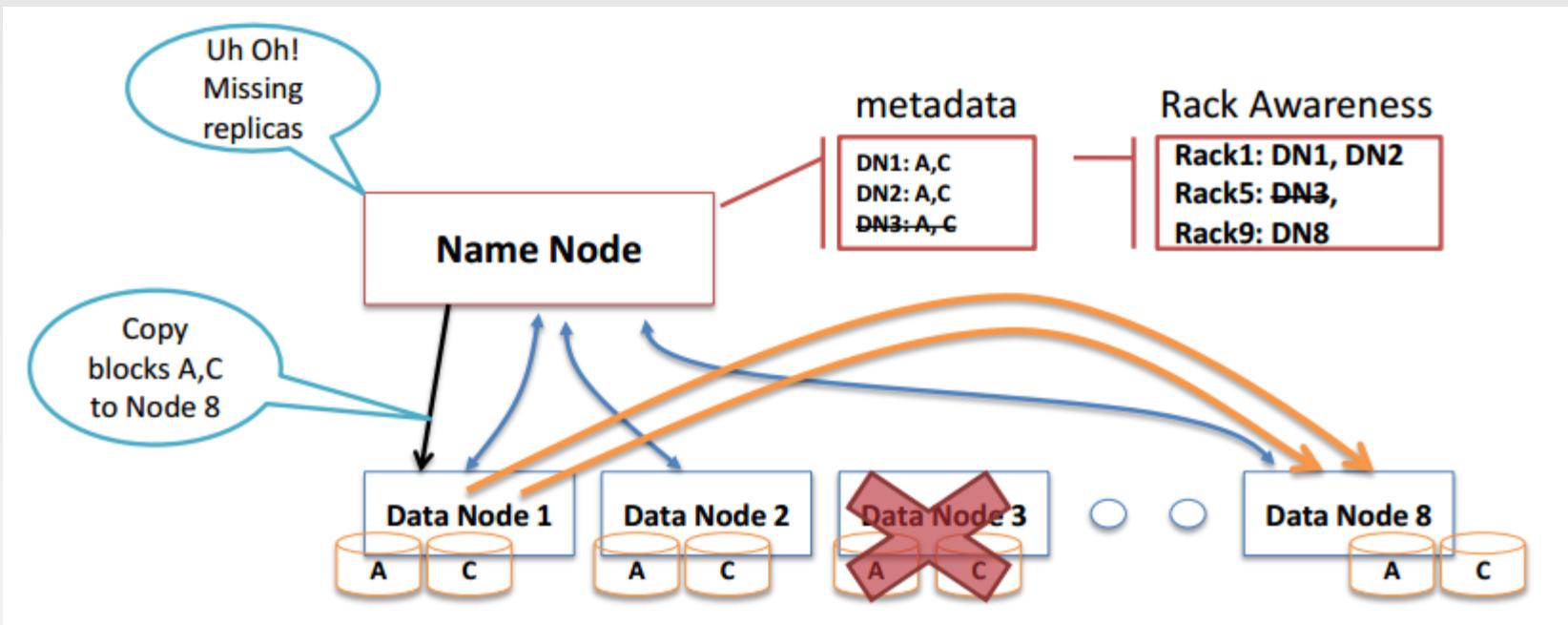
File Read Data Flow in HDFS



File Write Data Flow in HDFS



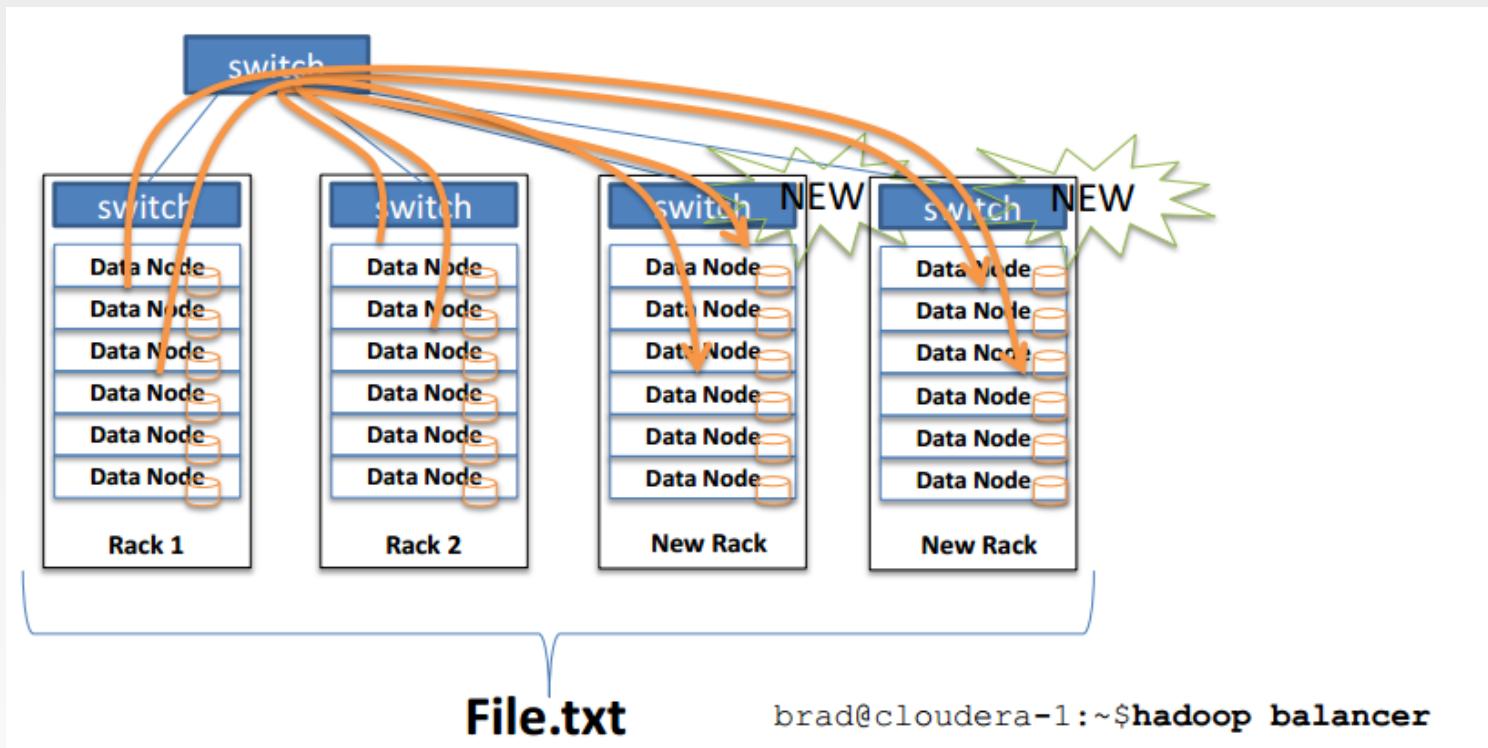
Replication Engine



- NameNode detects DataNode failures
 - Missing Heartbeats signify lost Nodes
 - NameNode consults metadata, finds affected data
 - Chooses new DataNodes for new replicas
 - Balances disk usage
 - Balances communication traffic to DataNodes

Cluster Rebalancing

- Goal: % disk full on DataNodes should be similar
 - Usually run when new DataNodes are added
 - Rebalancer is throttled to avoid network congestion
 - Does not interfere with MapReduce or HDFS
 - Command line tool



Fault tolerance

- Failure is the norm rather than exception
- A HDFS instance may consist of thousands of server machines, each storing part of the file system's data.
- Since we have huge number of components and that each component has non-trivial probability of failure means that there is always some component that is non-functional.
- Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

Metadata Disk Failure

- FslImage and EditLog are central data structures of HDFS. A corruption of these files can cause a HDFS instance to be non-functional.
 - A NameNode can be configured to maintain multiple copies of the FslImage and EditLog
 - Multiple copies of the FslImage and EditLog files are updated synchronously

Unique features of HDFS

- HDFS has a bunch of unique features that make it ideal for distributed systems:
 - Failure tolerant - data is duplicated across multiple DataNodes to protect against machine failures. The default is a replication factor of 3 (every block is stored on three machines).
 - Scalability - data transfers happen directly with the DataNodes so your read/write capacity scales fairly well with the number of DataNodes
 - Space - need more disk space? Just add more DataNodes and re-balance
 - Industry standard - Other distributed applications are built on top of HDFS (HBase, Map-Reduce)
- HDFS is designed to process large data sets with write-once-read-many semantics, it is not for low latency access

Part 2: Introduction to NoSQL

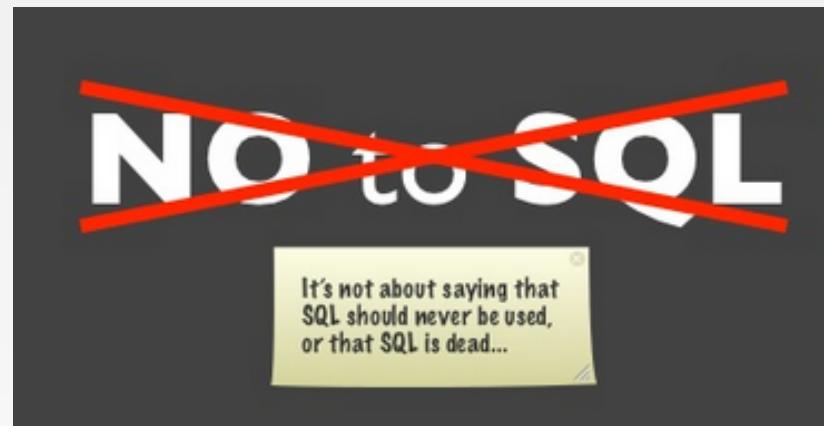
NoSQL

What does RDBMS provide?

- Relational model with schemas
- Powerful, flexible query language (SQL)
- Transactional semantics: ACID
- Rich ecosystem, lots of tool support (MySQL, PostgreSQL, etc.)

What is NoSQL?

- The name stands for **Not Only SQL**
- Does not use SQL as querying language
- Class of non-relational data storage systems
- The term NOSQL was introduced by Eric Evans when an event was organized to discuss open source distributed databases
- It's not a replacement for a RDBMS but compliments it
- All NoSQL offerings relax one or more of the ACID properties (will talk about the CAP theorem)



What is NoSQL?

■ Key features (advantages):

- non-relational
- don't require strict schema
- data are replicated to multiple nodes (so, identical & fault-tolerant) and can be partitioned:
 - ▶ down nodes easily replaced
 - ▶ no single point of failure
- horizontal scalable
- cheap, easy to implement (open-source)
- massive write performance
- fast key-value access



Why NoSQL ?

- Web apps have different needs (than the apps that RDBMS were designed for)
 - Low and predictable response time (latency)
 - Scalability & elasticity (at low cost!)
 - High availability
 - Flexible schemas / semi-structured data
 - Geographic distribution (multiple datacenters)
- Web apps can (usually) do without
 - Transactions / strong consistency / integrity
 - Complex queries

Who are Using NoSQL?

- Google (BigTable)
- LinkedIn (Voldemort)
- Facebook (Cassandra)
- Twitter (HBase, Cassandra)
- Baidu (HyperTable)

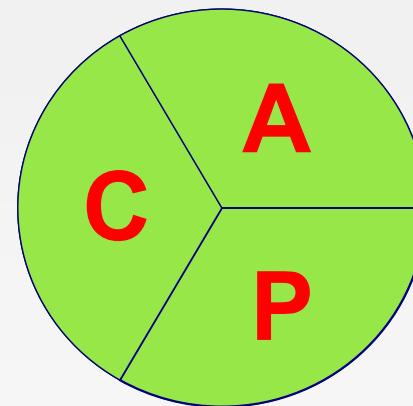


Three Major Papers for NoSQL

- Three major papers were the seeds of the NoSQL movement
 - BigTable (Google)
 - Dynamo (Amazon)
 - ▶ Ring partition and replication
 - ▶ Gossip protocol (discovery and error detection)
 - ▶ Distributed key-value data store
 - ▶ Eventual consistency
 - CAP Theorem (discuss in the next few slides)

CAP Theorem

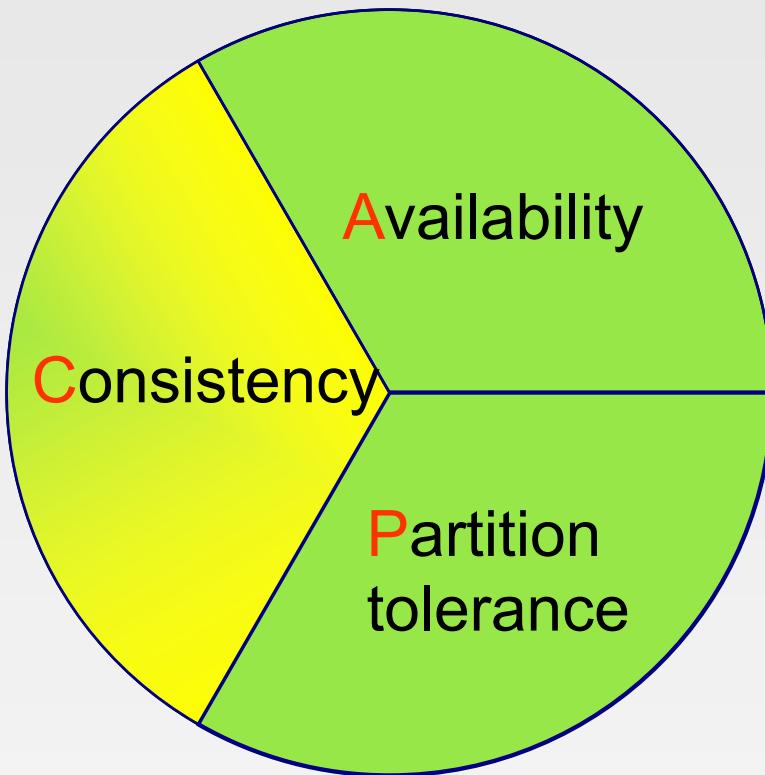
- Suppose three properties of a distributed system (sharing data)
 - **Consistency:**
 - ▶ all copies have same value
 - **Availability:**
 - ▶ reads and writes always succeed
 - **Partition-tolerance:**
 - ▶ system properties (consistency and/or availability) hold even when network failures prevent some machines from communicating with others



CAP Theorem

- Brewer's CAP Theorem:
 - *For any system sharing data, it is “impossible” to guarantee simultaneously all of these three properties*
 - You can have at most two of these three properties for any shared-data system
- Very large systems will “partition” at some point:
 - That leaves either **C** or **A** to choose from (traditional DBMS prefers **C** over **A** and **P**)
 - In almost all cases, you would choose **A** over **C** (except in specific applications such as order processing)

CAP Theorem: Consistency



All clients always have the same view of the data

Once a writer has written, all readers will see that write

- Two kinds of consistency:
 - strong consistency – ACID (Atomicity Consistency Isolation Durability)
 - weak consistency – BASE (Basically Available Soft-state Eventual consistency)

ACID & CAP

■ ACID

- A DBMS is expected to support “ACID transactions,” processes that are:
- **Atomicity:** either the whole process is done or none is
- **Consistency:** only valid data are written
- **Isolation:** one operation at a time
- **Durability:** once committed, it stays that way

■ CAP

- **Consistency:** all data on cluster has the same copies
- **Availability:** cluster always accepts reads and writes
- **Partition tolerance:** guaranteed properties are maintained even when network failures prevent some machines from communicating with others

Consistency Model

- A consistency model determines rules for visibility and apparent order of updates
- Example:
 - Row X is replicated on nodes M and N
 - Client A writes row X to node N
 - Some period of time t elapses
 - Client B reads row X from node M
 - **Does client B see the write from client A?**
 - Consistency is a continuum with tradeoffs
 - **For NOSQL, the answer would be: “maybe”**
 - CAP theorem states: *“strong consistency can't be achieved at the same time as availability and partition-tolerance”*

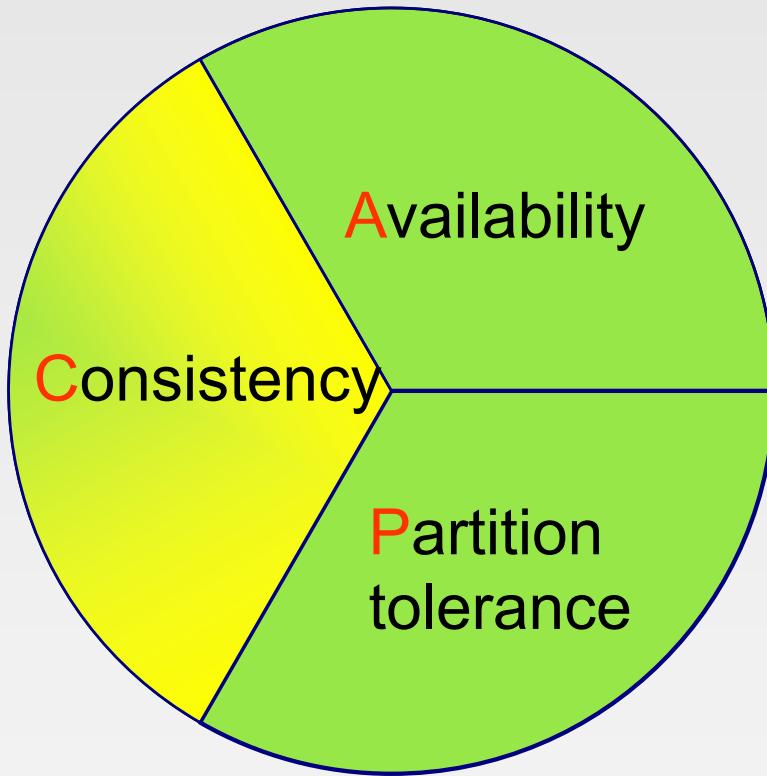
Eventual Consistency

- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
- Known as BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency), as opposed to ACID
- http://en.wikipedia.org/wiki/Eventual_consistency

Eventual Consistency

- The types of large systems based on CAP aren't ACID they are BASE (<http://queue.acm.org/detail.cfm?id=1394128>):
 - Basically Available - system seems to work all the time
 - Soft State - it doesn't have to be consistent all the time
 - Eventually Consistent - becomes consistent at some later time
- Everyone who builds big applications builds them on CAP and BASE: Google, Yahoo, Facebook, Amazon, eBay, etc.

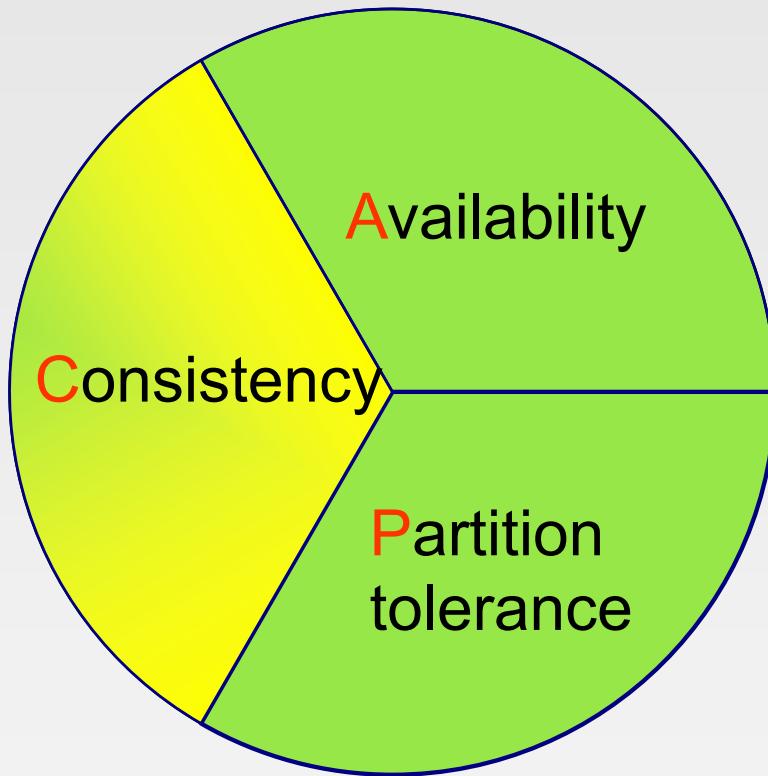
CAP Theorem: Availability



System is available during software and hardware upgrades and node failures.

- Traditionally, thought of as the server/process available five 9's (99.999 %).
 - However, for large node system, at almost any point in time there's a good chance that a node is either down or there is a network disruption among the nodes.
 - Want a system that is resilient in the face of network disruption

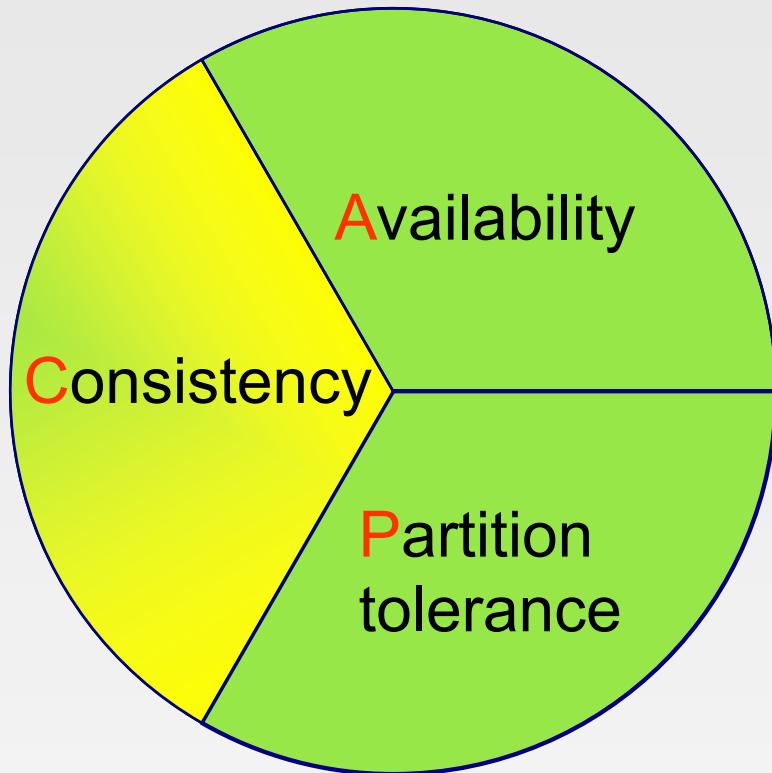
CAP Theorem: Partition-Tolerance



A system can continue to operate in the presence of a network partitions.



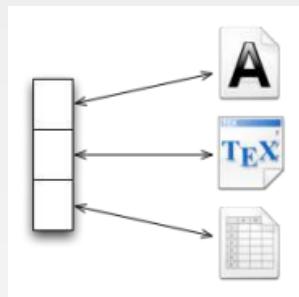
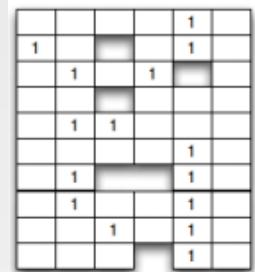
CAP Theorem



CAP Theorem: You can have at most **two** of these properties for any shared-data system

NoSQL Taxonomy

- Key-Value stores
 - Simple K/V lookups (DHT)
- Column stores
 - Each key is associated with many attributes (columns)
 - NoSQL column stores are actually hybrid row/column stores
 - ▶ Different from “pure” relational column stores!
- Document stores
 - Store semi-structured documents (JSON)
- Graph databases
 - Neo4j, etc.
 - Not exactly NoSQL
 - ▶ can't satisfy the requirements for High Availability and Scalability/Elasticity very well



Key-value

- Focus on scaling to huge amounts of data
- Designed to handle massive load
- Based on Amazon's dynamo paper
- Data model: (global) collection of Key-value pairs
- *Dynamo ring partitioning and replication*
- Example: (DynamoDB)
 - *items* having one or more attributes (name, value)
 - An *attribute* can be single-valued or multi-valued like set.
 - items are combined into a *table*

Key-value

■ Basic API access:

- `get(key)`: extract the value given a key
- `put(key, value)`: create or update the value given its key
- `delete(key)`: remove the key and its associated value
- `execute(key, operation, parameters)`: invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map etc)

Key-value

- Pros:
 - very fast
 - very scalable (horizontally distributed to nodes based on key)
 - simple data model
 - eventual consistency
 - fault-tolerance

- Cons
 - Can't model more complex data structure such as objects

Key-value

Name	Producer	Data model	Querying
SimpleDB	Amazon	set of couples (key, {attribute}), where attribute is a couple (name, value)	restricted SQL; select, delete, GetAttributes, and PutAttributes operations
Redis	Salvatore Sanfilippo	set of couples (key, value), where value is simple typed value, list, ordered (according to ranking) or unordered set, hash value	primitive operations for each value type
Dynamo	Amazon	like SimpleDB	simple get operation and put in a context
Voldemort	LinkedIn	like SimpleDB	similar to Dynamo

Document-based

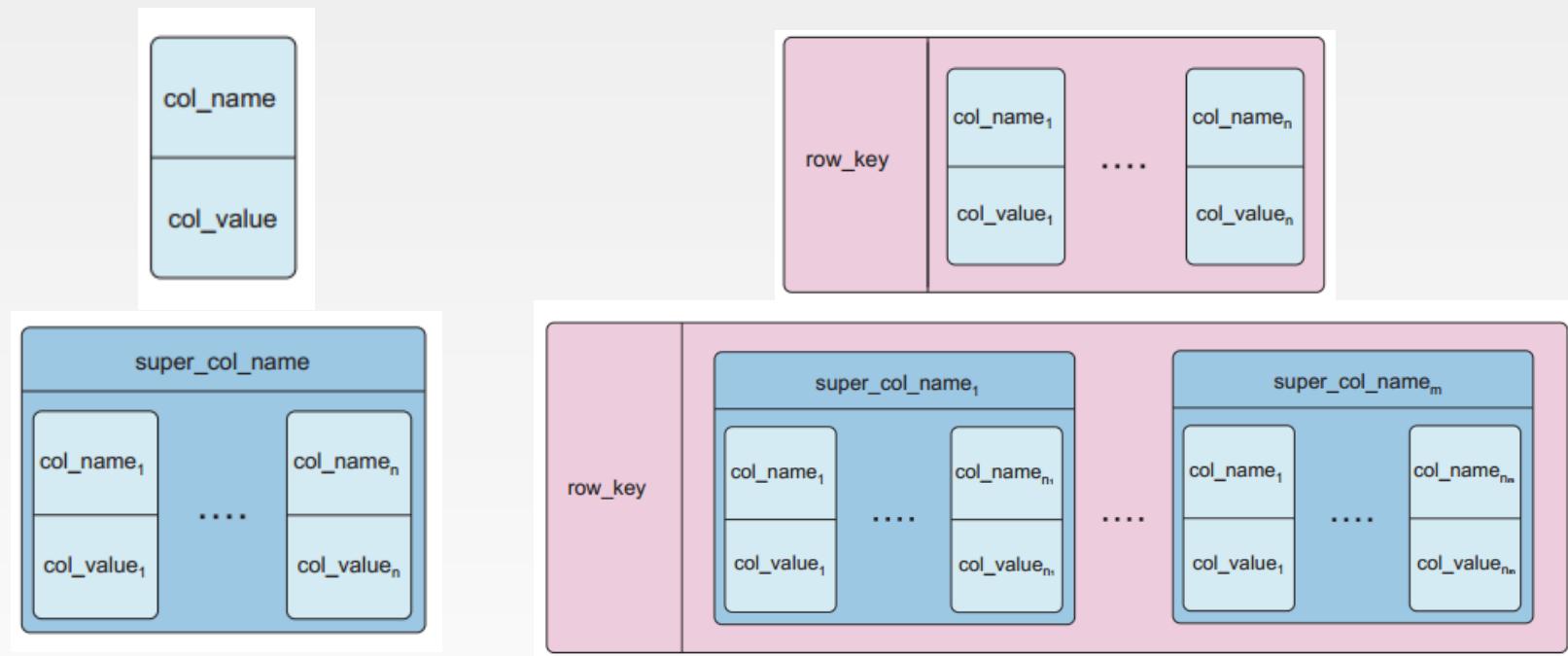
- Can model more complex objects
- Inspired by Lotus Notes
- Data model: collection of documents
- Document: JSON (**JavaScript Object Notation** is a data model, key-value pairs, which supports objects, records, structs, lists, array, maps, dates, Boolean with **nesting**), XML, other semi-structured formats.
- Example: (MongoDB) document
 - {Name:"Jaroslav",
Address:"Malostranske nám. 25, 118 00 Praha 1",
Grandchildren: {Claire: "7", Barbara: "6", "Magda: "3", "Kirsten: "1", "Otis: "3", Richard: "1"},
Phones: ["123-456-7890", "234-567-8963"]
}

Document-based

Name	Producer	Data model	Querying
MongoDB	10gen	object-structured documents stored in collections; each object has a primary key called ObjectId	manipulations with objects in collections (find object or objects via simple selections and logical expressions, delete, update,)
Couchbase	Couchbase	document as a list of named (structured) items (JSON document)	by key and key range, views via Javascript and MapReduce

Column-based

- Based on Google's BigTable paper
- Like column oriented relational databases (store data in column order) but with a twist
- Tables similarly to RDBMS, but handle semi-structured
- Data model:
 - Collection of Column Families
 - Column family = (key, value) where value = set of **related** columns (standard, super)
 - indexed by *row key*, *column key* and *timestamp*



Column-based

- One column family can have variable numbers of columns
- Cells within a column family are sorted “physically”
- Very sparse, most cells have null values
- Comparison: RDBMS vs column-based NoSQL
 - Query on multiple tables
 - ▶ RDBMS: must fetch data from several places on disk and glue together
 - ▶ Column-based NoSQL: only fetch column families of those columns that are required by a query (all columns in a column family are stored together on the disk, so multiple rows can be retrieved in one read operation data locality)

Column-based

Name	Producer	Data model	Querying
BigTable	Google	set of couples (key, {value})	selection (by combination of row, column, and time stamp ranges)
HBase	Apache	groups of columns (a BigTable clone)	JRUBY IRB-based shell (similar to SQL)
Hypertable	Hypertable	like BigTable	HQL (Hypertext Query Language)
CASSANDRA	Apache (originally Facebook)	columns, groups of columns corresponding to a key (supercolumns)	simple selections on key, range queries, column or columns ranges
PNUTS	Yahoo	(hashed or ordered) tables, typed arrays, flexible schema	selection and projection from a single table (retrieve an arbitrary single record by primary key, range queries, complex predicates, ordering, top-k)

Graph-based

- Focus on modeling the structure of data (*interconnectivity*)
- Scales to the complexity of data
- Inspired by mathematical Graph Theory ($G=(E,V)$)
- Data model:
 - (Property Graph) nodes and edges
 - ▶ Nodes may have properties (including ID)
 - ▶ Edges may have labels or roles
 - Key-value pairs on both
- Interfaces and query languages vary
- *Single-step vs path expressions vs full recursion*
- Example:
 - Neo4j, FlockDB, InfoGrid ...

NoSQL Pros/Cons

■ Advantages

- Massive scalability
- High availability
- Lower cost (than competitive solutions at that scale)
- (usually) predictable elasticity
- Schema flexibility, sparse & semi-structured data

■ Disadvantages

- Don't fully support relational features
 - ▶ no join, group by, order by operations (except within partitions)
 - ▶ no referential integrity constraints across partitions
- No declarative query language (e.g., SQL) → more programming
- Eventual consistency is not intuitive to program for
 - ▶ Makes client applications more complicated
- No easy integration with other applications that support SQL
- Relaxed ACID (see CAP theorem later) → fewer guarantees

Conclusion

- NOSQL database cover only a part of data-intensive cloud applications (mainly Web applications)
- Problems with cloud computing:
 - SaaS (Software as a Service or on-demand software) applications require enterprise-level functionality, including ACID transactions, security, and other features associated with commercial RDBMS technology, i.e. NOSQL should not be the only option in the cloud
 - Hybrid solutions:
 - ▶ Voldemort with MySQL as one of storage backend
 - ▶ deal with NOSQL data as semi-structured data
 - >integrating RDBMS and NOSQL via SQL/XML

Part 3: Introduction to HBase



What is HBase?

- HBase is an **open-source**, **distributed**, **column-oriented** database built on top of HDFS based on BigTable
 - Distributed – uses HDFS for storage
 - Row/column store
 - Column-oriented - nulls are free
 - Multi--Dimensional (Versions)
 - Untyped - stores byte[]
- HBase is part of Hadoop
- HBase is the Hadoop application to use when you require real-time read/write random access to very large datasets
 - Aim to support low-latency random access



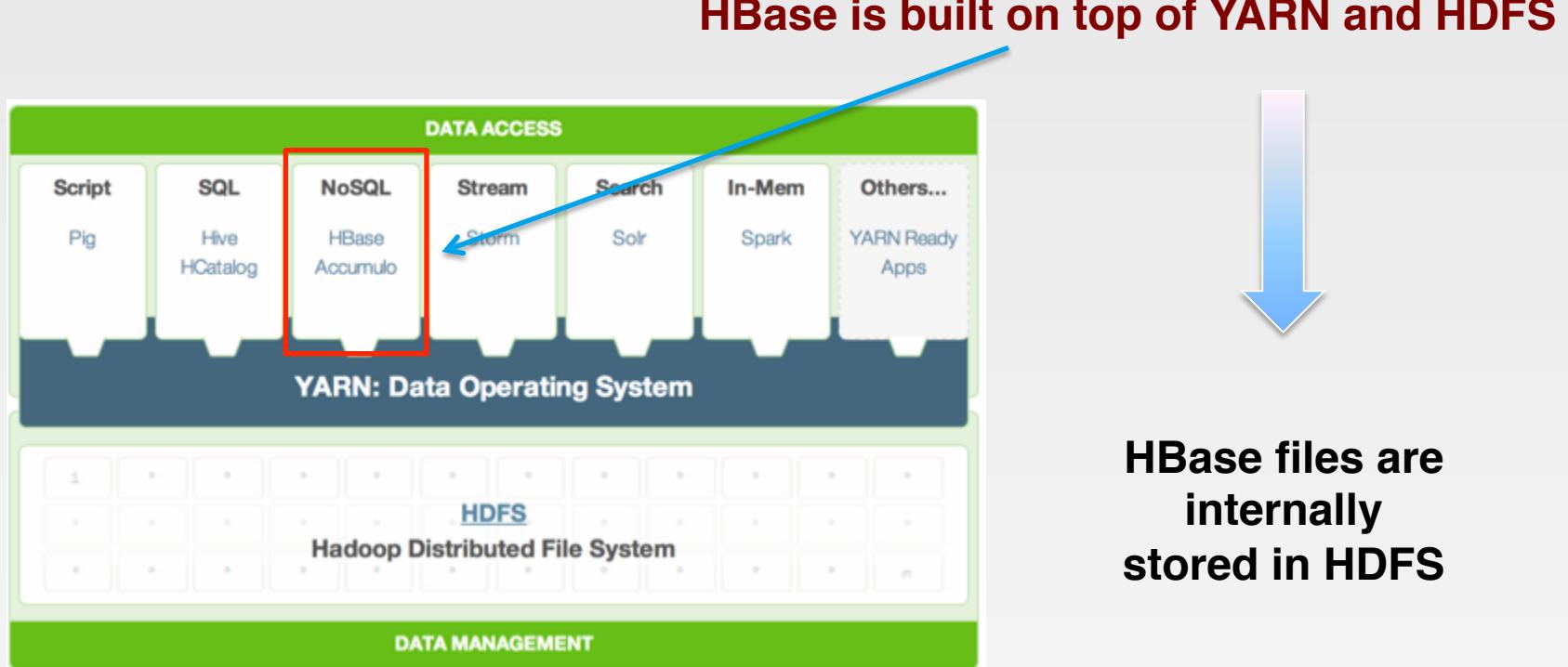
How Data is Stored in HBase ?

- A *sparse*, *distributed*, *persistent* *multi-dimensional sorted* map
- Sparse
 - Sparse data is supported with no waste of costly storage space
 - HBase can handle the fact that we don't (yet) know that information
 - HBase as a schema-less data store; that is, it's fluid — we can add to, subtract from or modify the schema as you go along
- Distributed and persistent
 - Persistent simply means that the data you store in HBase will persist or remain after our program or session ends
 - Just as HBase is an open source implementation of BigTable, HDFS is an open source implementation of GFS.
 - HBase leverages HDFS to persist its data to disk storage.
 - By storing data in HDFS, HBase offers reliability, availability, seamless scalability and high performance — all on cost effective distributed servers.

What is HBase? (Cont’)

- Multi-dimensional sorted map
 - A map (also known as an associative array) is an abstract collection of key-value pairs, where the key is unique.
 - The keys are stored in HBase and sorted.
 - Each value can have multiple versions, which makes the data model multidimensional. By default, data versions are implemented with a timestamp.

HBase: Part of Hadoop's Ecosystem



**HBase files are
internally
stored in HDFS**

HBase vs. HDFS

- Both are distributed systems that scale to hundreds or thousands of nodes
- HDFS is good for batch processing (scans over big files)
 - Not good for record lookup
 - Not good for incremental addition of small batches
 - Not good for updates
- 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 vs. HDFS

	Plain HDFS/MR	HBase
Write pattern	Append-only	Random write, bulk incremental
Read pattern	Full table scan, partition table scan	Random read, small range scan, or table scan
Hive (SQL) performance	Very good	4-5x slower
Structured storage	Do-it-yourself / TSV / SequenceFile / Avro / ?	Sparse column-family data model
Max data size	30+ PB	~1PB

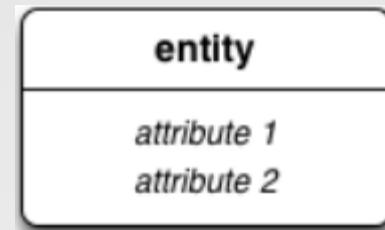
If application has neither random reads or writes → Stick to HDFS

HBase Characteristics

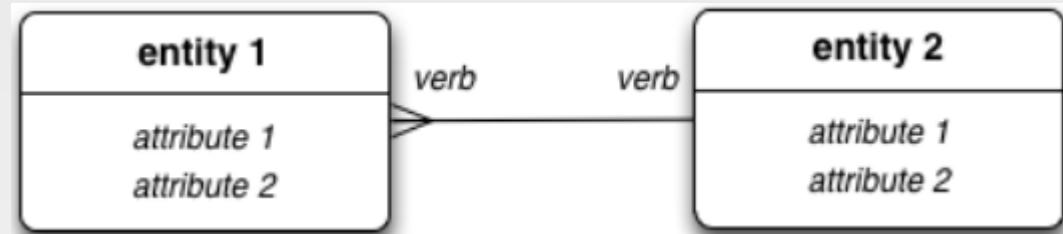
- Tables have one primary index, the *row key*.
- No join operators.
- Scans and queries can select a subset of available columns, perhaps by using a wildcard.
- There are three types of lookups:
 - Fast lookup using row key and optional timestamp.
 - Full table scan
 - Range scan from region start to end.
- Limited atomicity and transaction support.
 - HBase supports multiple batched mutations of single rows only.
 - Data is unstructured and untyped.
- No accessed or manipulated via SQL.
 - Programmatic access via Java, HBase shell, Thrift (Ruby, Python, Perl, C++, ..) etc.

A Review of ER-Model

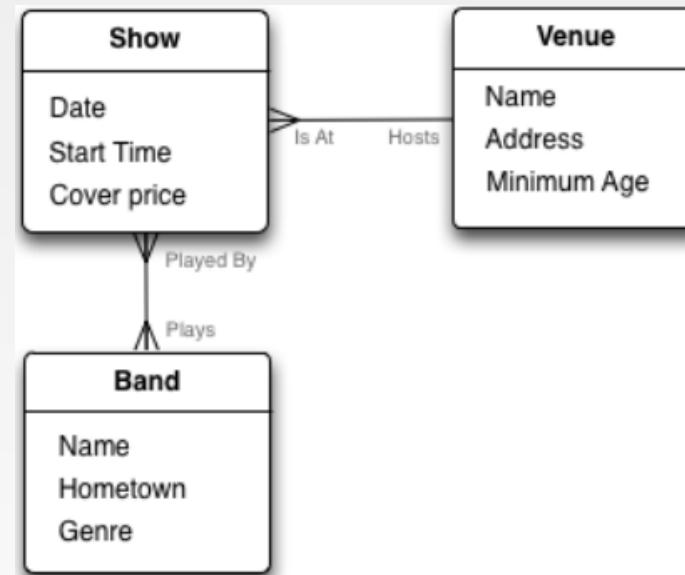
■ Entities



■ Relationships

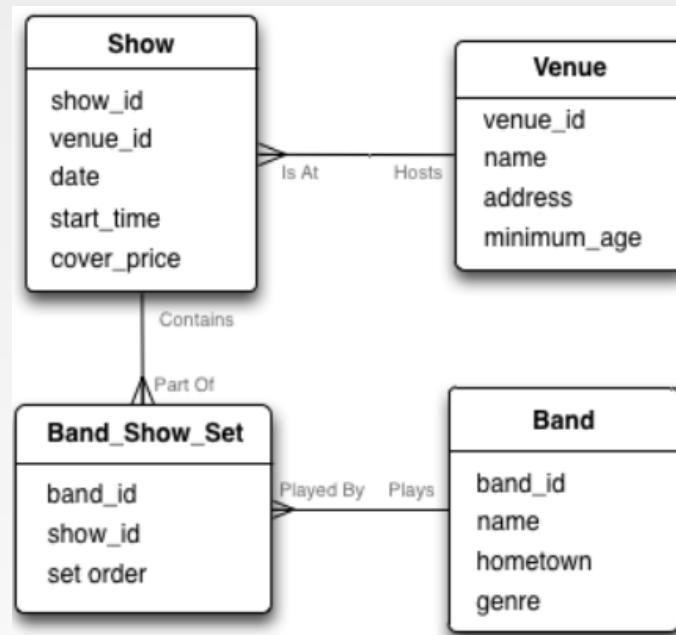


■ Examples: Concerts



Database Normalization

- Entities → Tables
- Attributes → Columns
- Relationships → Foreign Keys
- Many-to-many → Junction tables
- Natural keys → Artificial IDs
- 1NF, 2NF, BCNF, 3NF, 4NF...



Too Big, or Not Too Big

- Two types of data: two big, or not too big
- If data is not too big, a relational database should be used
 - The model is less likely to change as your business needs change. You may want to ask different questions over time, but if you got the logical model correct, you'll have the answers.
- The data is too big?
 - The relational model doesn't acknowledge scale.
 - You need to:
 - ▶ Add indexes
 - ▶ Write really complex, messy SQL
 - ▶ Denormalize
 - ▶ Cache
 - ▶
 - How NoSQL/HBase can help?

HBase Data Model

- **Table:** Design-time namespace, has multiple sorted rows.
- **Row:**
 - Atomic key/value container, with one row key
 - Rows are sorted alphabetically by the row key as they are stored
 - ▶ store data in such a way that related rows are near each other (e.g., a website domain)
- **Column:**
 - A column in HBase consists of a column family and a **column qualifier**, which are delimited by a : (colon) character.
- Table schema only define it's **Column Families**
 - Column families physically co-locate a set of columns and their values
 - ▶ **Column:** a key in the k/v container inside a row
 - ▶ **Value:** a time-versioned value in the k/v container
 - Each column consists of any number of versions
 - Each column family has a set of storage properties, such as whether its values should be cached in memory etc.
 - Columns within a family are sorted and stored together

HBase Data Model (Cont')

■ Column:

- A column qualifier is added to a column family to provide the index for a given piece of data
- Given a column family content, a column qualifier might be content:html, and another might be content:pdf
- Column families are fixed at table creation, but column qualifiers are mutable and may differ greatly between rows.

■ Timestamp: long milliseconds, sorted descending

- A timestamp is written alongside each value, and is the identifier for a given version of a value.
- By default, the timestamp represents the time on the RegionServer when the data was written, but you can specify a different timestamp value when you put data into the cell

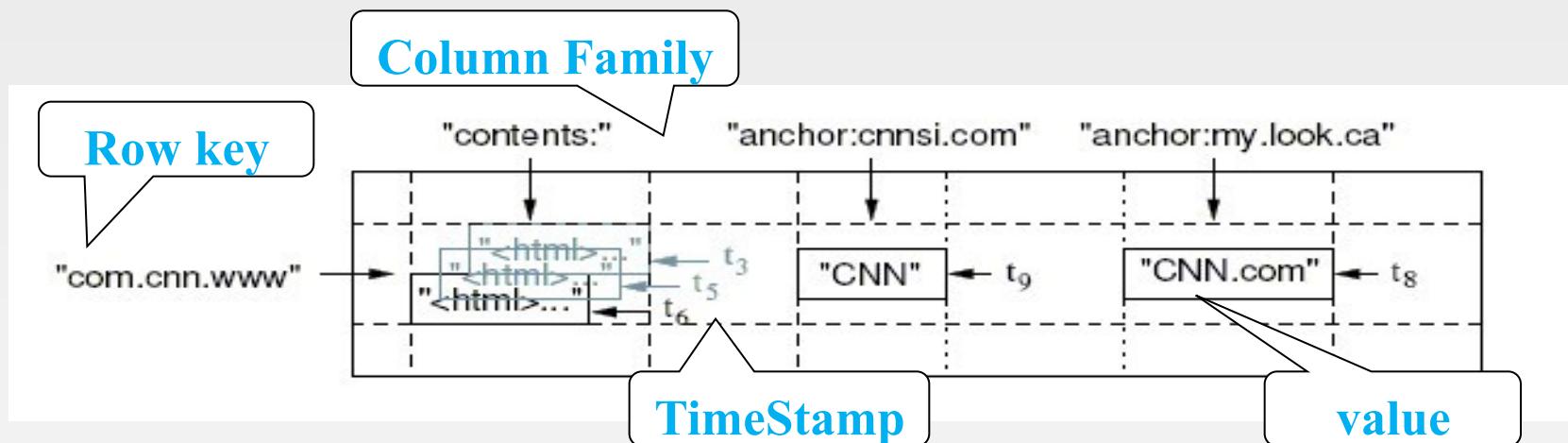
■ Cell:

- A combination of row, column family, and column qualifier, and contains a value and a timestamp, which represents the value's version

■ (Row, Family:<Column, Value>, Timestamp) → Value

HBase Data Model Examples

HBase is based on Google's Bigtable model



HBase Data Model Examples

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

Row key	Time Stamp	Column “content s:”	Column “anchor:”	
“com.apac he.ww w”	t12	“<html> ...”		
	t11	“<html> ...”	Column qualifier	
	t10		“anchor:apache .com”	“APACH E”
	t15		“anchor:cnnsi.co m”	“CNN”
	t13		“anchor:my.look. ca”	“CNN.co m”
	t6	“<html> ...”		
	t5	“<html> ...”		
	t3	“<html> ...”		

HBase Data Model Examples

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

Version number for each row

Row key	Time Stamp	Column “content s:”	Column “anchor:”
“com.apac he.ww w”	t12	“<html> ...”	
	t11	“<html> ...”	value
	t10		“anchor:apache .com”
	t15		“APACH E”
	t13		“CNN”
	t6	“<html> ...”	“CNN.co m”
	t5	“<html> ...”	
	t3	“<html> ...”	

HBase Data Model Examples

Row	Timestamp	Column family: animal:		Column family: repairs:
		animal:type	animal:size	repairs:cost
enclosure1	t2	zebra		1000 EUR
	t1	lion	big	
enclosure2

- Storage: every "cell" (i.e. the time-versioned value of one column in one row) is stored "fully qualified" (with its full rowkey, column family, column name, etc.) on disk

Column family animal:

(enclosure1, t2, animal:type)	zebra
(enclosure1, t1, animal:size)	big
(enclosure1, t1, animal:type)	lion

Column family repairs:

(enclosure1, t1, repairs:cost)	1000 EUR
--------------------------------	----------

HBase Data Model Examples

		Implicit PRIMARY KEY in RDBMS terms	Data is all byte[] in HBase
		Row key	Data
Different types of data separated into different "column families"		cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
		tipcon	info: { 'height': '5ft7', 'state': 'CA' } roles: { 'Hadoop': 'Committer' @ts=2010, 'Hadoop': 'PMC' @ts=2011, 'Hive': 'Contributor' }
Different rows may have different sets of columns(table is sparse)			
			A single cell might have different values at different timestamps
			Useful for *-To-Many mappings

HBase Data Model

- Row:
 - The "row" is atomic, and gets flushed to disk periodically. But it doesn't have to be flushed into just a single file!
 - It can be broken up into different files with different properties, and reads can look at just a subset.
- Column Family: divide columns into physical files
 - Columns within the same family are stored together
 - Why? **Table is sparse, many columns**
 - ▶ No need to scan the whole row when accessing a few columns
 - ▶ Each column a file will generate too many files
- Row keys, column names, values: arbitrary bytes
- Table and column family names: printable characters
- Timestamps: long integers

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' }
tipcon	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

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>..."	

A conceptual view of HBase table

HBase Physical View

- 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

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>..."

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"

HBase Physical Model

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

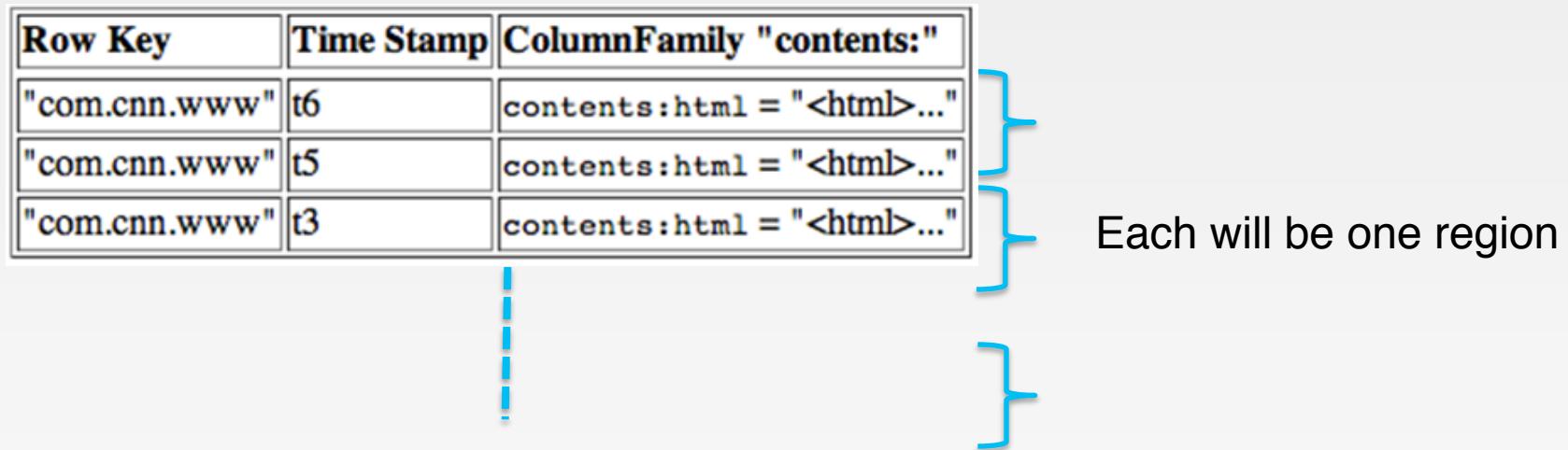
Row key	Column key	Timestamp	Cell value
cutting	roles:ASF	1273871823022	Director
cutting	roles:Hadoop	1183746289103	Founder
tlipcon	roles:Hadoop	1300062064923	PMC
tlipcon	roles:Hadoop	1293388212294	Committer
tlipcon	roles:Hive	1273616297446	Contributor

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

Milliseconds since unix epoch

HBase Physical Model

- Column Families stored separately on disk: access one without wasting I/O on the other
- HBase Regions
 - Each HTable (column family) is partitioned horizontally into *regions*
 - ▶ Regions are counterpart to HDFS blocks

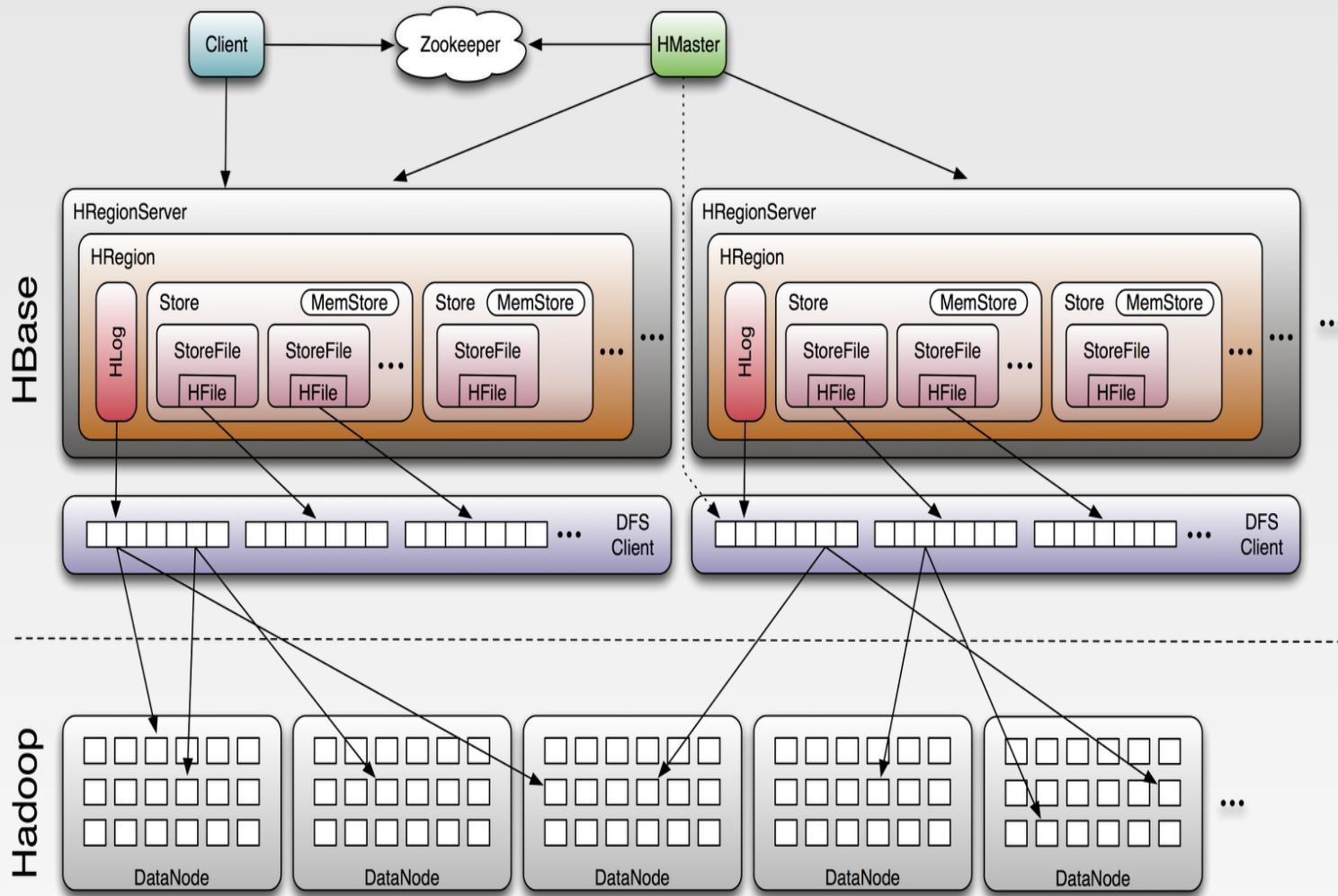


HBase Architecture

■ Major Components

- The MasterServer (HMaster)
 - ▶ One master server
 - ▶ Responsible for coordinating the slaves
 - ▶ Assigns regions, detects failures
 - ▶ Admin functions
- The RegionServer (HRegionServer)
 - ▶ Many region servers
 - ▶ Region (HRegion)
 - A subset of a table's rows, like horizontal range partitioning
 - Automatically done
 - ▶ Manages data regions
 - ▶ Serves data for reads and writes (using a log)
- The HBase client

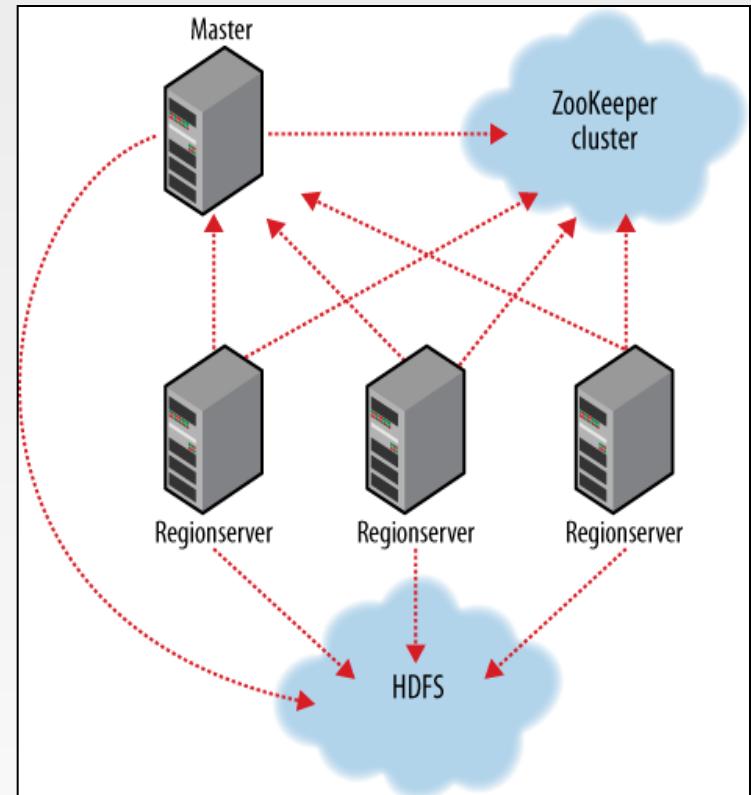
HBase Architecture



ZooKeeper

- HBase clusters can be huge and coordinating the operations of the MasterServers, RegionServers, and clients can be a daunting task, but that's where **Zookeeper** enters the picture.
- Zookeeper is a distributed cluster of servers that collectively provides reliable coordination and synchronization services for clustered applications.

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



Install HBase

- Install Java and Hadoop first
- Download at: <https://hbase.apache.org/>
- The current stable version: 2.0.0
- Install:

```
$ tar xzf hbase- 2.0.0.tar.gz  
$ mv hbase- 2.0.0 ~/hbase
```

- Environment variables in ~/.bashrc

```
export HBASE_HOME = ~/hbase  
export PATH = $HBASE_HOME/bin:$PATH
```

- Edit hbase-env.sh: \$ vim \$HBASE_HOME/conf/hbase-env.sh

```
export JAVA_HOME = /usr/lib/jvm/...  
export HBASE_MANAGES_ZK = true
```

- hbase maintains its own ZooKeeper

Configure HBase as Pseudo-Distributed Mode

- Configure hbase-site.xml: \$ vim \$HBASE_HOME/conf/hbase-site.xml

```
<configuration>
    <property>
        <name>hbase.rootdir</name>
        <value>hdfs://localhost:9000/hbase</value>

    </property>
    <property>
        <name>hbase.cluster.distributed</name>
        <value>true</value>
    </property>
</configuration>
```

- hbase.rootdir: must be consistent with HDFS configuration
- hbase.cluster.distributed: directs HBase to run in distributed mode, with one JVM instance per daemon
- More configurations refer to:
<https://hbase.apache.org/book.html#config.files>

- Start HBase: \$ start-hbase.sh
- Launch the HBase Shell: \$ hbase shell

HBase Shell Commands

■ Create a table

```
hbase(main):004:0> create 'test', 'data'  
0 row(s) in 1.4200 seconds  
=> Hbase::Table - test
```

■ List Information About your Table

```
hbase(main):005:0> list  
TABLE  
test  
1 row(s) in 0.0160 seconds  
=> ["test"]
```

■ Put data into your table

```
hbase(main):006:0> put 'test', 'row1', 'data:1', 'value1'  
0 row(s) in 0.1270 seconds  
  
hbase(main):007:0> put 'test', 'row1', 'data:2', 'value2'  
0 row(s) in 0.0070 seconds  
  
hbase(main):008:0> put 'test', 'row1', 'data:3', 'value3'  
0 row(s) in 0.0040 seconds
```

■ Scan the table for all data at once

```
hbase(main):009:0> scan 'test'  
ROW  
      COLUMN+CELL  
row1      column=data:1, timestamp=1472096331975, value=value1  
row1      column=data:2, timestamp=1472096340030, value=value2  
row1      column=data:3, timestamp=1472096344892, value=value3  
1 row(s) in 0.0670 seconds
```

HBase Shell Commands

■ Describe a table

```
hbase(main):010:0> describe 'test'
Table test is ENABLED
test
COLUMN FAMILIES DESCRIPTION
{NAME => 'data', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATIO
N_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL
=> 'FOREVER', KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536', IN_MEMORY =>
'false', BLOCKCACHE => 'true'}
1 row(s) in 0.0420 seconds
```

■ Get a single row of data

```
hbase(main):012:0> get 'test', 'row1'
COLUMN          CELL
  data:1        timestamp=1472096331975, value=value1
  data:2        timestamp=1472096340030, value=value2
  data:3        timestamp=1472096344892, value=value3
3 row(s) in 0.0250 seconds
```

■ Assign a defined table to a variable; use the variable for operation

```
hbase(main):015:0> t = get_table 'test'
0 row(s) in 0.0030 seconds

=> Hbase::Table - test
```

```
hbase(main):016:0> t.get 'row1'
COLUMN          CELL
  data:1        timestamp=1472096331975, value=value1
  data:2        timestamp=1472096340030, value=value2
  data:3        timestamp=1472096344892, value=value3
3 row(s) in 0.0100 seconds
```

HBase Shell Commands

■ Disable a table

- If you want to delete a table or change its settings, as well as in some other situations, you need to disable the table first

```
hbase(main):020:0> disable 'test'  
0 row(s) in 2.2760 seconds
```

- You can re-enable it using the enable command.

```
hbase(main):021:0> enable 'test'  
0 row(s) in 1.2450 seconds
```

■ Drop (delete) the table

```
hbase(main):023:0> disable 'test'  
0 row(s) in 2.2320 seconds  
  
hbase(main):024:0> drop 'test'  
0 row(s) in 1.2520 seconds  
  
hbase(main):025:0> list  
TABLE  
0 row(s) in 0.0060 seconds
```

■ Exit the HBase Shell

- To exit the HBase Shell and disconnect from your cluster, use the **quit** command. HBase is still running in the background.

HBase Shell Commands

- You can also enter HBase Shell commands into a text file, one command per line, and pass that file to the HBase Shell.

```
create 'test', 'cf'  
list 'test'  
put 'test', 'row1', 'cf:a', 'value1'  
put 'test', 'row2', 'cf:b', 'value2'  
put 'test', 'row3', 'cf:c', 'value3'  
scan 'test'  
get 'test', 'row1'
```

```
TABLE  
test  
1 row(s) in 0.0140 seconds  
  
0 row(s) in 0.3370 seconds  
  
0 row(s) in 0.0040 seconds  
  
0 row(s) in 0.0040 seconds  
  
ROW          COLUMN+CELL  
row1        column=cf:a, timestamp=1472097843848, value=value1  
row2        column=cf:b, timestamp=1472097843869, value=value2  
row3        column=cf:c, timestamp=1472097843874, value=value3  
3 row(s) in 0.0770 seconds  
  
COLUMN          CELL  
  cf:a        timestamp=1472097843848, value=value1  
1 row(s) in 0.0500 seconds
```

HBase benefits than RDBMS

- *No real indexes*
- *Automatic partitioning*
- *Scale linearly and automatically with new nodes*
- *Commodity hardware*
- *Fault tolerance*
- *Batch processing*

HBase vs. RDBMS

	RDBMS	HBase
Data layout	Row-oriented	Column-family-oriented
Transactions	Multi-row ACID	Single row only
Query language	SQL	get/put/scan/etc *
Security	Authentication/Authorization	Work in progress
Indexes	On arbitrary columns	Row-key only
Max data size	TBs	~1PB
Read/write throughput limits	1000s queries/second	Millions of queries/second

When to use HBase

- You need random write, random read, or both (but not neither, otherwise stick to HDFS)
- You need to do many thousands of operations per second on multiple TB of data
- Your access patterns are well-known and simple

Part 4: Introduction to Hive



What is Hive?

- A data warehouse system for Hadoop that
 - facilitates easy data summarization
 - supports ad-hoc queries (still batch though...)
 - created by Facebook
- A mechanism to project structure onto this data and query the data using a SQL-like language – HiveQL
 - Interactive-console –or–
 - Execute scripts
 - Kicks off one or more MapReduce jobs in the background
- An ability to use indexes, built-in user-defined functions

Motivation of Hive

■ Limitation of MR

- Have to use M/R model
- Not Reusable
- Error prone
- For complex jobs:
 - ▶ Multiple stage of Map/Reduce functions
 - ▶ Just like ask developer to write specified physical execution plan in the database

■ Hive intuitive

- Make the unstructured data looks like tables regardless how it really lays out
- SQL based query can be directly against these tables
- Generate specified execution plan for this query

Hive Features

- A subset of SQL covering the most common statements
- Agile data types: Array, Map, Struct, and JSON objects
- User Defined Functions and Aggregates
- Regular Expression support
- MapReduce support
- JDBC support
- Partitions and Buckets (for performance optimization)
- Views and Indexes

Word Count using MapReduce

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WordCount extends Configured implements Tool {

    public static class MapClass extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value,
                        OutputCollector<Text, IntWritable> output,
                        Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer itr = new StringTokenizer(line);
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                output.collect(word, one);
            }
        }
    }

    public static class Reduce extends MapReduceBase
        implements Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values,
                          OutputCollector<Text, IntWritable> output,
                          Reporter reporter) throws IOException {
            int sum = 0;
            while (values.hasNext()) {
                sum += values.next().get();
            }
            output.collect(key, new IntWritable(sum));
        }
    }
}
```

```
public int run(String[] args) throws Exception {
    JobConf conf = new JobConf(getConf(), WordCount.class);
    conf.setJobName("wordcount");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(MapClass.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

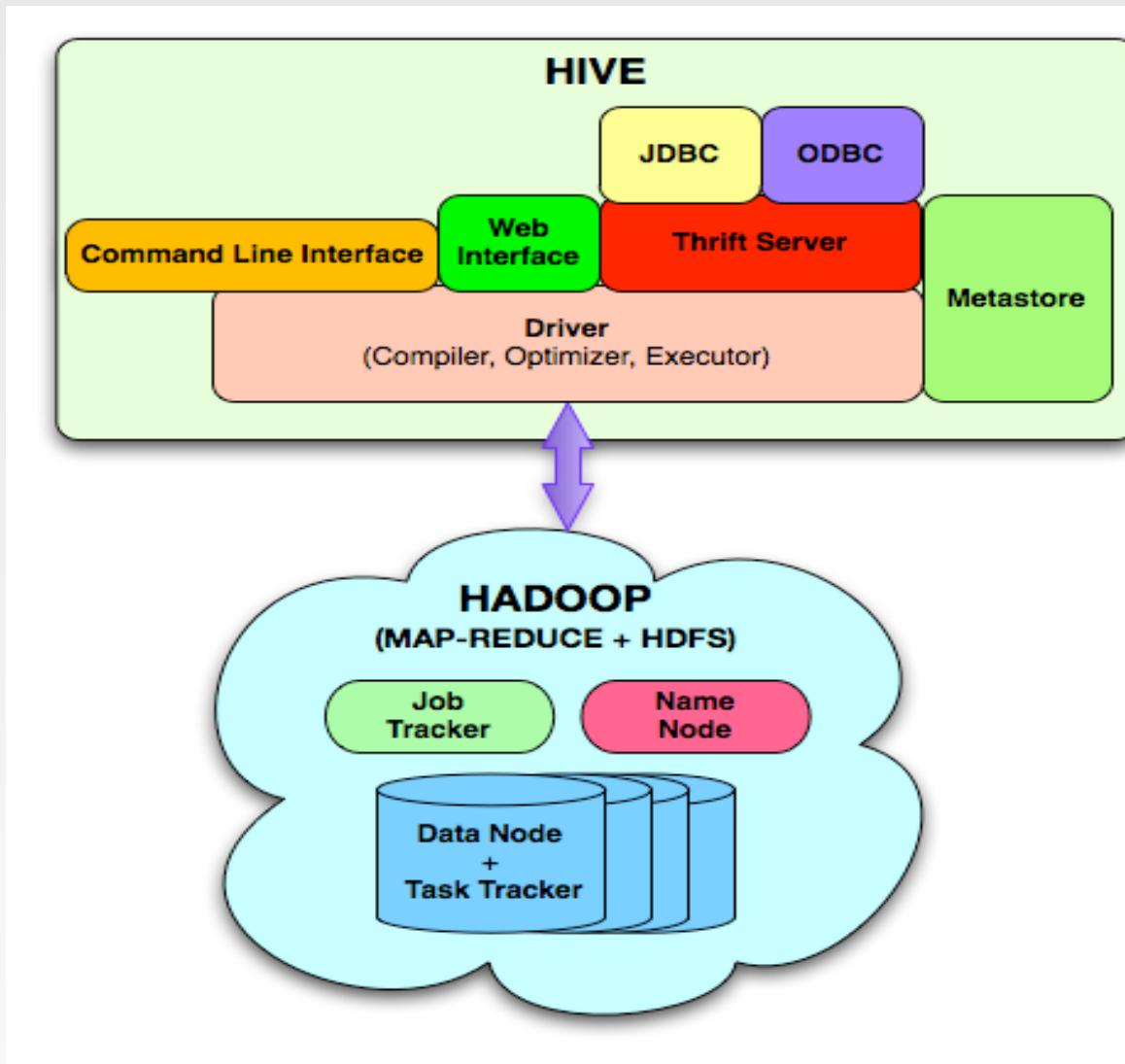
    List<String> other_args = new ArrayList<String>();
    for(int i=0; i < args.length; ++i) {
        try {
            if ("-m".equals(args[i])) {
                conf.setNumMapTasks(Integer.parseInt(args[++i]));
            } else if ("-r".equals(args[i])) {
                conf.setNumReduceTasks(Integer.parseInt(args[++i]));
            } else {
                other_args.add(args[i]);
            }
        } catch (NumberFormatException except) {
            System.out.println("ERROR: Integer expected instead of " + args[i]);
            return printUsage();
        } catch (ArrayIndexOutOfBoundsException except) {
            System.out.println("ERROR: Required parameter missing from " +
                               args[i-1]);
            return printUsage();
        }
    }
    // Make sure there are exactly 2 parameters left.
    if (other_args.size() != 2) {
        System.out.println("ERROR: Wrong number of parameters: " +
                           other_args.size() + " instead of 2.");
        return printUsage();
    }
    FileInputFormat.setInputPaths(conf, other_args.get(0));
    FileOutputFormat.setOutputPath(conf, new Path(other_args.get(1)));
    JobClient.runJob(conf);
    return 0;
}

public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new WordCount(), args);
    System.exit(res);
}
```

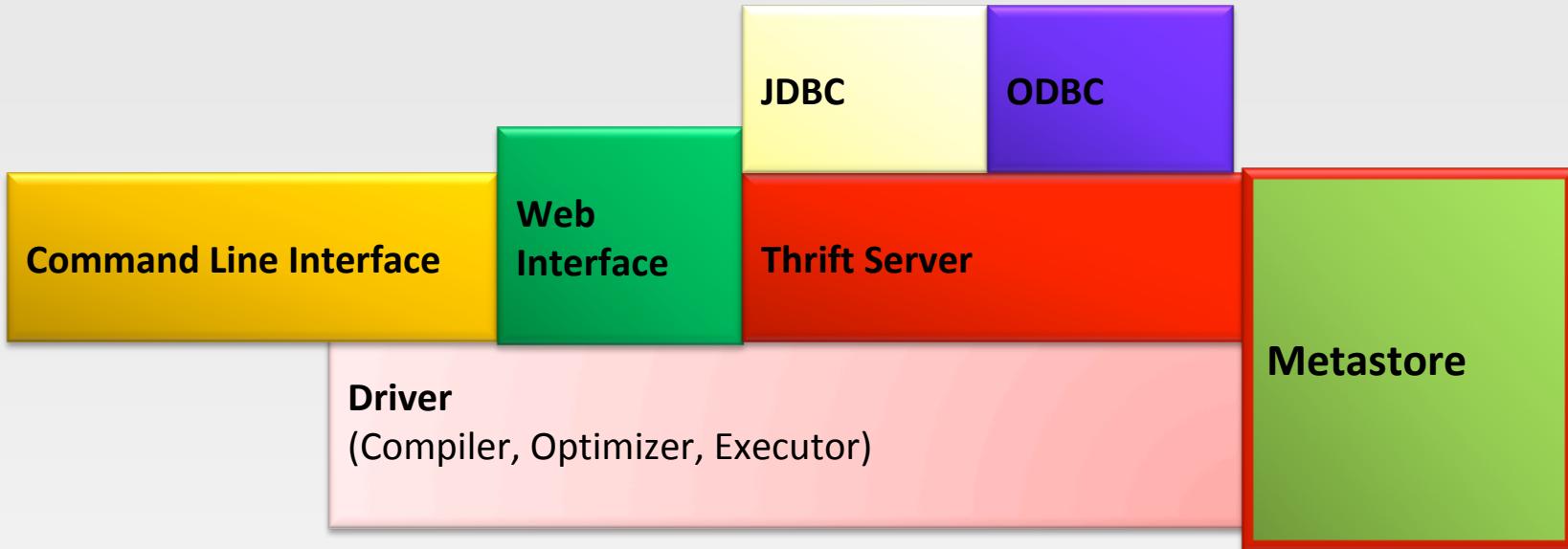
Word Count using Hive

```
create table doc(  
text string  
) row format delimited fields terminated by '\n' stored as textfile;  
  
load data local inpath '/home/Words' overwrite into table doc;  
  
SELECT word, COUNT(*) FROM doc LATERAL VIEW  
explode(split(text, ' ')) Table as word GROUP BY word;
```

Architecture of Hive



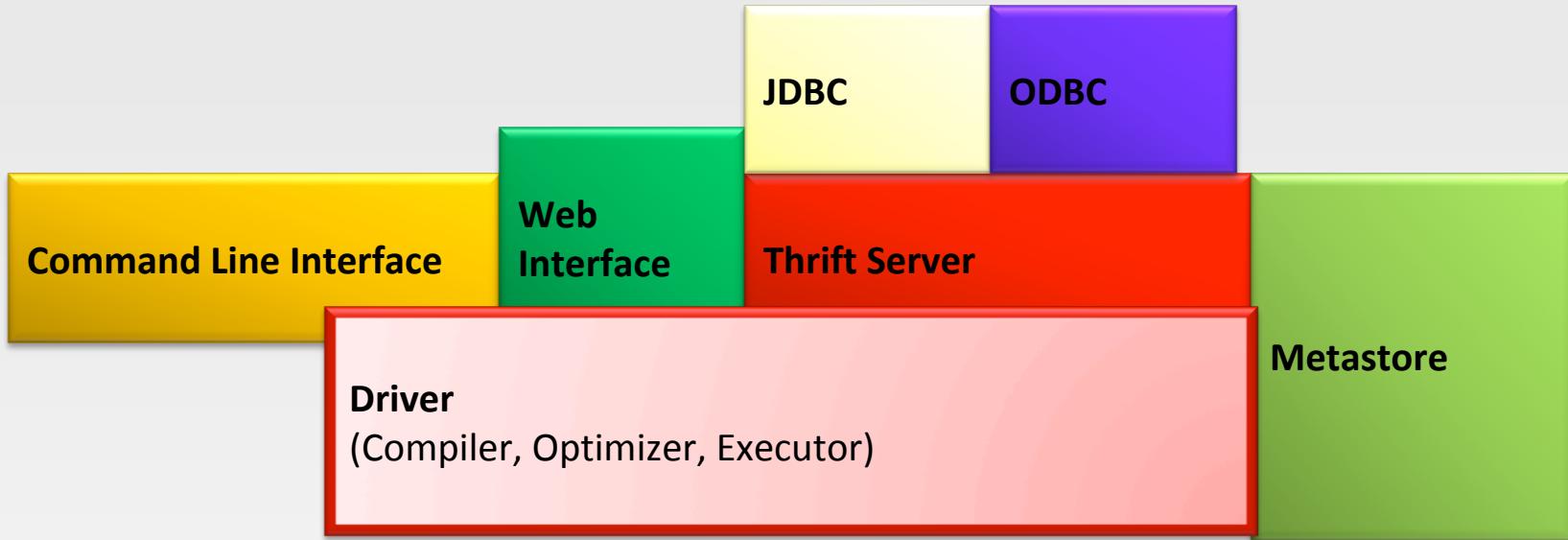
Architecture of Hive



■ Metastore

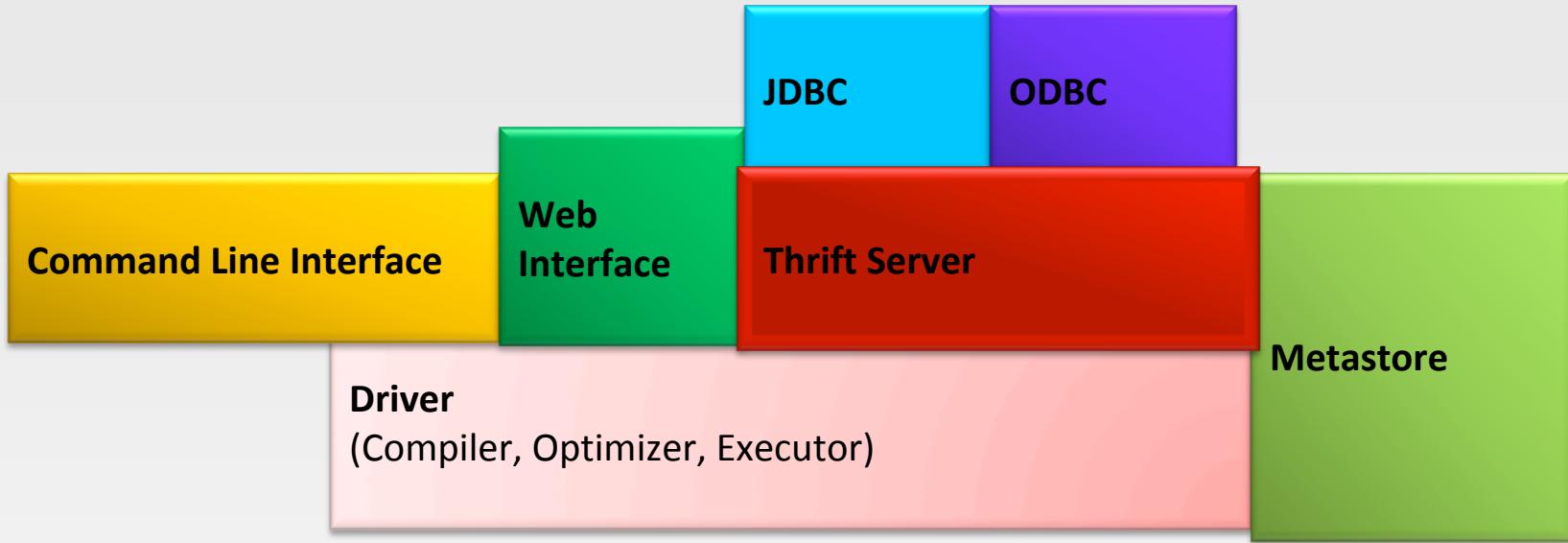
- The component that store the system catalog and meta data about tables, columns, partitions etc.
- Stored in a relational RDBMS (built-in Derby)

Architecture of Hive



- Driver: manages the lifecycle of a HiveQL statement as it moves through Hive.
 - Query Compiler: compiles HiveQL into map/reduce tasks
 - Optimizer: generate the best execution plan
 - Execution Engine: executes the tasks produced by the compiler in proper dependency order. The execution engine interacts with the underlying Hadoop instance.

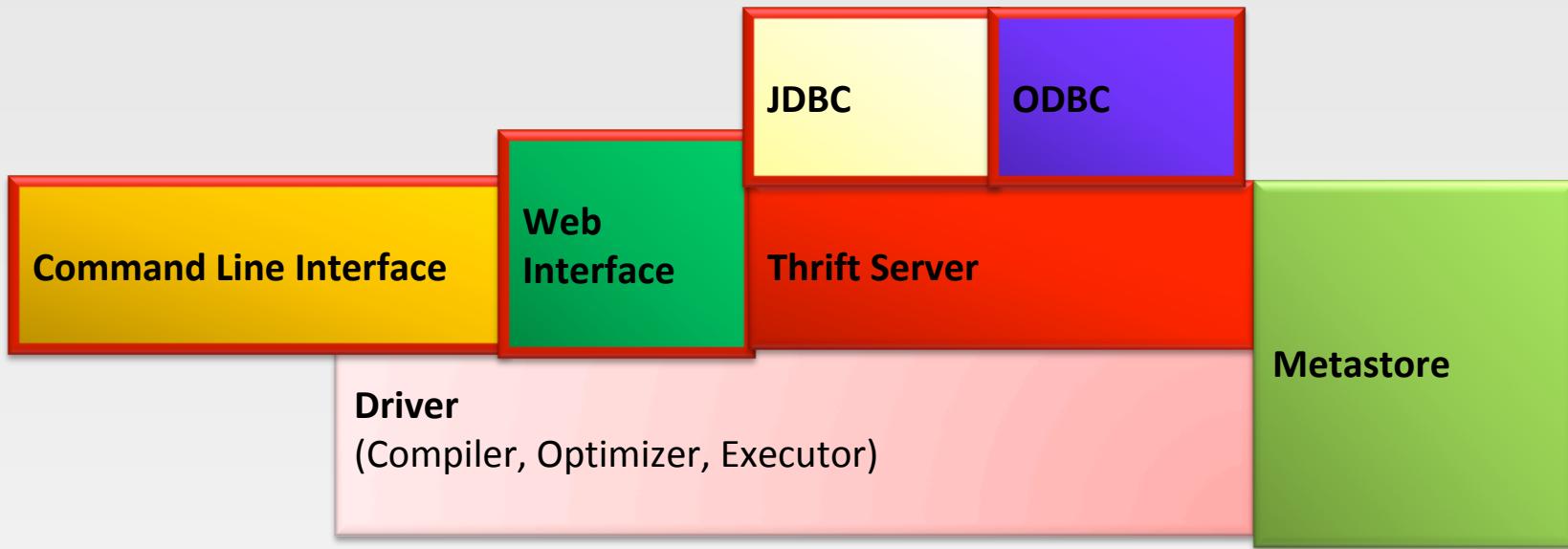
Architecture of Hive



■ Thrift Server

- Cross-language support
- Provides a thrift interface and a JDBC/ODBC server and provides a way of integrating Hive with other applications.

Architecture of Hive



■ Client Components

- Including Command Line Interface(CLI), the web UI and JDBC /ODBC driver.

Hive Installation and Configuration

- Download at: <https://hive.apache.org/downloads.html>
- The latest version: 2.3.3
- Install:

```
$ tar xzf apache-hive-2.3.3-bin.tar.gz  
$ mv apache-hive-2.3.3 ~/hive
```

- Environment variables in ~/.bashrc

```
export HIVE_HOME = ~/hive  
export PATH = $HIVE_HOME/bin:$PATH
```

- Create /tmp and /user/hive/warehouse and set them chmod g+w for more than one user usage

```
$ hdfs dfs -mkdir /tmp  
$ hdfs dfs -mkdir /user/hive/warehouse  
$ hdfs dfs -chmod g+w /tmp  
$ hdfs dfs -chmod g+w /user/hive/warehouse
```

- Run the schematool command to initialize Hive

```
$ schematool -dbType derby -initSchema
```

- Start Hive Shell: \$ hive

Hive Type System

- Primitive types
 - Integers: TINYINT, SMALLINT, INT, BIGINT.
 - Boolean: BOOLEAN.
 - Floating point numbers: FLOAT, DOUBLE.
 - Fixed point numbers: DECIMAL
 - String: STRING, CHAR, VARCHAR.
 - Date and time types: TIMESTAMP, DATE
- Complex types
 - Structs: c has type {a INT; b INT}. c.a to access the first field
 - Maps: M['group'].
 - Arrays: ['a', 'b', 'c'], A[1] returns 'b'.
- Example
 - `list< map<string, struct< p1:int,p2:int > > >`
 - Represents list of associative arrays that map strings to structs that contain two ints

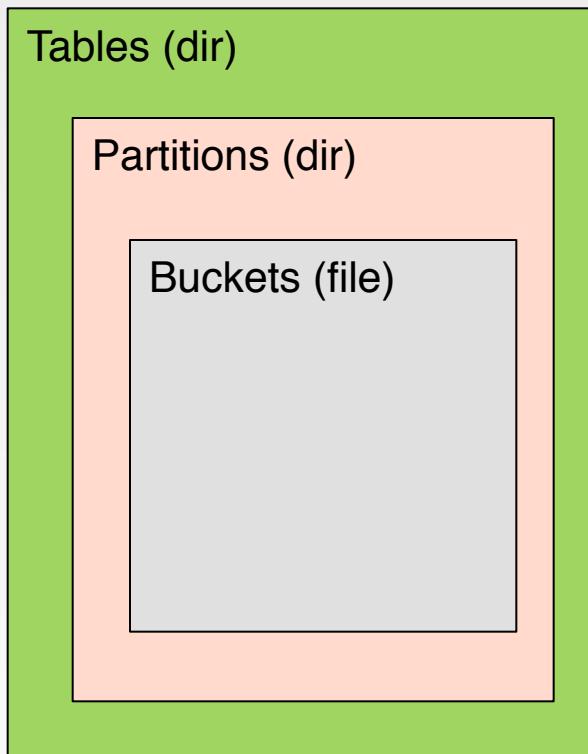
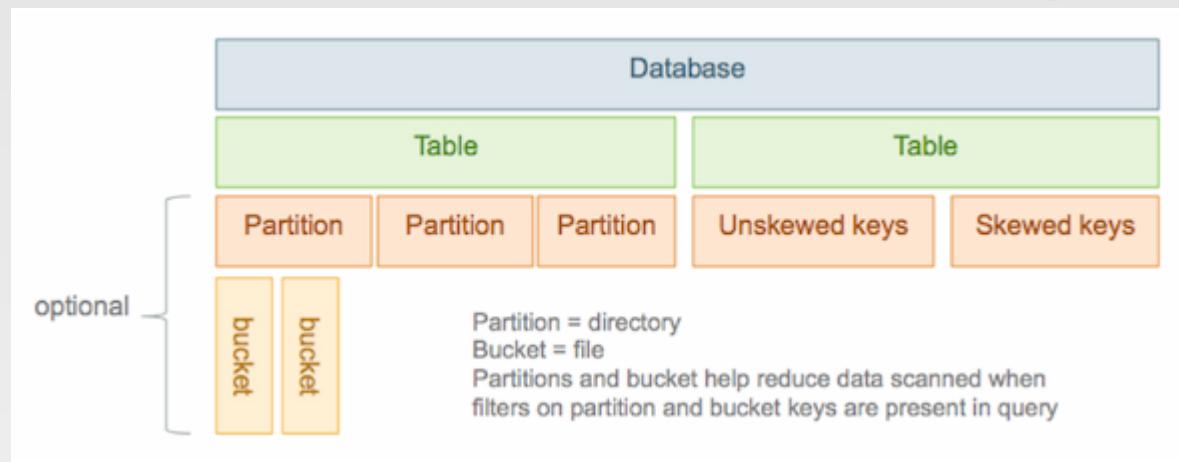
Hive Data Model

- Databases: Namespaces function to avoid naming conflicts for tables, views, partitions, columns, and so on.
- Tables: Homogeneous units of data which have the same schema.
 - Analogous to tables in relational DBs.
 - Each table has corresponding directory in HDFS.
 - An example table: page_views:
 - ▶ timestamp—which is of INT type that corresponds to a UNIX timestamp of when the page was viewed.
 - ▶ userid —which is of BIGINT type that identifies the user who viewed the page.
 - ▶ page_url—which is of STRING type that captures the location of the page.
 - ▶ referer_url—which is of STRING that captures the location of the page from where the user arrived at the current page.
 - ▶ IP—which is of STRING type that captures the IP address from where the page request was made.

Hive Data Model (Cont')

- Partitions:
 - Each Table can have one or more partition Keys which determines how the data is stored
 - Example:
 - ▶ Given the table page_views, we can define two partitions a date_partition of type STRING and country_partition of type STRING
 - ▶ All "US" data from "2009-12-23" is a partition of the page_views table
 - Partition columns are virtual columns, they are not part of the data itself but are derived on load
 - It is the user's job to guarantee the relationship between partition name and data content
- Buckets: Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of the Table
 - Example: the page_views table may be bucketed by userid

Data Model and Storage



/root-path

/table1

/partition1
(2011-11)

- /bucket1 (1/3)
- /bucket2 (2/3)
- /bucket3 (3/3)

/partition2
(2011-12)

- /bucket1 (1/3)
- /bucket2 (2/3)
- /bucket3 (3/3)

/table2

- /bucket1 (1/2)
- /bucket2 (2/2)

Create Table

■ Syntax:

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name  
[(col_name data_type [COMMENT col_comment], ...)]  
[COMMENT table_comment]  
[PARTITIONED BY (col_name data_type [COMMENT  
col_comment], ...)]  
[CLUSTERED BY (col_name, col_name, ...) [SORTED BY  
(col_name [ASC|DESC], ...)] INTO num_buckets  
BUCKETS]  
[ROW FORMAT row_format]  
[STORED AS file_format]
```

See full CREATE TABLE command at:

<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL>

Hive SerDe

- SerDe is a short name for "Serializer and Deserializer."
 - Describe how to load the data from the file into a representation that make it looks like a table;
- Hive uses SerDe (and FileFormat) to read and write table rows.
- HDFS files --> InputFileFormat --> <key, value> --> Deserializer --> Row object
- Row object --> Serializer --> <key, value> --> OutputFileFormat --> HDFS files
- More details see:

<https://cwiki.apache.org/confluence/display/Hive/DeveloperGuide#DeveloperGuide-HiveSerDe>

Hive SerDe

row_format

: DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]]
[COLLECTION ITEMS TERMINATED BY char]
[MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]

Default values: Ctrl+A, Ctrl+B, Ctrl+C, new line, respectively

file_format:

: SEQUENCEFILE
| TEXTFILE -- (Default, depending on `hive.default.fileformat` configuration)
| RCFILE -- (Note: Available in Hive 0.6.0 and later)
| ORC -- (Note: Available in Hive 0.11.0 and later)
| PARQUET -- (Note: Available in Hive 0.13.0 and later)
| AVRO -- (Note: Available in Hive 0.14.0 and later)
| INPUTFORMAT `input_format_classname` OUTPUTFORMAT
`output_format_classname`

Create Table Example

■ Example:

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
    page_url STRING, referrer_url STRING,
    ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY(dt STRING, country STRING)
CLUSTERED BY(userid) SORTED BY(viewTime)
INTO 32 BUCKETS

ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\001'
    COLLECTION ITEMS TERMINATED BY '\002'
    MAP KEYS TERMINATED BY '\003'
    LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Browsing Tables and Partitions

- To list existing tables in the warehouse
 - SHOW TABLES;
- To list tables with prefix 'page'
 - SHOW TABLES 'page.*';
- To list partitions of a table
 - SHOW PARTITIONS page_view;
- To list columns and column types of table.
 - DESCRIBE page_view;

Alter Table/Partition/Column

- To rename existing table to a new name
 - `ALTER TABLE old_table_name RENAME TO new_table_name;`
- To rename the columns of an existing table
 - `ALTER TABLE old_table_name REPLACE COLUMNS (col1 TYPE, ...);`
- To add columns to an existing table
 - `ALTER TABLE tab1 ADD COLUMNS (c1 INT COMMENT 'a new int column', c2 STRING DEFAULT 'def val');`
- To rename a partition
 - `ALTER TABLE table_name PARTITION old_partition_spec RENAME TO PARTITION new_partition_spec;`
- To rename a column
 - `ALTER TABLE table_name CHANGE old_col_name new_col_name column_type`
- More details see:

<https://cwiki.apache.org/confluence/display/Hive>

"How to Manage DDL" "How to Manage DDL Alter Table"

Drop Table/Partition

- To drop a table
 - DROP TABLE [IF EXISTS] table_name
 - Example:
 - ▶ DROP TABLE page_view
- To drop a partition
 - ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec[, PARTITION partition_spec, ...]
 - Example:
 - ▶ ALTER TABLE pv_users DROP PARTITION (ds='2008-08-08')

Loading Data

- Hive does not do any transformation while loading data into tables.
Load operations are currently pure copy/move operations that move datafiles into locations corresponding to Hive tables.
- Syntax:

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO  
TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]
```

- Load data from a file in the local files system
 - ▶ LOAD DATA **LOCAL** INPATH '/tmp/pv_2008-06-08_us.txt'
INTO TABLE page_view PARTITION(date='2008-06-08',
country='US')
- Load data from a file in HDFS
 - ▶ LOAD DATA INPATH '/user/data/pv_2008-06-08_us.txt' INTO
TABLE page_view PARTITION(date='2008-06-08',
country='US')
- The input data format must be the same as the table format!

Insert Data

■ Insert rows into a table:

- Syntax

```
INSERT INTO TABLE tablename [PARTITION (partcol1[=val1],  
partcol2[=val2] ...)] VALUES values_row [, values_row ...]
```

■ Inserting data into Hive Tables from queries

- Syntax

```
INSERT INTO TABLE tablename [PARTITION (partcol1=val1,  
partcol2=val2 ...)] select_statement FROM from_statement;
```

- Example:

```
INSERT OVERWRITE TABLE user_active  
SELECT user.*  
FROM user  
WHERE user.active = 1;
```

Update Data

■ Syntax:

```
UPDATE tablename SET column = value [, column = value ...] [WHERE expression]
```

■ Synopsis

- The referenced column must be a column of the table being updated.
- The value assigned must be an expression that Hive supports in the select clause. Thus arithmetic operators, UDFs, casts, literals, etc. are supported. Subqueries are not supported.
- Only rows that match the WHERE clause will be updated.
- Partitioning columns cannot be updated.
- Bucketing columns cannot be updated.

Query Data

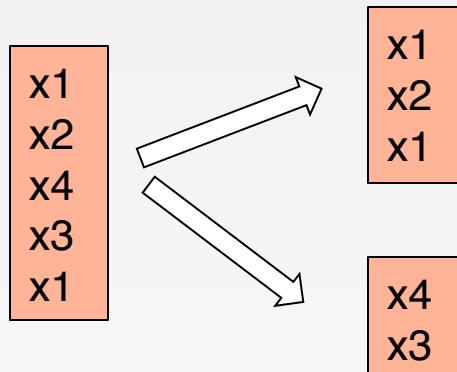
■ Select Syntax:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
      FROM table_reference
      [WHERE where_condition]
      [GROUP BY col_list]
      [ORDER BY col_list]
      [CLUSTER BY col_list
           | [DISTRIBUTE BY col_list] [SORT BY col_list]
      ]
      [LIMIT number]
```

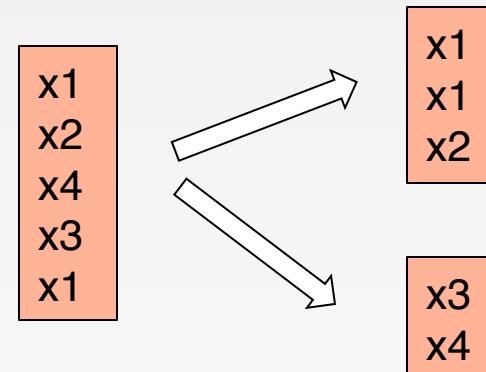
Order, Sort, Cluster, and Distribute By

- Difference between *Order By* and *Sort By*
 - The former guarantees total order in the output while the latter only guarantees ordering of the rows within a reducer

- Cluster By
 - *Cluster By* is a short-cut for both *Distribute By* and *Sort By*.
 - Hive uses the columns in *Distribute By* to distribute the rows among reducers. All rows with the same *Distribute By* columns will go to the same reducer. However, *Distribute By* does not guarantee clustering or sorting properties on the distributed keys.



Distribute By



Cluster By

Query Examples

- Selects column 'foo' from all rows of partition ds=2008-08-15 of the invites table. The results are not stored anywhere, but are displayed on the console.

```
hive> SELECT a.foo FROM invites a WHERE a.ds='2008-08-15';
```

- Selects all rows from partition ds=2008-08-15 of the invites table into an HDFS directory.

```
hive> INSERT OVERWRITE DIRECTORY '/tmp/hdfs_out' SELECT  
a.* FROM invites a WHERE a.ds='2008-08-15';
```

- Selects all rows from pokes table into a local directory.

```
hive> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/local_out'  
SELECT a.* FROM pokes a;
```

Group By

- Count the number of distinct users by gender

```
INSERT OVERWRITE TABLE pv_gender_sum
SELECT pv_users.gender, count(DISTINCT pv_users.userid)
FROM pv_users
GROUP BY pv_users.gender;
```

- Multiple DISTINCT expressions in the same query is not allowed

```
INSERT OVERWRITE TABLE pv_gender_agg
SELECT pv_users.gender, count(DISTINCT pv_users.userid),
count(DISTINCT pv_users.ip)
FROM pv_users
GROUP BY pv_users.gender;
```

Joins

- Hive does not support join conditions that are not equality conditions
 - it is very difficult to express such conditions as a map/reduce job
 - SELECT a.* FROM a JOIN b ON (a.id = b.id)
 - However, the following statement is not allowed:
 - ▶ SELECT a.* FROM a JOIN b ON (a.id <> b.id)
- More than 2 tables can be joined in the same query.
 - SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key2)
- Example:

```
SELECT s.word, s.freq, k.freq
  FROM shakespeare s
  JOIN bible k ON (s.word =
k.word) WHERE s.freq >= 1
  AND k.freq >= 1 ORDER BY
s.freq DESC LIMIT 10;
```

the	25848	62394
I	23031	8854
and	19671	38985
to	18038	13526
of	16700	34654
a	14170	8057
you	12702	2720
my	11297	4135
in	10797	12445
is	8882	6884

Behind the Scenes

```
SELECT s.word, s.freq, k.freq FROM shakespeare s  
JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1  
ORDER BY s.freq DESC LIMIT 10;
```



(Abstract Syntax Tree)

```
(TOK_QUERY (TOK_FROM (TOK_JOIN (TOK_TABREF shakespeare s) (TOK_TABREF bible k) (= (. (TOK_TABLE_OR_COL s)  
word) (. (TOK_TABLE_OR_COL k) word)))) (TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE)) (TOK_SELECT  
(TOK_SELEXPR (. (TOK_TABLE_OR_COL s) word)) (TOK_SELEXPR (. (TOK_TABLE_OR_COL s) freq)) (TOK_SELEXPR (.  
(TOK_TABLE_OR_COL k) freq))) (TOK_WHERE (AND (>= (. (TOK_TABLE_OR_COL s) freq) 1) (>= (. (TOK_TABLE_OR_COL k)  
freq) 1))) (TOK_ORDERBY (TOK_TABSORTCOLNAMEDESC (. (TOK_TABLE_OR_COL s) freq))) (TOK_LIMIT 10)))
```



(one or more of MapReduce jobs)

Hive Operators and User-Defined Functions (UDFs)

- Built-in operators:
 - relational, arithmetic, logical, etc.
- Built-in functions:
 - mathematical, date function, string function, etc.
- Built-in aggregate functions:
 - max, min, count, etc.
- Built-in table-generating functions: transform a single input row to multiple output rows
 - explode(ARRAY): Returns one row for each element from the array.
 - explode(MAP): Returns one row for each key-value pair from the input map with two columns in each row
- Create Custom UDFs
- More details see:

<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF#LanguageManualUDF-explode>

WordCount in Hive

■ Create a table in Hive

```
create table doc(  
    text string  
) row format delimited fields terminated by '\n' stored as textfile;
```

■ Load file into table

```
load data local inpath '/home/Words' overwrite into table doc;
```

■ Compute word count using select

```
SELECT word, COUNT(*) FROM doc LATERAL VIEW  
explode(split(text, ' ')) Table as word GROUP BY word;
```

- explode() takes in an array (or a map) as an input and outputs the elements of the array (map) as separate rows.
- Lateral view is used in conjunction with user-defined table generating functions such as explode()
- A lateral view first applies the UDTF to each row of base table and then joins resulting output rows to form a virtual table

Another Example

■ Load Data

```
LOAD DATA LOCAL INPATH '/logs/status_updates'  
INTO TABLE status_updates PARTITION (ds='2009-03-20')
```

■ Two insertion from select

```
FROM (SELECT a.status, b.school, b.gender  
      FROM status_updates a JOIN profiles b  
        ON (a.userid = b.userid and  
            a.ds='2009-03-20' )  
    ) subq1  
INSERT OVERWRITE TABLE gender_summary  
                  PARTITION(ds='2009-03-20')  
SELECT subq1.gender, COUNT(1) GROUP BY subq1.gender  
INSERT OVERWRITE TABLE school_summary  
                  PARTITION(ds='2009-03-20')  
SELECT subq1.school, COUNT(1) GROUP BY subq1.school
```

Pros/Cons

■ Pros

- A easy way to process large scale data
- Support SQL-based queries
- Provide more user defined interfaces to extend
- Programmability
- Efficient execution plans for performance
- Interoperability with other database

■ Cons

- No easy way to append data
- Files in HDFS are immutable

Applications of Hive

- Log processing
 - Daily Report
 - User Activity Measurement
- Data/Text mining
 - Machine learning (Training Data)
- Business intelligence
 - Advertising Delivery
 - Spam Detection

References

- HDFS design:

<https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>

- <https://hbase.apache.org/book.html>
- <https://cwiki.apache.org/confluence/display/Hive/Home#Home-HiveDocumentation>
- <http://www.tutorialspoint.com/hive/>
- Hadoop The Definitive Guide. HDFS Chapter.
- Hadoop The Definitive Guide. HBase Chapter.
- Hadoop the Definitive Guide. Hive Chapter

End of Chapter 10