

Hadoop

Essentials

Chapter 1

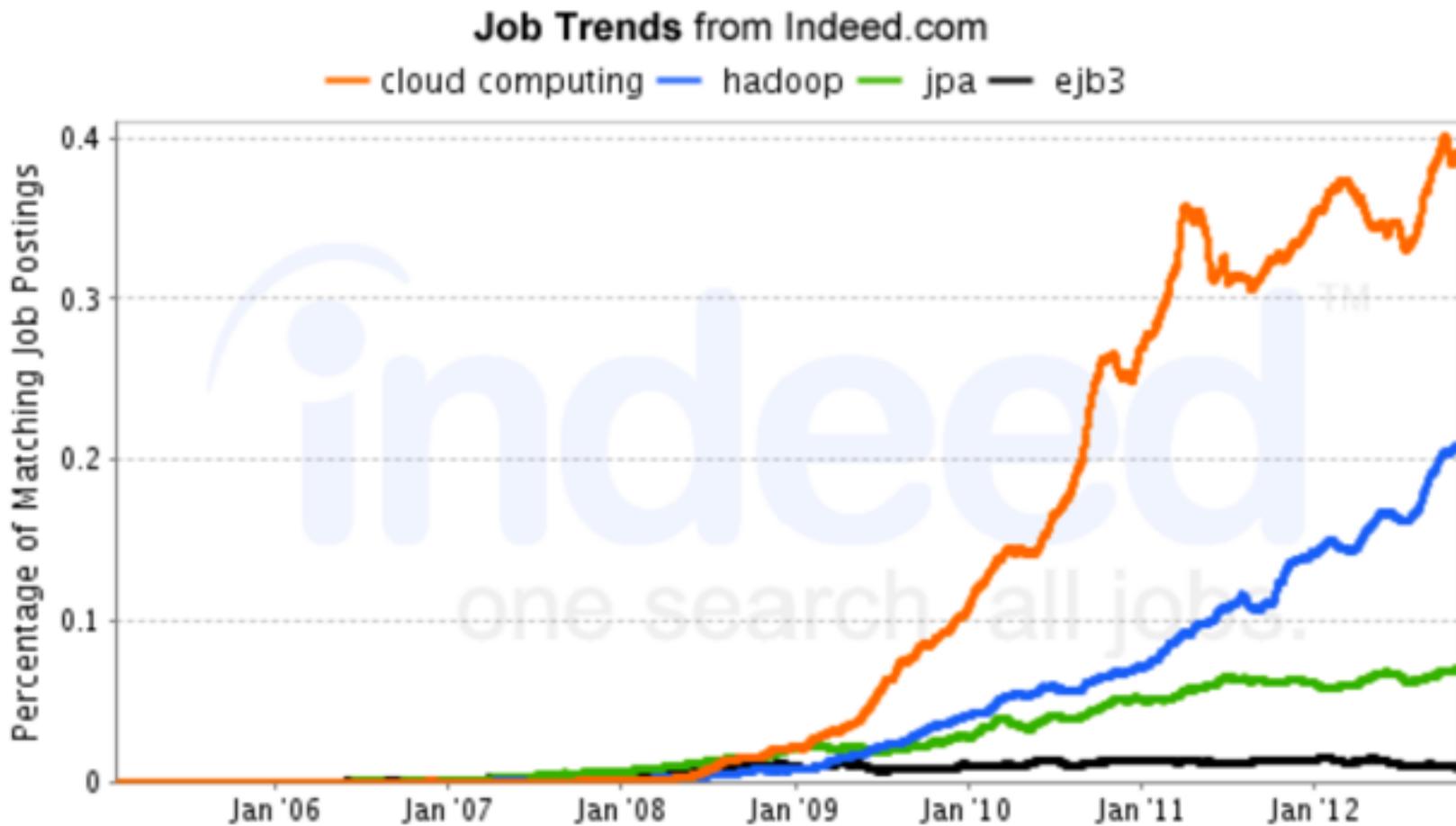
BIG DATA

The Numbers



- Information Data Corporation (IDC) estimates data created in 2010 to be
- 1.2 ZETTABYTES
- (1.2 Trillion Gigabytes)
- Companies continue to generate large amounts of data, here are some 2011 stats:
 - Facebook ~ 6 billion messages per day
 - EBay ~ 2 billion page views a day, ~ 9 Petabytes of storage
 - Satellite Images by Skybox Imaging ~ 1 Terabyte per day

Hadoop Jobs



Who Uses Hadoop?



facebook

hulu

last.fm
the social music revolution

LinkedIn

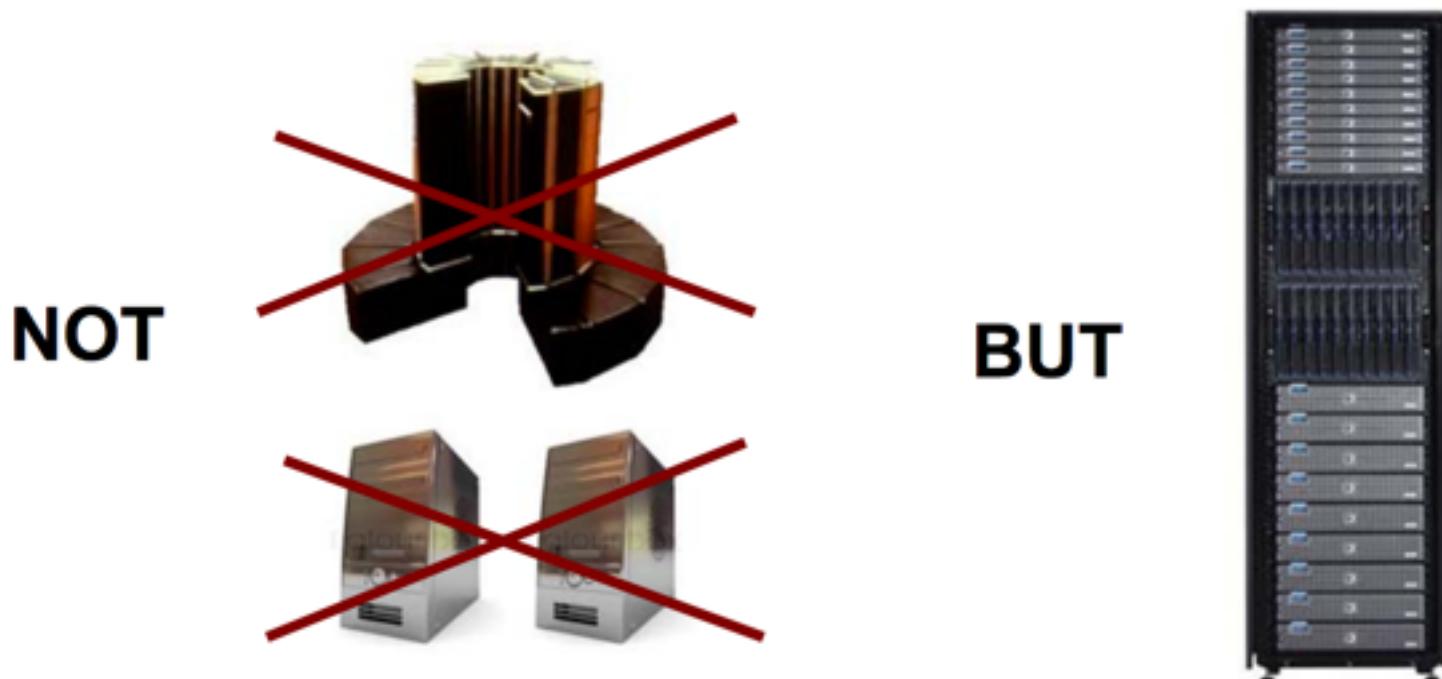
YAHOO!

Data Storage

- Storage capacity has grown exponentially but read speed has not kept up
 - 1990:
 - Store 1,400 MB (4.5MB/s)
 - Read the entire drive in ~ 5 minutes
 - 2010:
 - Store 1 TB (100MB/s)
 - Read the entire drive in ~ 3 hours
- Hadoop - 100 drives working at the same time can read 1TB of data in 2 minutes

Cheap Commodity Hardware

- “Cheap” Commodity Server Hardware
 - No need for super-computers, use commodity unreliable hardware
 - Not desktops

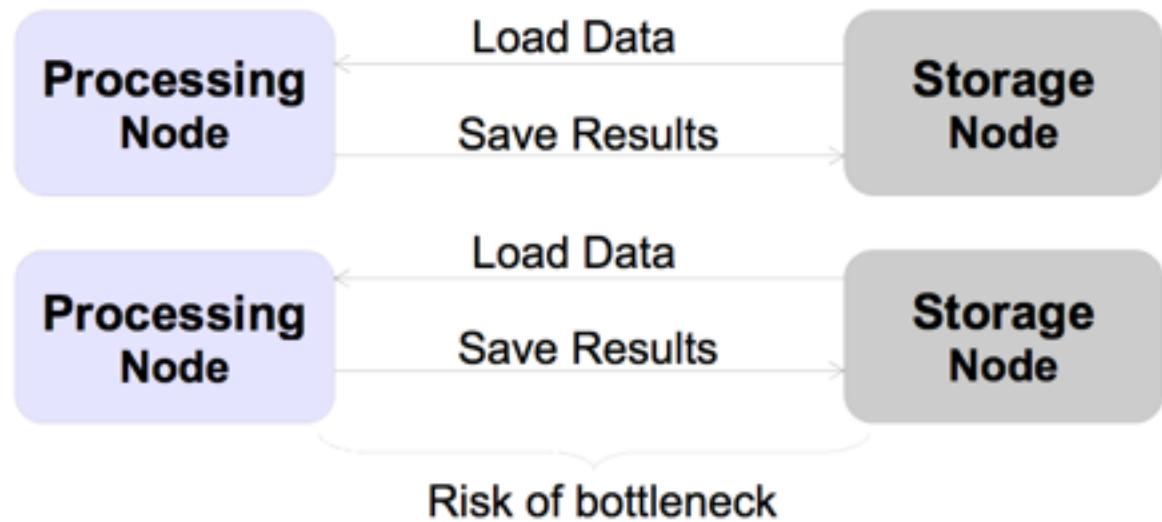


Scale Out not Up

- It is harder and more expensive to scale-up
 - Add additional resources to an existing node (CPU, RAM)
 - Moore's Law can't keep up with data growth
 - Also known as scale vertically

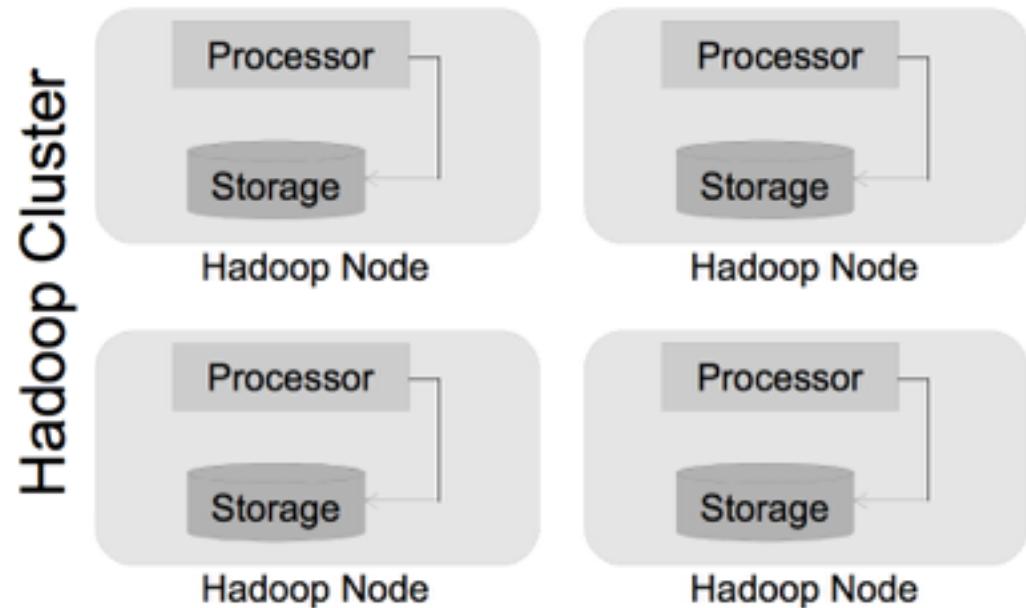
Code to Data

- Traditional data processing architecture
 - nodes are broken up into separate processing and storage nodes connected by high-capacity link
- Many data-intensive applications are not CPU demanding causing bottlenecks in network



Code to Data

- Hadoop co-locates processors and storage
 - Code is moved to data (size is tiny, usually in KBs)
 - Processors execute code and access underlying local storage



Embrace Failure

- Given a large number machines, failures are common
 - Large warehouses may see machine failures weekly or even daily
- Hadoop is designed to cope with node failures
 - Data is replicated
 - Tasks are retried

Abstracting Complexity

- Hadoop abstracts many complexities in distributed and concurrent applications
- Frees developers from worrying about system level changes
- Allows developers to focus on application development and business logic

Hadoop History

- Started as a sub-project of Apache Nutch
 - Nutch's job is to index the web and expose it for searching
 - Open Source alternative to Google
 - Started by Doug Cutting
- In 2004 Google publishes Google File System (GFS) and MapReduce framework papers
- Nutch team implemented Google's frameworks
- 2006 Yahoo! hires Doug Cutting to work on Hadoop with a dedicated team
- 2008 Hadoop became Apache Top Level Project

Compared to RDBMs

- Until recently many applications utilized relational Database Management Systems (RDBMS) for batch processing
 - Hadoop doesn't fully replace relational products
- Structured Relational vs. Semi-Structured vs. Unstructured

Comparison to RDBMs (cont:)

- Offline batch vs. online transactions
 - Hadoop was not designed for real-time or low latency queries
 - Hadoop performs best for offline batch processing on large amounts of data
 - RDBMS is best for online transactions and low-latency queries

Big Data

- Hadoop and RDBMS frequently complement each other within an architecture
- For example, a website that
 - has a small number of users
 - produces a large amount of audit logs



- 1 Utilize RDBMS to provide rich User Interface and enforce data integrity
- 2 RDBMS generates large amounts of audit logs; the logs are moved periodically to the Hadoop cluster
- 3 All logs are kept in Hadoop; Various analytics are executed periodically
- 4 Results copied to RDBMS to be used by Web Server; for example "suggestions" based on audit history

The Hadoop Ecosystem

- At first Hadoop was mainly known for two core products:
 - HDFS: Hadoop Distributed FileSystem
 - MapReduce: Distributed data processing framework

Hadoop Ecosystem

- Today, in addition to HDFS and MapReduce, the term also represents a multitude of products:
 - HBase: Hadoop column database; supports batch and random reads and limited queries
 - Zookeeper: Highly-Available Coordination Service
 - Pig: Data processing language and execution environment
 - Hive: Data warehouse with SQL interface
 - Many others...

Hadoop Building Blocks

- To start building an application, you need a file system
 - In the Hadoop world that would be Hadoop Distributed File System (HDFS)
 - In Linux it could be ext3 or ext4
- Addition of a data store would provide a nicer interface to store and manage your data
 - HBase: A key-value store implemented on top of HDFS
 - Traditionally one could use RDBMS on top of a local file system

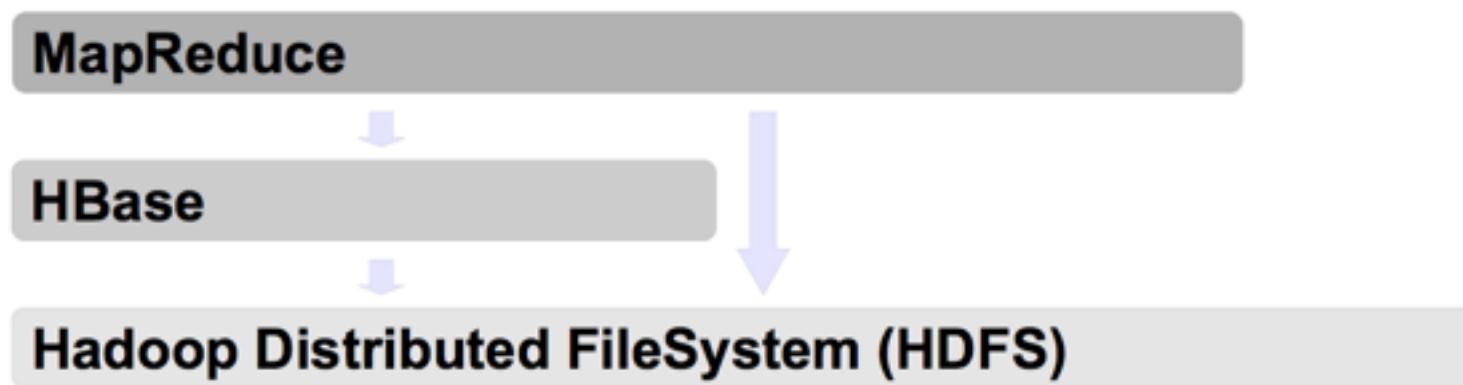
HBase



Hadoop Distributed FileSystem (HDFS)

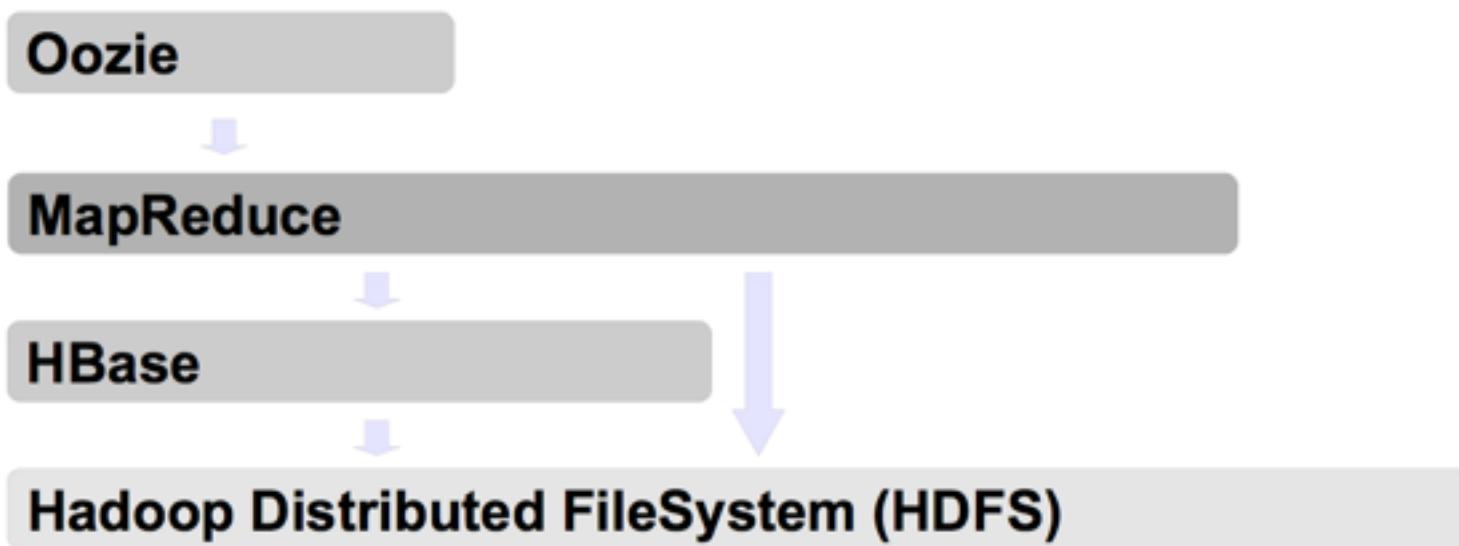
Hadoop Building Blocks

- For batch processing, you will need to utilize a framework
 - In Hadoop's world that would be MapReduce
 - MapReduce will ease implementation of distributed applications that will run on a cluster of commodity hardware



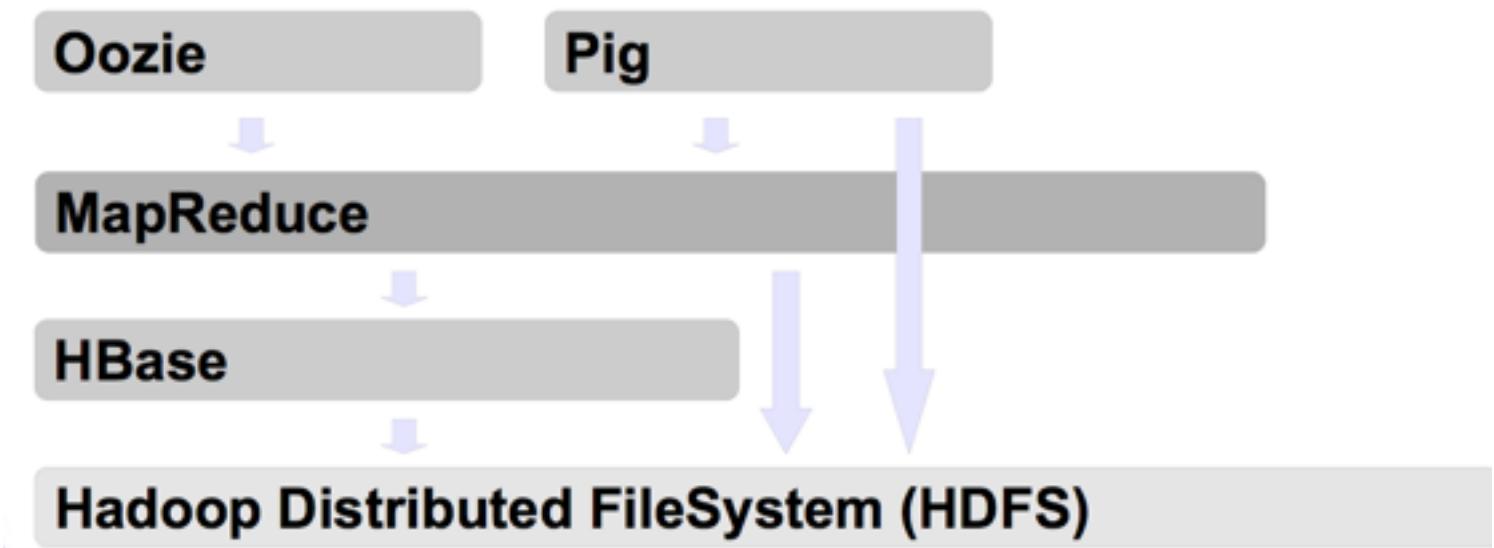
Hadoop Building Blocks

- Many problems lend themselves to a MapReduce solution with multiple jobs
 - Apache Oozie is a popular MapReduce workflow and coordination product



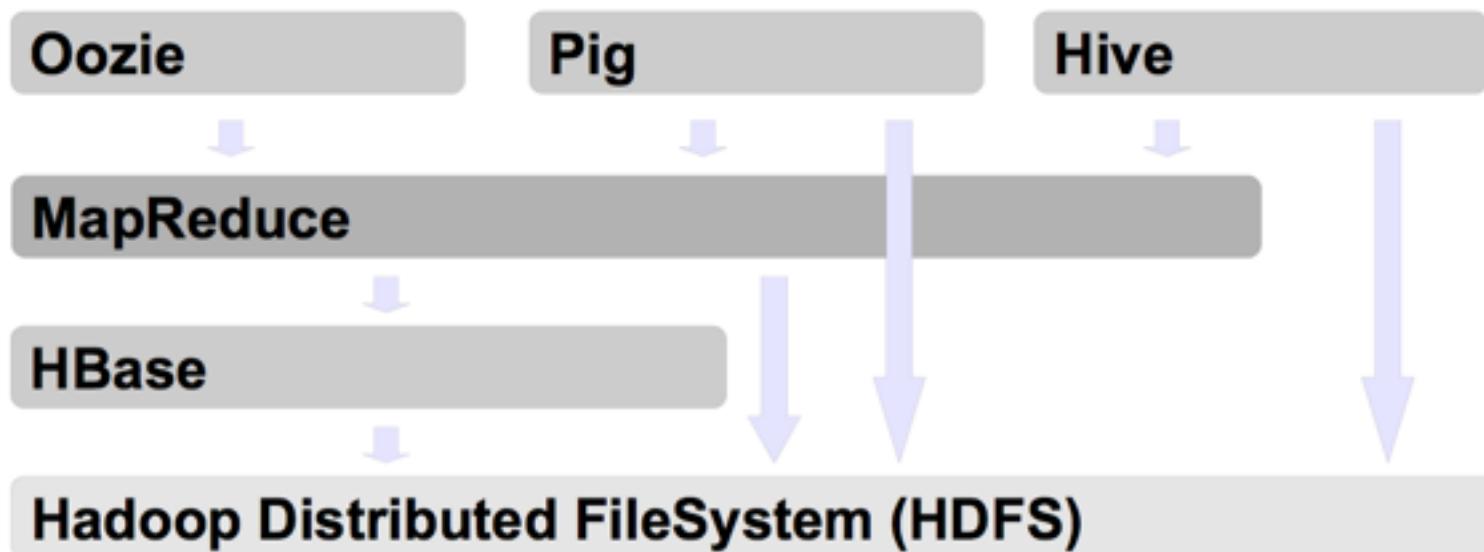
Hadoop Building Blocks

- MapReduce paradigm may not work well for analysts and data scientists
 - Addition of Apache Pig, a high-level data flow scripting language, may be beneficial



Hadoop Building Blocks

- Your organization may have a good number of SQL experts
 - Addition of Apache Hive, a data warehouse solution that provides a SQL based interface, may bridge the gap



The Distribution Vendor

- Hadoop Distributions aim to resolve version incompatibilities
- Distribution Vendor will
 - Integration Test a set of Hadoop products
 - Package Hadoop products in various installation formats

Distribution Vendors

- Cloudera Distribution for Hadoop (CDH)
- MapR Distribution
- Hortonworks Data Platform (HDP)
- Apache BigTop Distribution
- Greenplum HD Data Computing
- Appliance



GREENPLUM[®]
A DIVISION OF EMC

Supported Operating Systems

- Each Distribution will support its own list of Operating Systems (OS)
- Common OS supported
 - Red Hat Enterprise
 - CentOS
 - Oracle Linux
 - Ubuntu
 - SUSE Linux Enterprise Server

Chapter 2

HDFS

HDFS

- Appears as a single disk
- Runs on top of a native filesystem
 - Ext3,Ext4,XFS
- Fault Tolerant
 - Can handle disk crashes, machine crashes, etc...
- Based on Google's Filesystem (GFS or GoogleFS)
 - http://en.wikipedia.org/wiki/Google_File_System

What is HDFS good for?

- Storing large files
 - Terabytes, Petabytes, etc...
 - Millions rather than billions of files
 - 100MB or more per file
- Streaming data
 - Write once and read-many times patterns
 - Optimized for streaming reads rather than random reads
 - Append operation added to Hadoop 0.21

What is HDFS not good for?

- Low-latency reads
 - High-throughput rather than low latency for small chunks of data
 - HBase addresses this issue
- Large amount of small files
 - Better for millions of large files instead of billions of small files (e.g 100MB or more)
- Multiple Writers
 - Single writer per file
 - Writes at the EOF, no-support for arbitrary offset

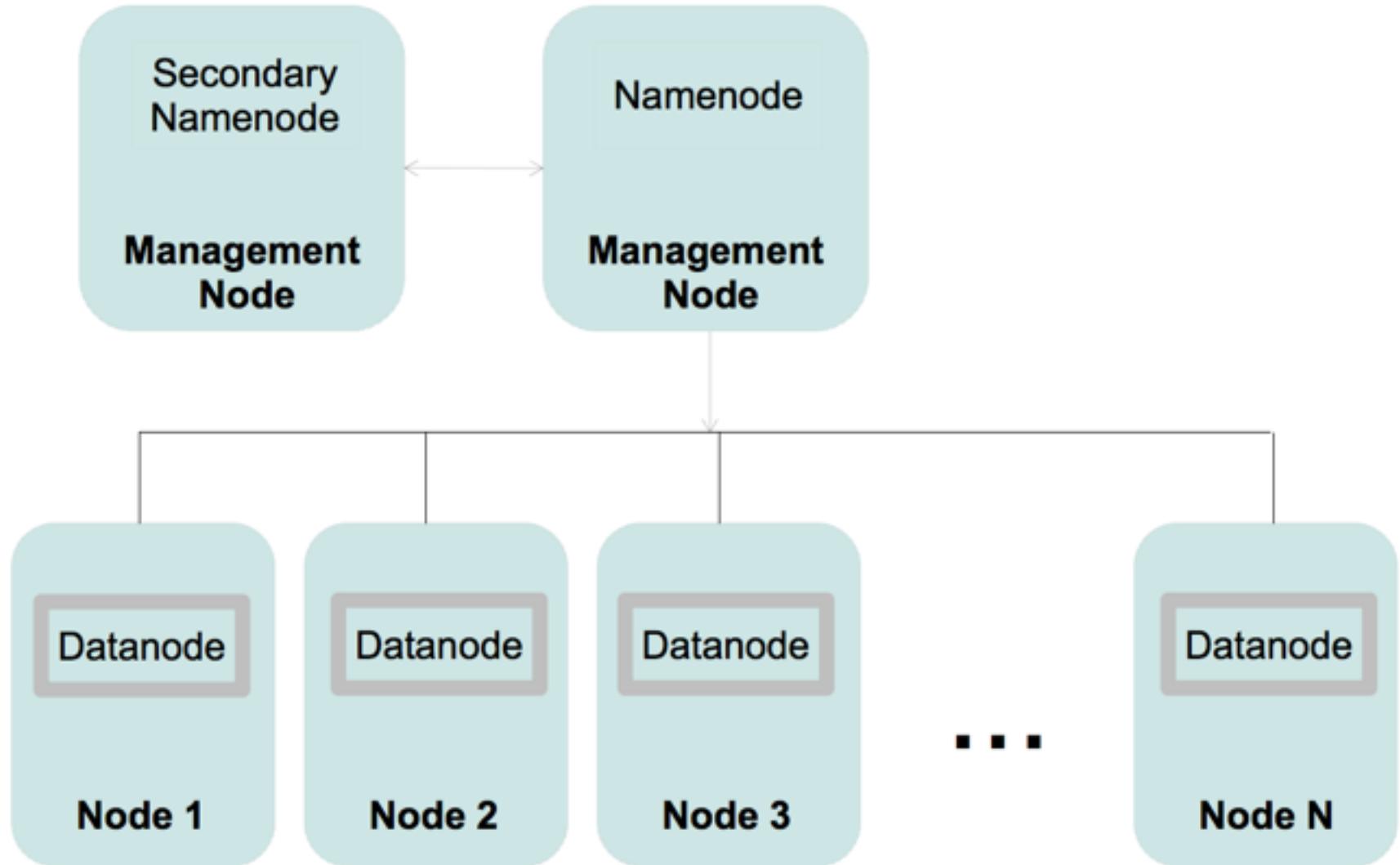
HDFS Daemons

- Filesystem cluster is managed by three types of processes
 - Namenode
 - Manages the File System's namespace/meta-data/file blocks
 - Runs on 1 machine to several machines
 - Can have a hot stand by for failover

HDFS Daemons (cont:)

- Datanode
 - Stores and retrieves data blocks
 - Reports to Namenode
 - Runs on many machines
- Secondary Namenode
 - Performs house keeping work
 - Requires similar hardware as Namenode machine
 - Not used for high-availability - not a backup for Namenode

HDFS Daemons

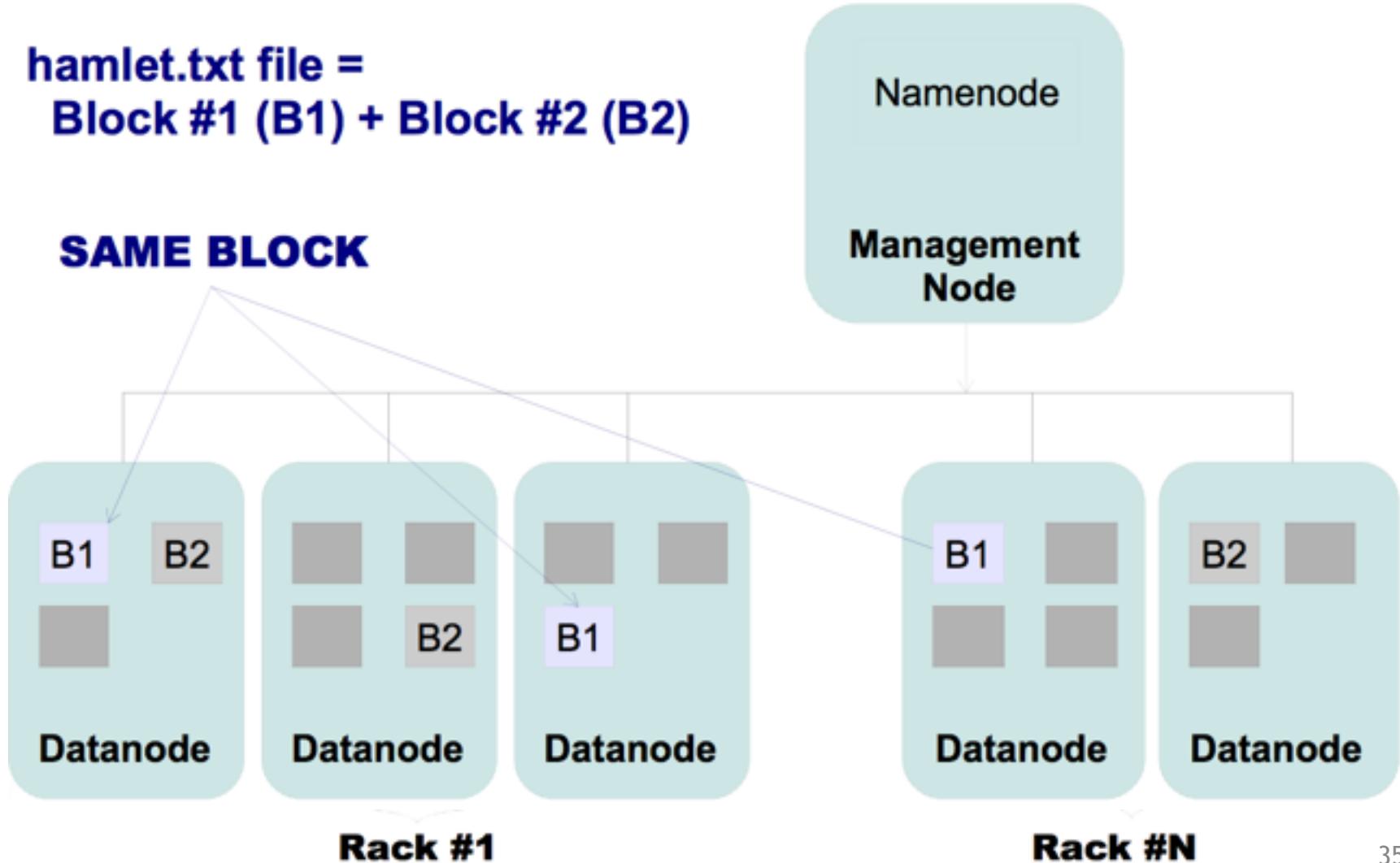


Files and Blocks

- Files are split into blocks (single unit of storage)
 - Managed by Namenode, stored by Datanode
 - Transparent to user
- Replicated across machines at load time
 - Same block is stored on multiple machines
 - Good for fault-tolerance and access
 - Default replication is 3

Files and Blocks

**hamlet.txt file =
Block #1 (B1) + Block #2 (B2)**



HDFS Blocks

- Blocks are traditionally either 64MB or 128MB
 - Default is 64MB
- The motivation is to minimize the cost of seeks as compared to transfer rate
 - 'Time to transfer' > 'Time to seek'
- For example, lets say
 - seek time = 10ms
 - Transfer rate = 100 MB/s
- To achieve seek time of 1% transfer rate
 - Block size will need to be = 100MB

Block Replication

- Namenode determines replica placement
- Replica placements can be rack aware
 - Balance between reliability and performance
- Attempts to reduce bandwidth
- Attempts to improve reliability by putting replicas on multiple racks

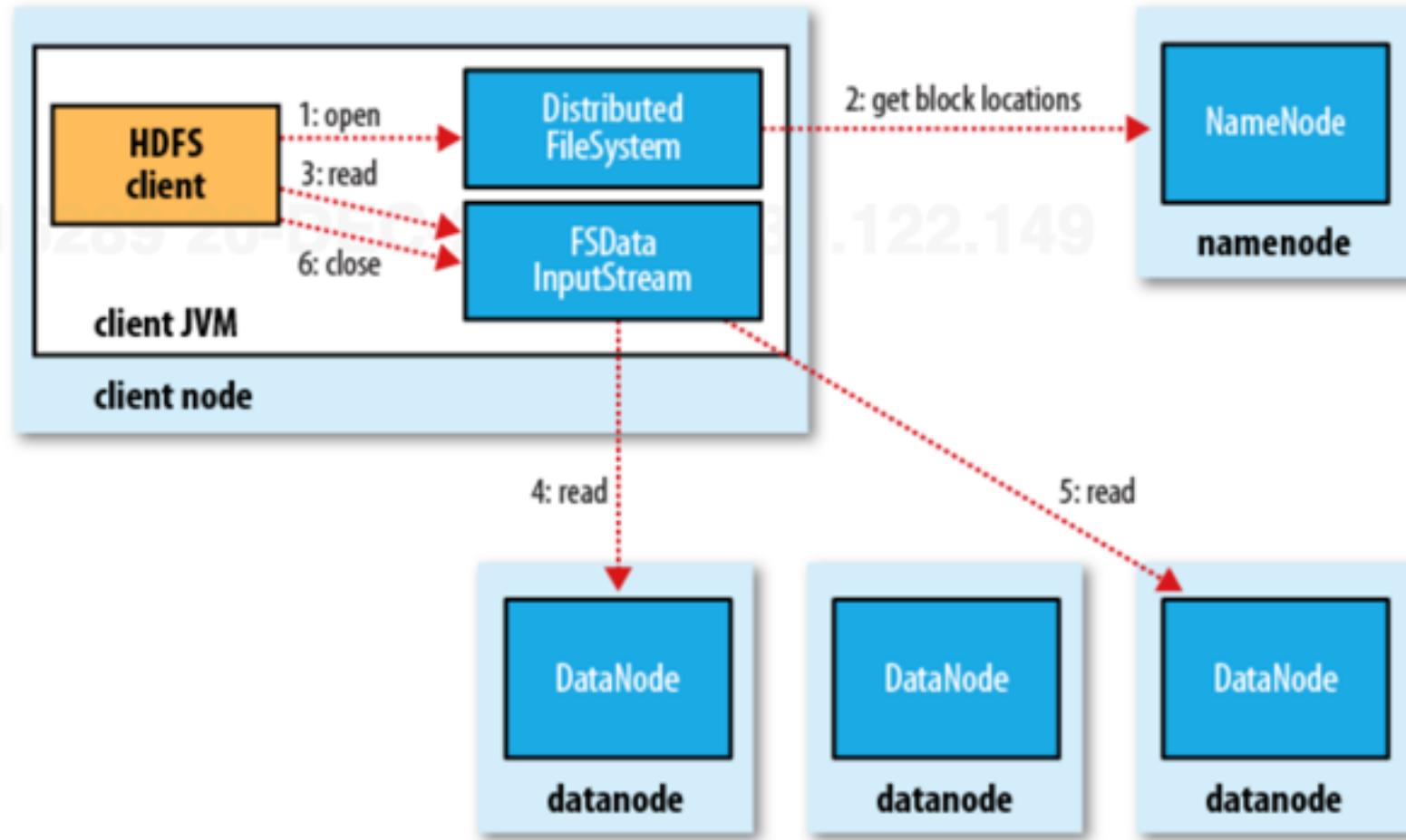
Block Replication (cont:)

- Default replication is 3
- 1st replica on the local rack
- 2nd replica on the local rack but different machine
- 3rd replica on the different rack
 - This policy may change/improve in the future

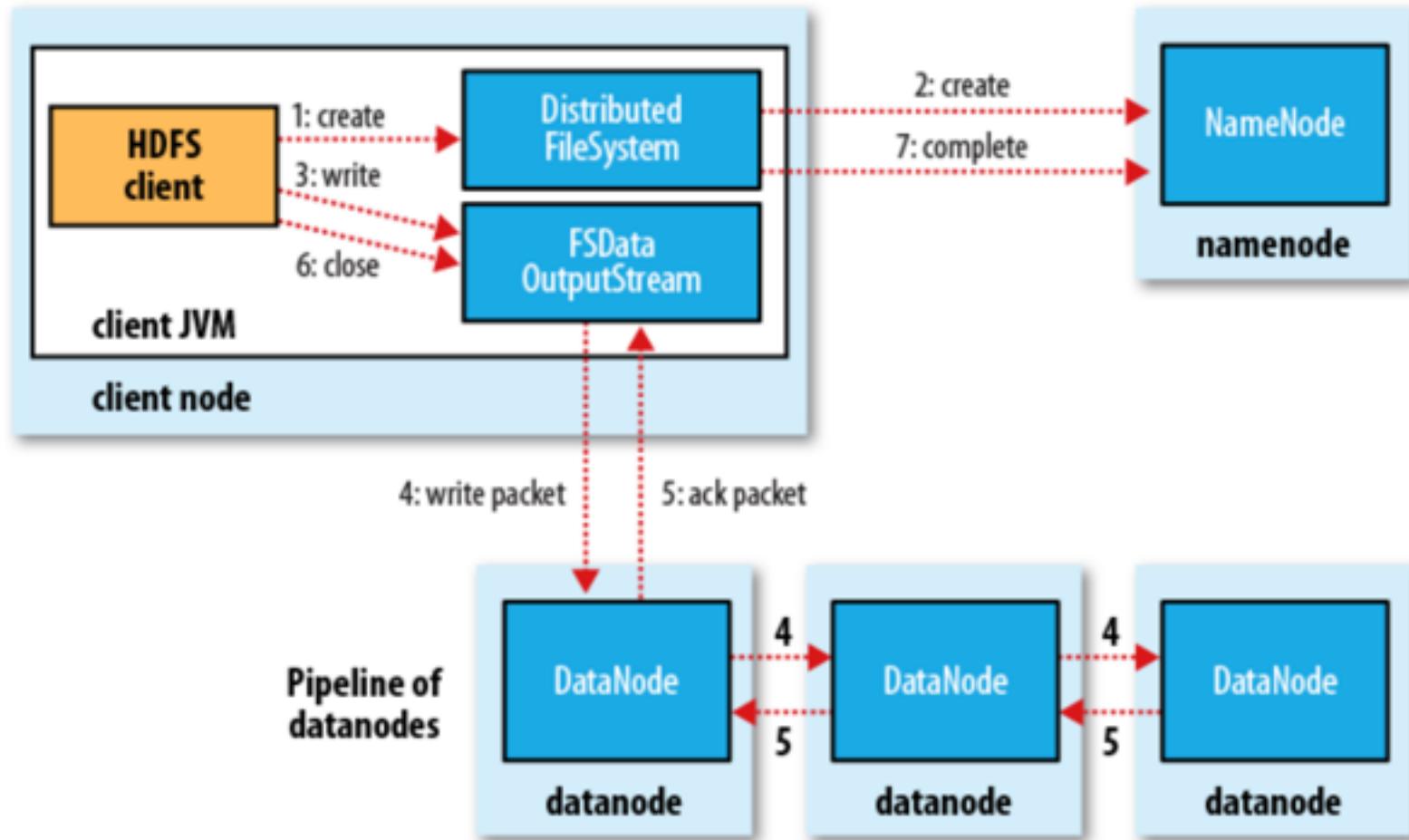
Namenode and Datanodes

- **Namenode does NOT directly write or read data**
 - One of the reasons for HDFS's Scalability
- **Client interacts with Namenode to update**
- **Namenode's HDFS namespace retrieves block locations for writing and reading**
- **Client interacts directly with Datanode to read/write data**

HDFS File Read



HDFS File Write



Namenode Memory

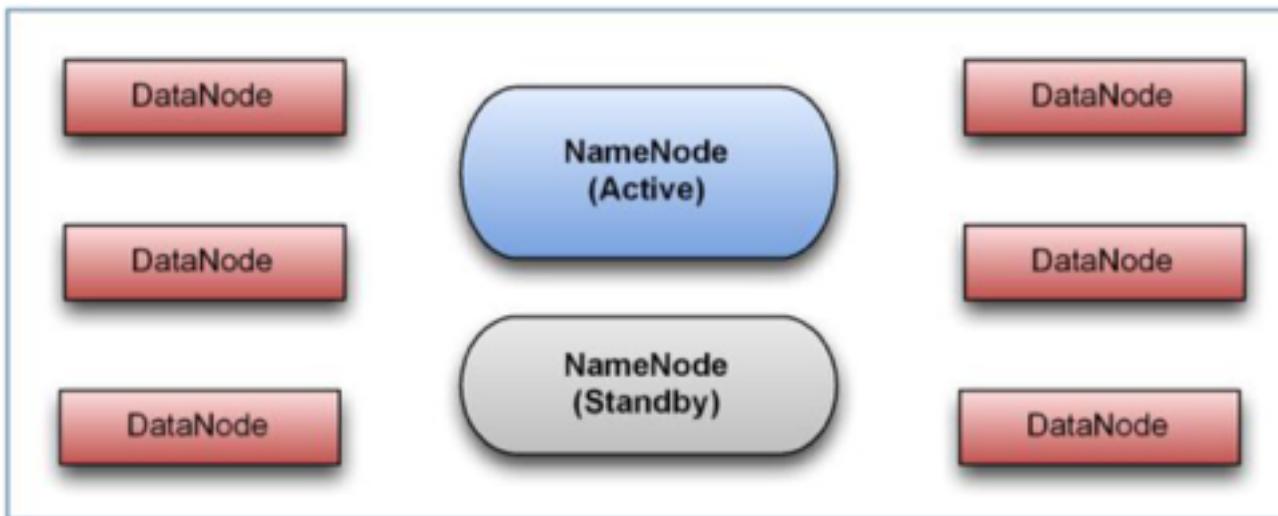
- For fast access Namenode keeps all block metadata in-memory
 - The bigger the cluster - the more RAM required
- Best for millions of large files (100mb or more) rather than billions
- Will work well for clusters of 100s machines

Contemporary HDFS Architecture

- The preceding slides described the "classic" architecture
- HDFS now has High Availability and Federation features
 - Initially developed on the Apache Hadoop 0.23 branch
 - Now part of Hadoop 2.x

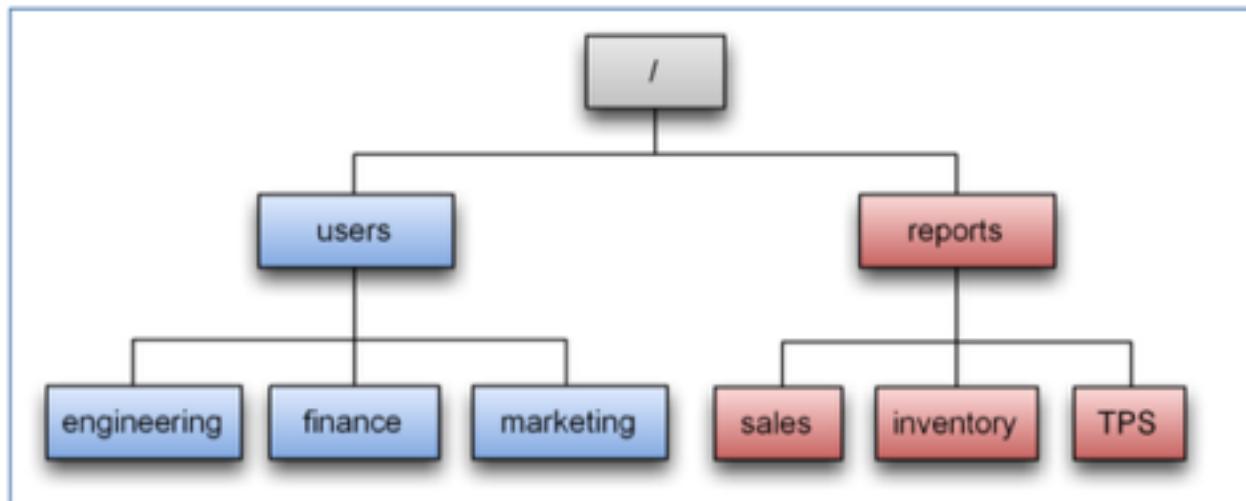
HDFS High Availability

- HDFS High Availability addresses the NameNode SPOF
- Two NameNodes: one active and one standby
 - Standby NameNode takes over when active NameNode fails
 - Standby NameNode also does checkpointing (Secondary NameNode no longer needed)



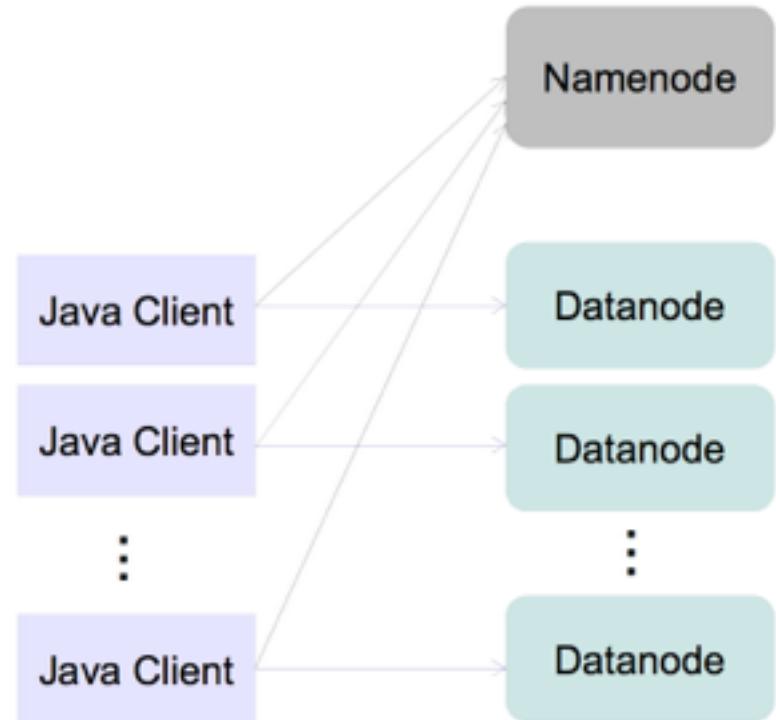
HDFS Federation

- Federation improves the scalability of HDFS
 - Federation allows for multiple independent NameNodes
- Each NameNode manages a namespace volume
 - Client-side mount tables define the overall view
 - NN1 might provide /users and NN2 might provide /reports



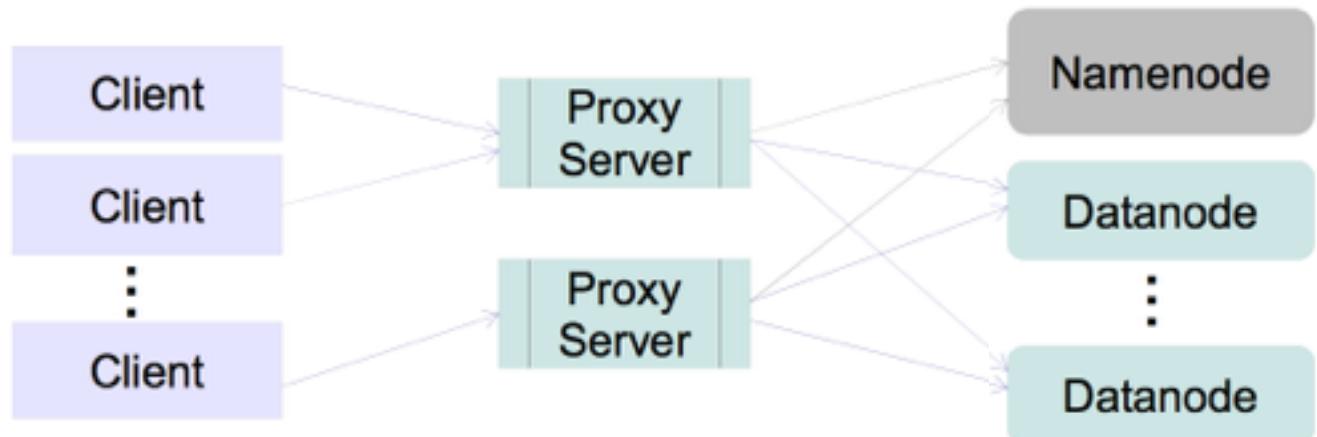
Direct Access

- Java and C++ APIs
- Clients retrieve metadata such as blocks' locations from Namenode
- Client directly access datanode(s)
- Java API
 - Most commonly used
 - Used by MapReduce



Proxy Based Access

- Clients communicate through a proxy
 - Strives to be language independent
- Several Proxy Servers are packaged with Hadoop:
 - Thrift - interface definition language
 - WebHDFS REST - response formatted in JSON, XML or Protocol Buffers
 - Avro - Data Serialization mechanism



Chapter 3

HDFS INSTALLATION AND SHELL

Installation

- **Three options**
 - Local (Standalone) Mode
 - Pseudo-Distributed Mode
 - Fully-Distributed Mode

Local

- Default configuration after the download
- Executes as a single Java process
- Works directly with local filesystem
- Useful for debugging

Pseudo-Distributed

- Still runs on a single node
- Each daemon runs in its own Java process
 - Namenode
 - Secondary Namenode
 - Datanode
- Location for configuration files is specified via HADOOP_CONF_DIR environment property
- Configuration files
 - core-site.xml
 - hdfs-site.xml
 - hadoop-env.sh

Pseudo-Distributed

- **hadoop-env.sh**
 - Specify environment variables
 - Java and Log locations
- Utilized by scripts that execute and manage hadoop

Pseudo-Distributed

- **\$HADOOP_CONF_DIR/core-site.xml**
 - Configurations for core of Hadoop, for example IO properties
- Specify location of Namenode

```
<property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:8020</value>
    <description>NameNode URL</description>
</property>
```

Pseudo-Distributed

- **\$HADOOP_CONF_DIR/hdfs-site.xml**
 - Configurations for Namenode, Datanode and Secondary Namenode daemons
- **\$HADOOP_CONF_DIR/hdfs-site.xml**

Pseudo-Distributed

- **\$HADOOP_CONF_DIR/slaves**
 - Specifies which machines Datanodes will run on
 - One node per line
- **\$HADOOP_CONF_DIR/masters**
 - Specifies which machines Secondary Namenode will run on

Pseudo-Distributed

- Prepare filesystem for use by formatting
 - \$ hdfs namenode -format
- Start the distributed filesystem
 - \$ cd <hadoop_install>/sbin
 - \$ start-dfs.sh
- start-dfs.sh prints the location of the logs

```
$ ./start-dfs.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/hadoop/Training/logs/hdfs/hadoop-
hadoop-namenode-hadoop-laptop.out
localhost: 2012-07-17 22:17:17,054 INFO namenode.NameNode
(StringUtils.java:startupShutdownMessage(594)) - STARTUP_MSG:
localhost: ****
localhost: STARTUP_MSG: Starting NameNode
```

...

Logs

- Each Hadoop daemon writes a log file:
 - Namenode, Datanode, Secondary Namenode
 - Location of these logs are set in \$HADOOP_CONF_DIR/`hadoop-env.sh`
- Log naming convention:
 - `hadoop-hadoop-namenode-hadoop-laptop.out`
 - `product-username-daemon-hostname`

Management Web Interface

- Namenode comes with web based management
 - <http://localhost:50070>
- Features
 - Cluster status
 - View Namenode and Datanode logs
 - Browse HDFS
- Can be configured for SSL ([https:](https://)) based access
- Secondary Namenode also has web UI
 - <http://localhost:50090>

Namenode's Safemode

- HDFS cluster read-only mode
- Modifications to filesystem and blocks are not allowed
- Happens on start-up
 - Loads file system state from fsimage and edits-log files
 - Waits for Datanodes to come up to avoid over-replication
- Could be placed in safemode explicitly

Secondary Namenode

- Namenode stores its state on local/native file-system mainly in two files: edits and fsimage
 - Stored in a directory configured via `dfs.name.dir` property in `hdfs-site.xml`
 - edits : log file where all filesystem modifications are appended
 - fsimage: on start-up namenode reads hdfs state, then merges edits file into fsimage and starts normal operations with empty edits file
- Namenode start-up merges will become slower over time but ...
 - Secondary Namenode to the rescue

Secondary Namenode

- Secondary Namenode is a separate process
 - Responsible for merging edits and fsimage file to limit the size of edits file
 - Usually runs on a different machine than Namenode
 - Memory requirements are very similar to Namenode's

Secondary Namenode

- Checkpoint is kicked off by two properties in hdfs-site.xml
 - `fs.checkpoint.period`: maximum time period between two checkpoints
 - Default is 1 hour
 - Specified in seconds (3600)
 - `fs.checkpoint.size`: when the size of the edits file exceeds this threshold a checkpoint is kicked off
 - Default is 64 MB
 - Specified in bytes (67108864)

Secondary Namenode

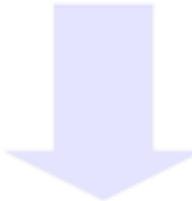
- Secondary Namenode uses the same directory structure as Namenode
 - This checkpoint may be imported if Namenode's image is lost
- Remember, with classic config, Secondary Namenode is NOT
 - Fail-over for Namenode
 - Doesn't provide high availability
 - Doesn't improve Namenode's performance
- As discussed before, name node federation and failover are now available

Shell Commands

- Interact with the FileSystem by executing shell-like commands
- Usage: \$hdfs dfs -<command> -<option> <URI>
 - Example \$hdfs dfs -ls /
- URI usage:
 - HDFS: \$hdfs dfs -ls hdfs://localhost/to/path/dir
 - Local: \$hdfs dfs -ls file://path1/path/file3
 - Schema and namenode host is optional, default is used from the configuration
 - In core-site.xml - fs.default.name property

Hadoop URI

- **scheme://authority/path**



- **hdfs://localhost:8020/user/home**

Scheme and authority determine which file system implementation to use. In this case it will be HDFS

Path on the file system

Shell Commands

- Most commands behave like UNIX commands
 - ls, cat, du, etc..
- Supports HDFS specific operations
 - Ex: changing replication
- List supported commands
 - \$ hdfs dfs -help
- Display detailed help for a command
 - \$ hdfs dfs -help <command_name>

Shell Commands

- Relative Path
 - Is always relative to user's home directory
 - Home directory is at /user/<username>
- Shell commands follow the same format:

\$ hdfs dfs -<command> -<option> <path>

- For example:
 - \$ hdfs dfs -rm -r /removeMe

Shell Basic Commands

- **cat** - stream source to stdout
 - entire file: `$hdfs dfs -cat /dir/file.txt`
 - Almost always a good idea to pipe to head, tail, more or less
 - Get the first 25 lines of file.txt
 - `$hdfs dfs -cat /dir/file.txt | head -n 25`

Shell Basic Commands

- **cp** - copy files from source to destination
 - `$hdfs dfs -cp /dir/file1 /otherDir/file2`
- **ls** - for a file displays stats, for a directory displays immediate children
 - `$hdfs dfs -ls /dir/`
- **mkdir** - create a directory
 - `$hdfs dfs -mkdir /brandNewDir`

Moving Data with Shell

- **mv** - move from source to destination
 - `$hdfs dfs -mv /dir/file1 /dir2/file2`
- **put** - copy file from local filesystem to hdfs
 - `$hdfs dfs -put localfile /dir/file1`
 - Can also use `copyFromLocal`
- **get** - copy file to the local filesystem
 - `$hdfs dfs -get /dir/file localfile`
 - Can also use `copyToLocal`

Deleting Data with Shell

- **rm - delete files**
 - `$hdfs dfs -rm /dir/fileToDelete`
- **rm -r - delete directories recursively**
 - `$hdfs dfs -rm -r /dirWithStuff`

Filesystem Stats with Shell

- du - displays length for each file/dir (in bytes)
 - \$hdfs dfs -du /someDir/
- Add -h option to display in human-readable format instead of bytes
 - \$hdfs dfs -du -h /someDir
 - 206.3k /someDir

Learn More About Shell

- More commands
 - tail, chmod, count, touchz, test, etc...
- To learn more
 - \$hdfs dfs -help
 - \$hdfs dfs -help <command>
- For example:
 - \$ hdfs dfs -help rm

fsck Command

- Check for inconsistencies
- Reports problems
 - Missing blocks
 - Under-replicated blocks
- Doesn't correct problems, just reports (unlike native fsck)
 - Namenode attempts to automatically correct issues that fsck would report
- \$ hdfs fsck <path>
 - Example : \$ hdfs fsck /

HDFS Permissions

- Limited to File permission
 - Similar to POSIX model, each file/directory
 - has Read (r), Write (w) and Execute (x)
 - associated with owner, group or all others
- Client's identity determined on host OS
 - Username = `whoami`
 - Group = `bash -c groups`

DFSAdmin Command

- HDFS administrative operations
 - `$hdfs dfsadmin <command>`
 - Example: `$hdfs dfsadmin -report`
- **-report** : displays statistic about HDFS
 - Some of these stats are available on Web Interface
- **-safemode** : enter or leave safemode
 - Maintenance, backups, upgrades, etc..

Rebalancer

- Data on HDFS Clusters may not be uniformly spread between available Datanodes.
 - Ex: New nodes will have significantly less data for some time
 - The location for new incoming blocks will be chosen based on status of Datanode topology, but the cluster doesn't automatically rebalance
- Rebalancer is an administrative tool that analyzes block placement on the HDFS cluster and re-balances
 - \$ hdfs balancer

Chapter 4

HDFS - JAVA API

File System Java API

- `org.apache.hadoop.fs.FileSystem`
 - Abstract class that serves as a generic file system representation
 - Note it's a class and not an Interface
- Implemented in several flavors
 - Ex. Distributed or Local

FileSystem Implementations

- Hadoop ships with multiple concrete implementations:
 - `org.apache.hadoop.fs.LocalFileSystem`
 - Good old native file system using local disk(s)
 - `org.apache.hadoop.hdfs.DistributedFileSystem`
 - Hadoop Distributed File System (HDFS)
 - Will mostly focus on this implementation
 - `org.apache.hadoop.hdfs.HftpFileSystem`
 - Access HDFS in read-only mode over HTTP
 - `org.apache.hadoop.fs.ftp.FTPFileSystem`
 - File system on FTP server

FileSystem Implementations (cont:)

- **FileSystem concrete implementations**
 - Two options that are backed by Amazon S3 cloud
 - `org.apache.hadoop.fs.s3.S3FileSystem`
 - <http://wiki.apache.org/hadoop/AmazonS3>
 - `org.apache.hadoop.fs.kfs.KosmosFileSystem`
 - Backed by CloudStore
 - <http://code.google.com/p/kosmosfs>

FileSystem Implementations

- Different use cases for different concrete implementations
- HDFS is the most common choice
 - `org.apache.hadoop.hdfs.DistributedFileSystem`
 - See `SimpleLS.java` in `HadoopSamples`

FileSystem API: Path

- Hadoop's Path object represents a file or a directory
 - Not `java.io.File` which tightly couples to local filesystem
- Path is really a URI on the FileSystem
 - HDFS: `hdfs://localhost/user/file1`
 - Local: `file:///user/file1`
- Examples:
 - `new Path("/test/file1.txt");`
 - `new Path("hdfs://localhost:9000/test/file1.txt");`

Hadoop's Configuration Object

- Configuration object stores clients' and servers' configuration
 - Very heavily used in Hadoop
 - HDFS, MapReduce, HBase, etc...
- Simple key-value paradigm
 - Wrapper for `java.util.Properties` class which itself is just a wrapper for `java.util.Hashtable`
- Several construction options
- See `LoadConfigurations.java` in `HadoopSamples`

FileSystem API

- Recall `FileSystem` is a generic abstract class used to interface with a file system
- `FileSystem` class also serves as a factory for concrete implementations, with the following methods
- What Happens when `SimpleLS` is Run?

Reading Data from HDFS

- Create FileSystem
 - Open InputStream to a Path
 - Copy bytes using IOUtils
 - Close Stream
-
- See `ReadFile.java` in `HadoopSamples`

Reading Data - Seek

- `FileSystem.open` returns `FSDataInputStream`
 - Extension of `java.io.DataInputStream`
 - Supports random access and reading via interfaces:
 - `PositionedReadable` : read chunks of the stream
 - `Seekable` : seek to a particular position in the stream
- See `SeekReadFile.java` in `HadoopSamples`

Write Data

- Create FileSystem instance
- Open OutputStream
 - FSDataOutputStream in this case
 - Open a stream directly to a Path from FileSystem
 - Creates all needed directories on the provided path
- Copy data using IOUtils
- See WriteToFile.java in HadoopSamples

FileSystem: Writing Data

- Append to the end of the existing file
 - `fs.append(path)`
 - Optional support by concrete FileSystem
 - HDFS supports
- No support for writing in the middle of the file

Overwrite Flag

- Recall FileSystem's `create(Path)` creates all the directories on the provided path
 - `create(new Path("/doesnt_exist/doesnt_exist/file.txt"))`
- See `BadWriteToFile.java` in `HadoopSamples`

Copy/Move from and to Local FileSystem

- Higher level abstractions that allow you to copy and move from and to HDFS
 - `copyFromLocalFile`
 - `moveFromLocalFile`
 - `copyToLocalFile`
 - `moveToLocalFile`
- See `CopyToHdfs.java` in `HadoopSamples`

Delete/Move/Browse

- See `DeleteFile.java` in `HadoopSamples`
- See `Mkdir.java` in `HadoopSamples`
- See `SimpleLS.java` and `LSWithPathFilter.java` in `HadoopSamples`

FileSystem: Globbing

- FileSystem supports file name pattern matching via globStatus() methods
- Good for traversing through a sub-set of files by using a pattern
- Support is similar to bash glob: *, ?, etc...
- See SimpleGlobbing.java in HadoopSamples

FileSystem: Globbing

Glob	Explanation
?	Matches any single character
*	Matches zero or more characters
[abc]	Matches a single character from character set {a,b,c}.
[a-b]	Matches a single character from the character range {a...b}. Note that character a must be lexicographically less than or equal to character b.
[^a]	Matches a single character that is not from character set or range {a}. Note that the ^ character must occur immediately to the right of the opening bracket.
\c	Removes (escapes) any special meaning of character c.
{ab,cd}	Matches a string from the string set {ab, cd}
{ab,c{de,fh}}	Matches a string from the string set {ab, cde, cfh}

Chapter 5

INTEGRATING HADOOP INTO THE ENTERPRISE WORKFLOW

RDBMS Strengths

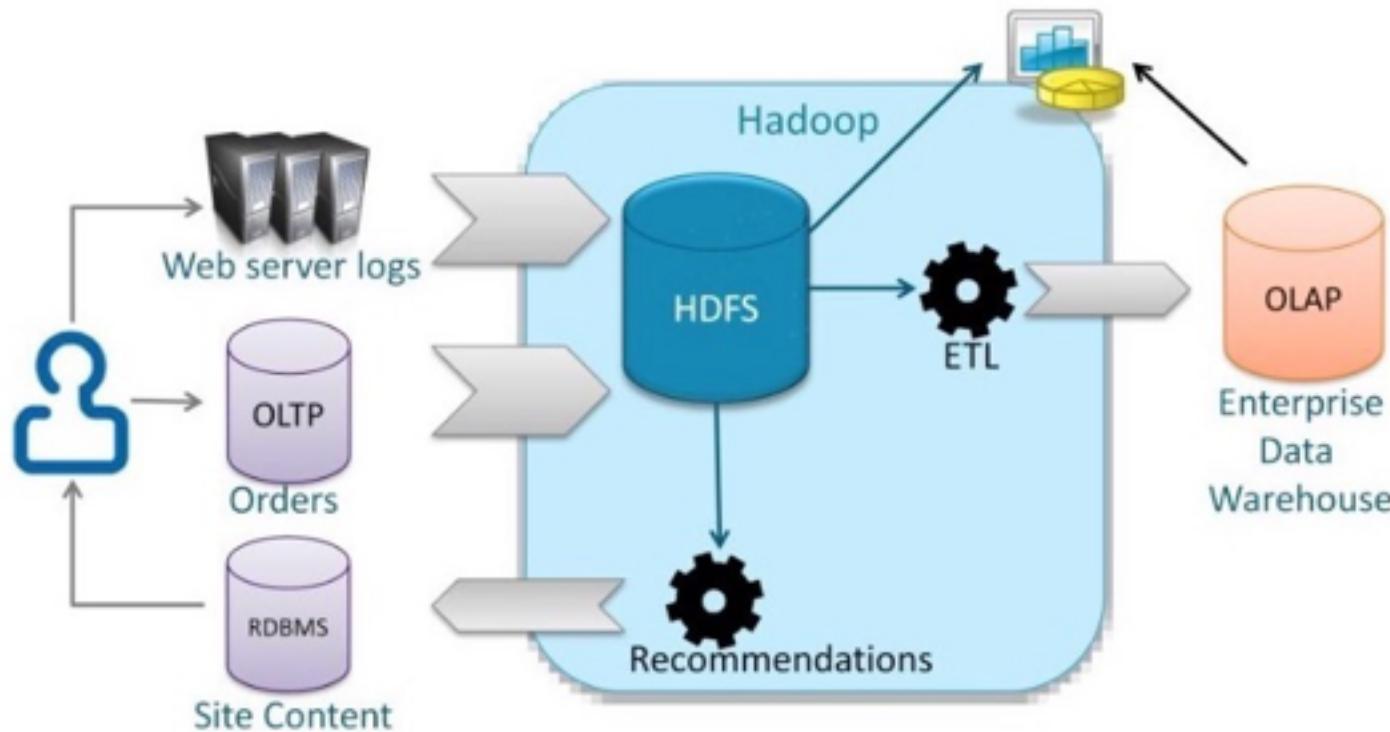
- Relational Database Management Systems (RDBMSs) have many strengths
 - Ability to handle complex transactions
 - Ability to process hundreds or thousands of queries per second
 - Real-time delivery of results
 - Simple but powerful query language

RDBMS Weaknesses

- There are some areas where RDBMS are less ideal
 - Data schema is determined before data is ingested
 - Can make ad-hoc data collection difficult
 - Upper bound on data storage of 100s of terabytes
 - Practical upper bound on data in a single query of 10s of terabytes

Using Hadoop to Augment Existing Databases

- With Hadoop you can store and process all your data
 - The ‘Enterprise Data Hub’



Benefits of Hadoop

- Processing power scales with data storage
 - As you add more nodes for storage, you get more processing power ‘for free’
- Views do not need pre materialization
 - Ad-hoc full or partial dataset queries are possible
- Total query size can be multiple petabytes

Hadoop Tradeoffs

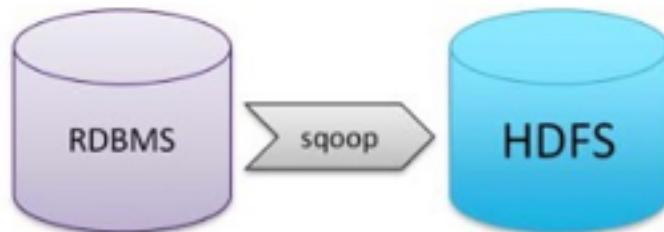
- Cannot serve interactive queries
 - The fastest MapReduce job will still take several seconds to run
- Less powerful updates
 - No transactions
 - No modification of existing records

File Servers and Hadoop

- Choice of destination medium depends on the expected access patterns
 - Sequentially read, append-only data: HDFS
 - Random Access: file server
- HDFS can crunch sequential data faster
- Offloading data to HDFS leaves more room on file servers for ‘interactive’ data
- Use the right tool for the job!

Importing Data from an RDBMS to HDFS

- Typical Scenario: data stored in an RDBMS is needed in a MapReduce job
 - LookUp tables
 - Legacy Data
- Possible to read directly from an RDBMS in your Mapper
 - Can lead to the equivalent of a distributed denial of service (DDoS) attack on your RDBMS
 - In practice - don't do it!
- Better idea: use Sqoop to import the data into HDFS beforehand



Sqoop: SQL to Hadoop (1)



- Sqoop: open source tool
 - Now a top-level Apache Software Foundation project
- Imports tables from an RDBMS into HDFS
 - Just one table
 - All tables in a database
 - Just portions of a table
 - Sqoop supports WHERE clause
- Uses MapReduce to actually import the data
 - ‘Throttles’ the number of Mappers to avoid DDoS scenarios
 - Uses 10 Mappers by default
 - Value is configurable
- Uses a JDBC interface
 - Should work with virtually any JDBC-compatible database

Sqoop: SQL to Hadoop (2)

- Imports data to HDFS as delimited text files or SequenceFiles
 - Default is a comma-delimited text file
- Can be used for incremental data imports
 - First import retrieves all rows in a table
 - Subsequent imports retrieve just rows created since the last import
- Generate a class file which can encapsulate a row of the imported data
 - Useful for serializing and deserializing data in subsequent MapReduce jobs

Basic Syntax

- Standard syntax:

```
sqoop tool-name [tool-options]
```

- Tools Include:

```
import  
import-all-tables  
list-tables
```

- Options include:

```
--connect  
--username  
--password
```

Example

- Import a table called employees from a database called personnel in a MySQL RDBMS

```
$ sqoop import --username fred --password derf \
  --connect jdbc:mysql://database.example.com/personnel \
  --table employees
```

- As above, but only records with an ID greater than 1000

```
$ sqoop import --username fred --password derf \
  --connect jdbc:mysql://database.example.com/personnel \
  --table employees \
  --where "id > 1000"
```

Other Options

- Sqoop can take data from HDFS and insert it into an already-existing table in an RDBMS with the command

```
$ sqoop export [options]
```

- For general Sqoop Help:

```
$ sqoop help
```

- For help on a particular command:

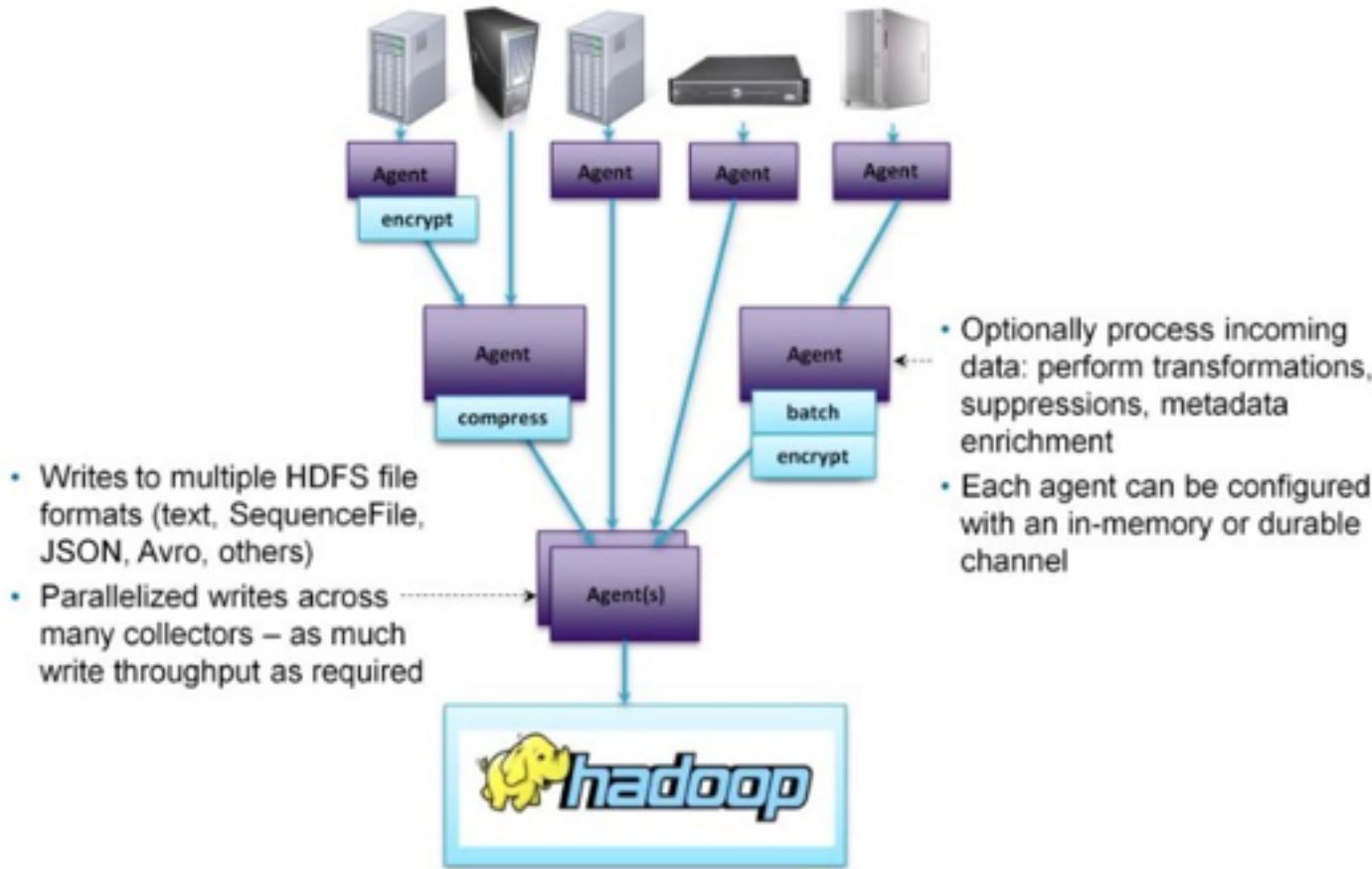
```
$ sqoop help command
```

Flume

- Flume is a distributed, reliable, available service for efficiently moving large amounts of data as it is produced
 - Ideally situated to gathering logs from multiple systems and inserting them into HDFS as they are generated
- Flume is Open Source
- Flume's design goals:
 - Reliability
 - Scalability
 - Extensibility



High-Level Overview



Agent Characteristics

- Each Flume agent has a source, a sink and a channel
- Source
 - Tells the node where to receive data from
- Sink
 - Tells the node where to send data to
- Channel
 - A queue between the Source and Sink
 - Can be in-memory only or ‘Durable’
 - Durable channels will not lose data if power is lost

Reliability

- Channels provide Flume's reliability
- Memory Channel
 - Data will be lost if power is lost
- File Channel
 - Data stored on disk
 - Guarantees durability of data in face of a power loss
- Data transfer between Agents and Channels is transactional
 - A failed data transfer to a downstream agent rolls back and retires
- Can configure multiple Agents with the same task
 - e.g., two Agents doing the job of one “collector” - if one agent fails then upstream agents would fail over

Scalability

- **Scalability**
 - The ability to increase system performance linearly by adding more resources to the system
 - Flume scales horizontally
 - As load increases, more machines can be added to the configuration

Extensibility

- Extensibility
 - The ability to add new functionality to a system
- Flume can be extended by adding Sources and Sinks to existing storage layers or data platforms
 - General sources include data from files, syslog, and standard output from a process
 - General Sinks include files on the local filesystem or HDFS
 - Developers can write their own Source or Sinks

Usage Patterns

- Flume is typically used to ingest log files from real-time systems such as Web Servers, firewalls and mailservers into HDFS
- Currently in use in many large organization, ingesting millions of events per day
- At least one organization is using Flume to ingest over 200 million events per day
- Flume is typically installed and configured by a system administrator

FuseDFS and HttpFS: Motivation

- Many applications generate data which will ultimately reside in HDFS
- If Flume is not an appropriate solution for ingesting the data, some other method must be used
- Typically this is done as a batch process
- Problem: many legacy systems do not ‘understand’ HDFS
 - Difficult to write to HDFS if the application is not written in Java
 - May not have Hadoop installed on the system generating the data
- We need some way for these systems to access HDFS

FuseDFS

- FuseDFS is based on FUSE (Filesystem in USER space)
- Allows you to mount HDFS as a ‘regular’ filesystem
- Note: HDFS limitations still exist!
 - Not intended as a general-purpose filesystem
 - Files are write-once
 - Not optimized for low latency
- FuseDFS included as part of the Hadoop distribution

HttpFS

- Provides an HTTP/HTTPS REST interface to HDFS
 - Supports both reads and writes from/to HDFS
 - Can be accessed from within a program
 - Can be used via command-line tools such as curl or wget
- Client accesses the HttpFS server
 - HttpFS server then accesses HDFS
- Example:
 - `curl -i -L http://httpfs-host:14000/webhdfs/v1/user/foo/README.txt?op=OPEN`
 - returns the contents of the HDFS /user/foo/README.txt file

Chapter 6

HBASE OVERVIEW

HBase

- Column-Oriented data store, known as “Hadoop Database”
- Supports random real-time CRUD operations (unlike HDFS)
- Distributed - designed to serve large tables
 - Billions of rows and millions of columns
- Runs on a cluster of commodity hardware
 - Server hardware, not laptop/desktops

HBase

- Open-source, written in Java
- Type of “NoSQL” DB
 - Does not provide a SQL based access
 - Does not adhere to Relational Model for storage
- Horizontally scalable
 - Automatic sharding

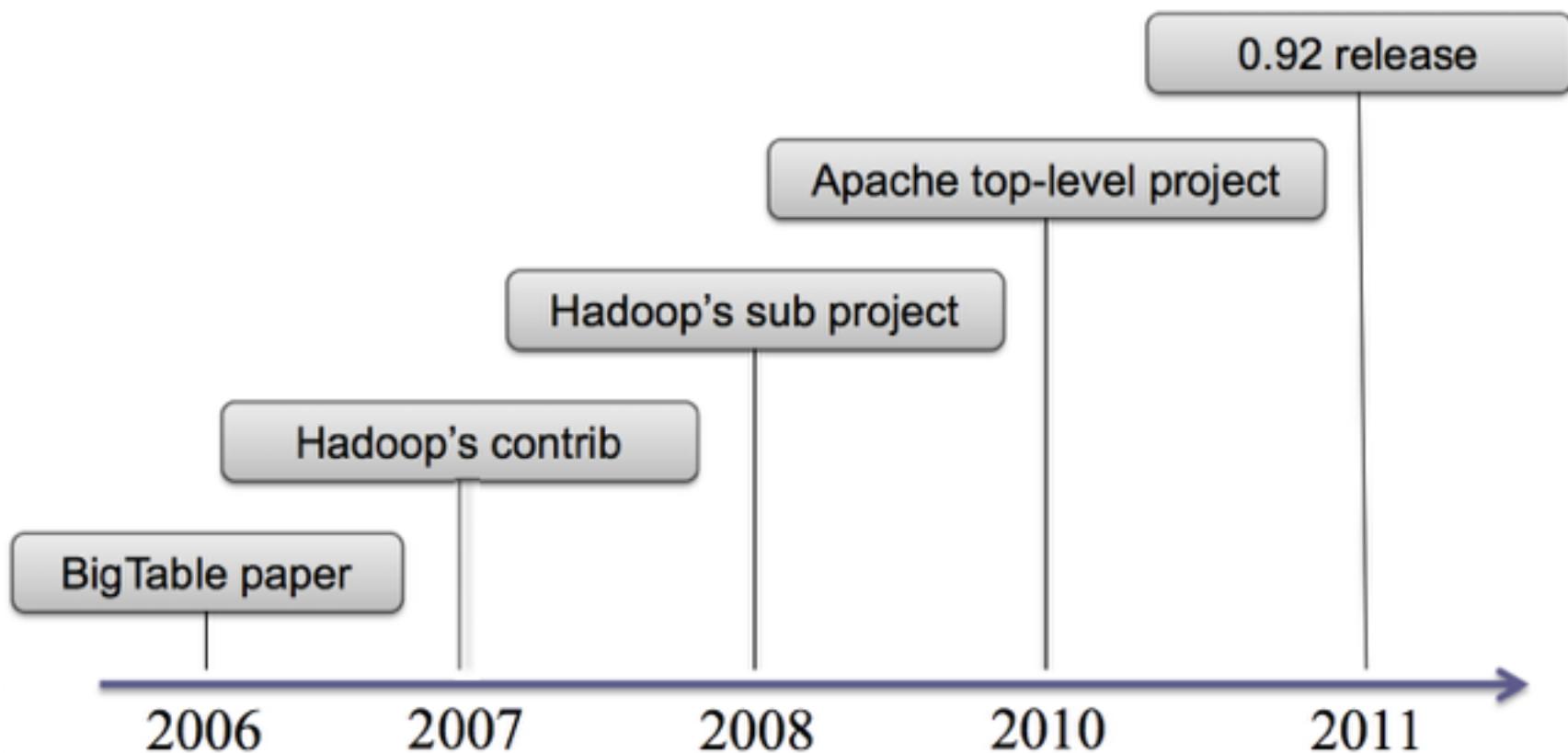
HBase

- **Strongly consistent reads and writes**
- **Automatic fail-over**
- **Simple Java API**
- **Integration with Map/Reduce framework**
- **Thrift, Avro and REST-ful Web-services**

HBase

- Based on Google's Bigtable
 - <http://labs.google.com/papers/bigtable.html>
- Just like BigTable is built on top of Google's File System (GFS), HBase is implemented on top of HDFS

HBase History



Who Uses HBase?

- Here is a very limited list of well known names
 - Facebook
 - Adobe
 - Twitter
 - Yahoo!
 - Netflix
 - Meetup
 - Stumbleupon
 - You??



When To Use HBase

- Not suitable for every problem
 - Compared to RDBMs has **VERY simple and limited API**
- Good for large amounts of data
 - 100s of millions or billions of rows
 - If data is too small all the records will end up on a single node leaving the rest of the cluster idle

When to Use HBase

- Two well-known use cases
 - Lots and lots of data (already mentioned)
 - Large amount of clients/requests (usually cause a lot of data)
- Great for single random selects and range scans by key
- Great for variable schema
 - Rows may drastically differ
 - If your schema has many columns and most of them are null

When NOT to Use HBase

- Bad for traditional RDBMs retrieval
 - Transactional applications
 - Relational Analytics
 - 'group by', 'join', and 'where column like', etc....

HBase Data Model

- Data is stored in Tables
- Tables contain rows
 - Rows are referenced by a unique key
 - Key is an array of bytes - good news
 - Anything can be a key: string, long and your own serialized data structures
- Rows made of columns which are grouped in column families
- Data is stored in cells
 - Identified by row x column-family x column
 - Cell's content is also an array of bytes

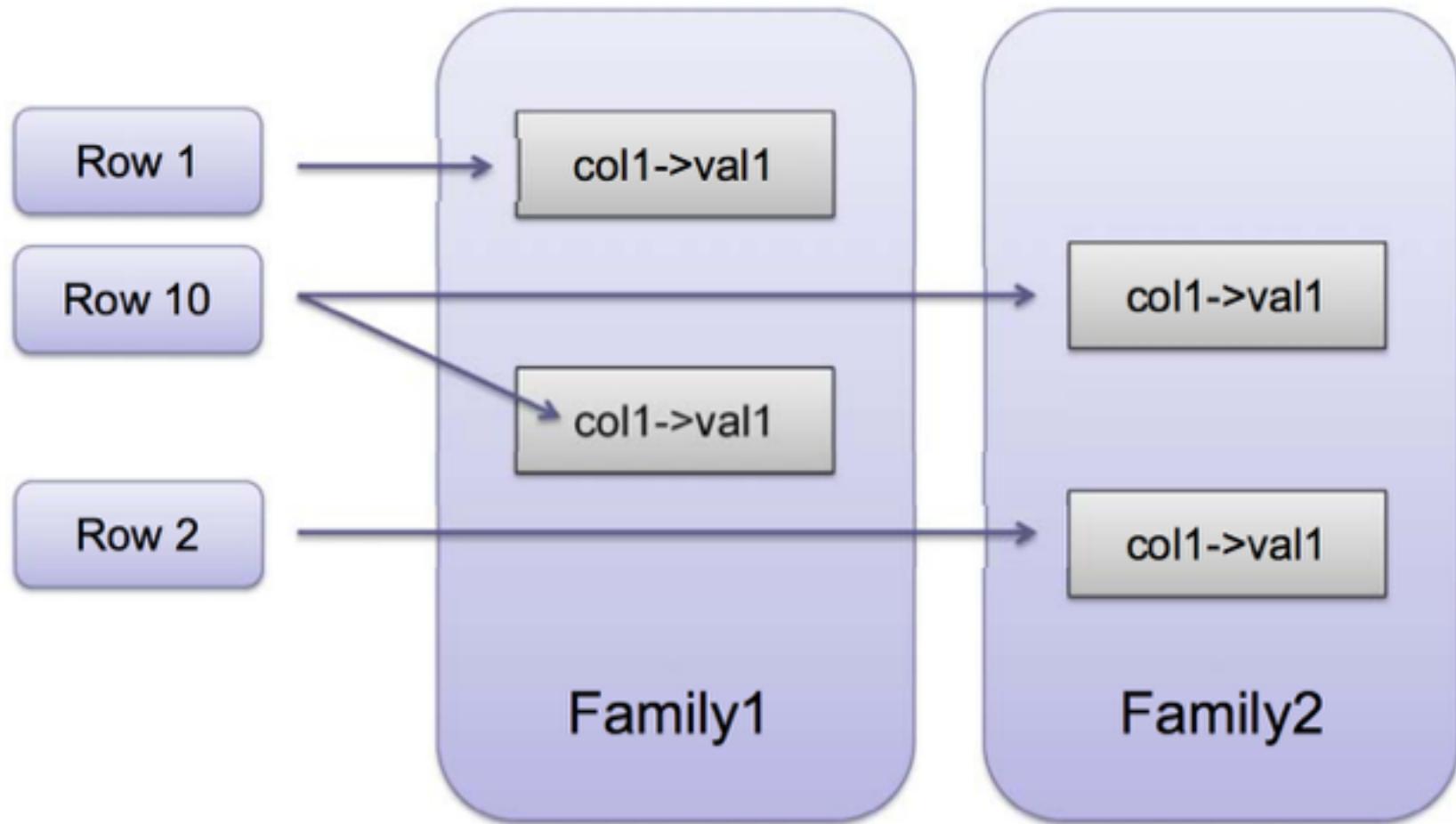
HBase Families

- Rows are grouped into families
 - Labeled as “family:column”
 - Example “user:first_name”
 - Various features are applied to families
 - Compression
 - In-memory option
 - Stored together - in an HFile/StoreFile
- Family definitions are static
 - Created with table
 - Limited to small number of families
 - unlike columns that you can have millions of

HBase Families

- Family name must be composed of printable characters
 - Not bytes, unlike keys and values
- Think of family:column as a tag for a cell value and NOT as a spreadsheet
- Columns on the other hand are NOT static
 - Create new columns at run-time
 - Can scale to millions for a family

Rows Composed Of Cells Stored In Families:Columns

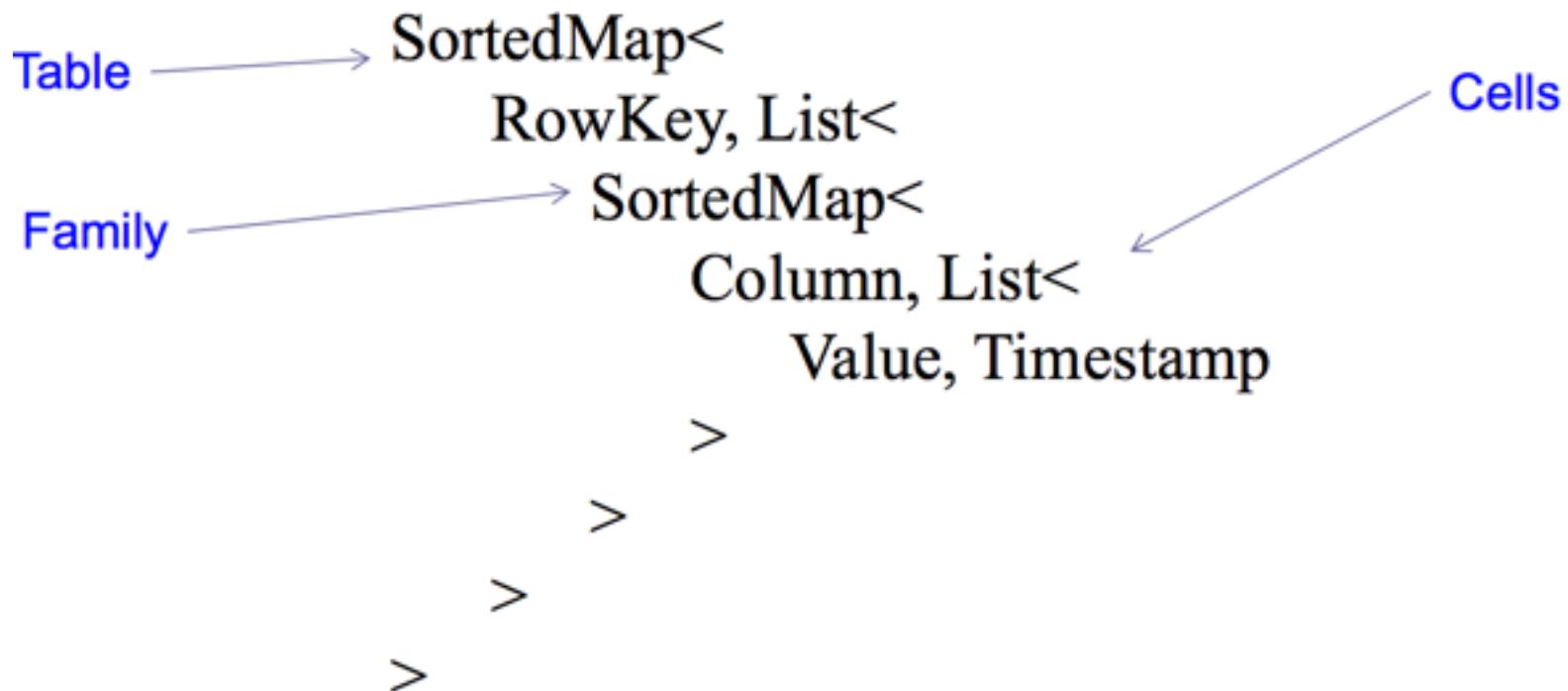


HBase Timestamps

- Cells' values are versioned
 - For each cell multiple versions are kept
 - 3 by default
 - Another dimension to identify your data
 - Either explicitly timestamped by region server or provided by the client
 - Versions are stored in decreasing timestamp order
 - Read the latest first - optimization to read the current value
- You can specify how many versions are kept

HBase Cells

- Value = Table+RowKey+Family+Column+Timestamp
- Programming language style:



HBase Row Keys

- Rows are sorted lexicographically by key
 - Compared on a binary level from left to right
 - For example keys 1,2,3,10,15 will get sorted as
 - 1, 10, 15, 2, 3
- Somewhat similar to Relational DB primary index
 - Always unique

HBase Cells

Row Key	Time stamp	Name Family		Address Family	
		first_name	last_name	number	address
row1	t1	<u>Bob</u>	<u>Smith</u>		
	t5			10	First Lane
	t10			30	Other Lane
	t15			7	<u>Last Street</u>
row2	t20	<u>Mary</u>	Tompson		
	t22			77	One Street
	t30		<u>Thompson</u>		

HBase Cells

- Can ask for
 - Most recent value (default)
 - Specific timestamp
 - Multiple values such as range of timestamps

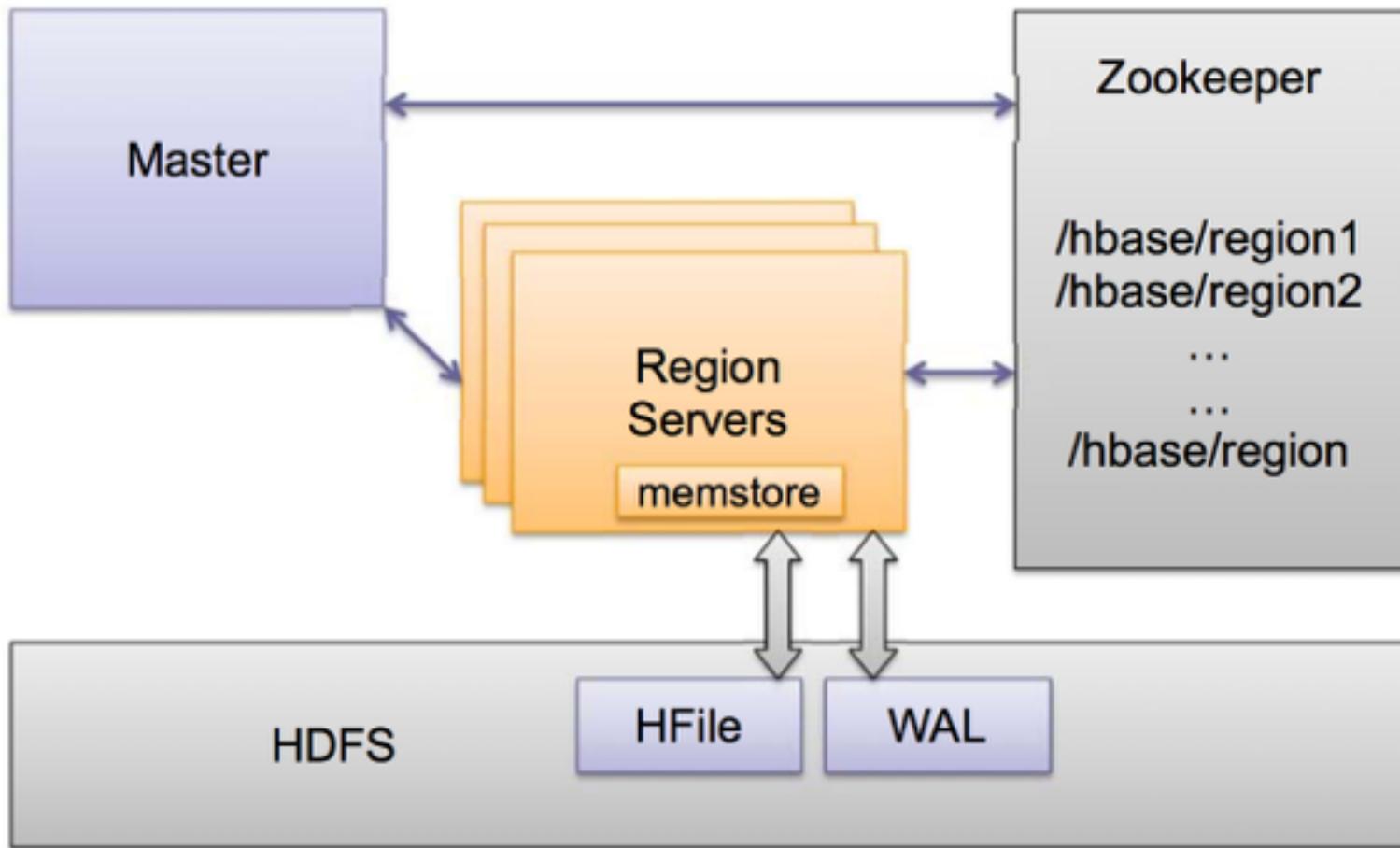
HBase Architecture

- Table is made of regions
- Region - a range of rows stored together
 - Single shard, used for scaling
 - Dynamically split as they become too big and merged if too small
- Region Server- serves one or more regions
 - A region is served by only 1 Region Server

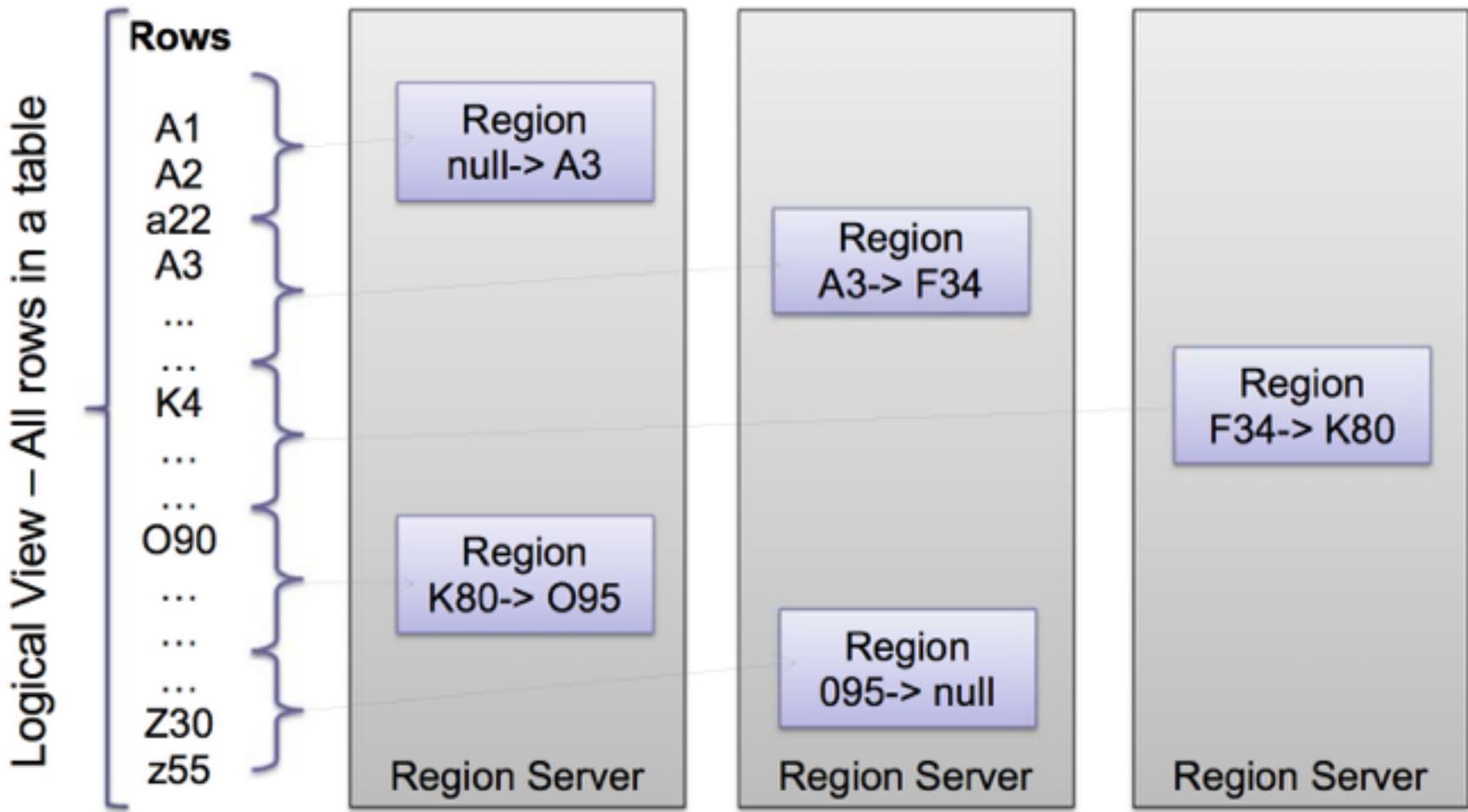
HBase Architecture

- Master Server - daemon responsible for managing HBase cluster, aka Region Servers
- HBase stores its data into HDFS
 - relies on HDFS's high availability and fault-tolerance features
- HBase utilizes Zookeeper for distributed coordination

HBase Components



Rows Distribution Between Region Servers



HBase Regions

- Region is a range of keys
 - start key → stop key (ex. k3cod → odiekd)
 - start key inclusive and stop key exclusive
- Addition of data
 - At first there is only 1 region
 - Addition of data will eventually exceed the configured maximum → the region is split
 - Default is 256MB
 - The region is split into 2 regions at the middle key

HBase Regions

- Regions per server depend on hardware specs, with today's hardware it's common to have:
 - 10 to 1000 regions per Region Server
 - Managing as much as 1GB to 2 GB per region

HBase Regions

- Splitting data into regions allows
 - Fast recovery when a region fails
 - Load balancing when a server is overloaded
 - May be moved between servers
 - Splitting is fast
 - Reads from an original file while asynchronous process performs a split
 - All of these happen automatically without user's involvement

Data Storage

- Data is stored in files called HFiles/StoreFiles
 - Usually saved in HDFS
- HFile is basically a key-value map

Data Storage

- When data is added it's written to a log called Write Ahead Log (WAL) and is also stored in memory (memstore)
- Flush: when in-memory data exceeds maximum value it is flushed to an HFile
 - Data persisted to HFile can then be removed from WAL
 - Region Server continues serving read-writes during the flush operations, writing values to the WAL and memstore

Data Storage

- Recall that HDFS doesn't support updates to an existing file therefore HFiles are immutable
 - Cannot remove key-values out of HFile(s)
 - Over time more and more HFiles are created
- Delete marker is saved to indicate that a record was removed
 - These markers are used to filter the data - to “hide” the deleted records
 - At runtime, data is merged between the content of the HFile and WAL

Data Storage

- To control the number of HFiles and to keep cluster well balanced HBase periodically performs data compactions
 - Minor Compaction: Smaller HFiles are merged into larger HFiles (n-way merge)
 - Fast - Data is already sorted within files
 - Delete markers are not applied
 - Major Compaction:
 - For each region merges all the files within a column-family into a single file
 - Scan all the entries and apply all the deletes as necessary

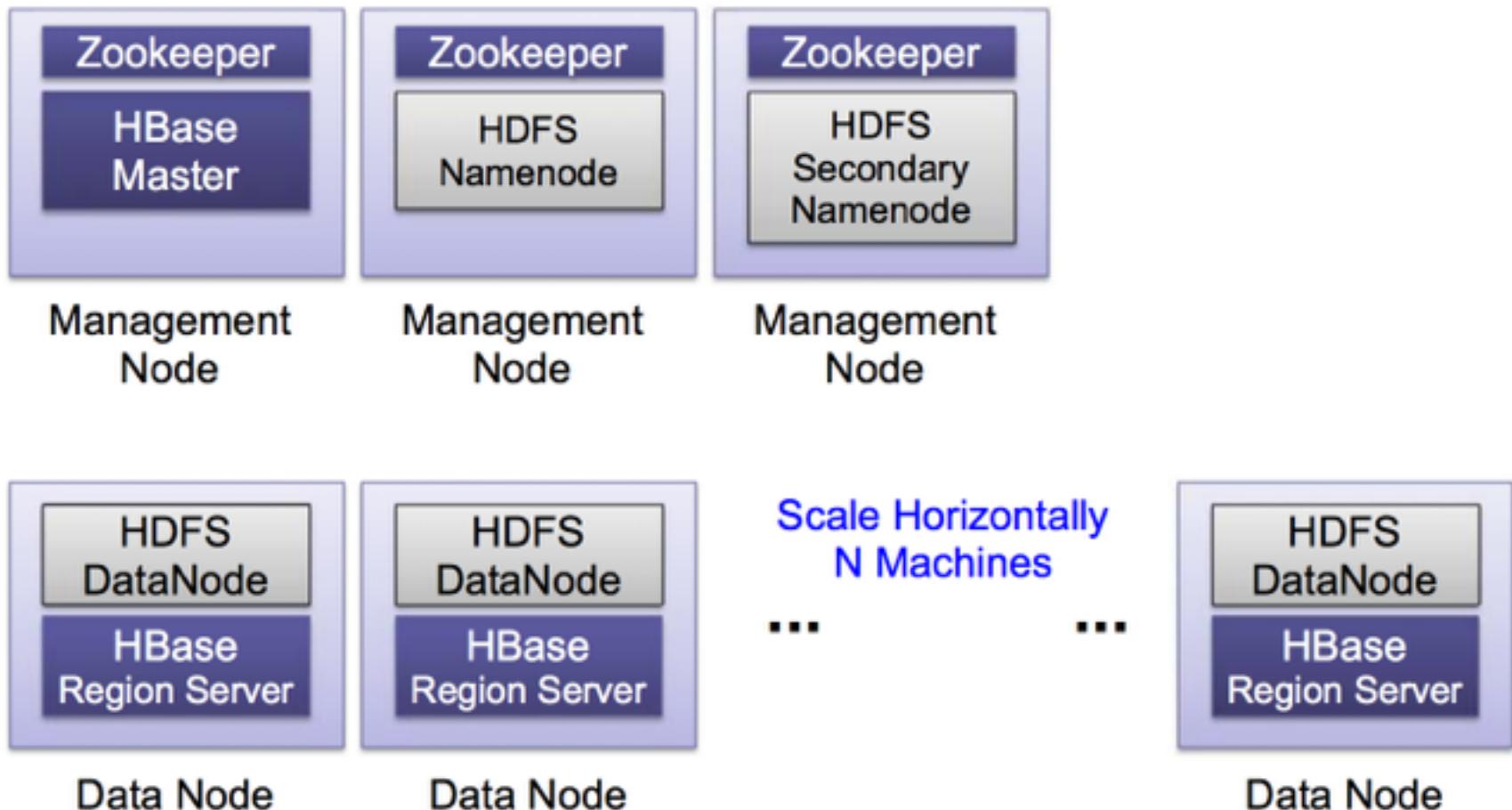
HBase Master

- Responsible for managing regions and their locations
 - Assigns regions to region servers
 - Re-balanced to accommodate workloads
 - Recovers if a region server becomes unavailable
 - Uses Zookeeper - distributed coordination service

HBase Master (cont:)

- Doesn't actually store or read data
 - Clients communicate directly with Region Servers
 - Usually lightly loaded
- Responsible for schema management and changes
 - Adding/Removing tables and column families

HBase Deployment



HBase Access

- HBase Shell
- Native Java API
 - Fastest and very capable options
- Avro Server
 - Apache Avro is also a cross-language schema compiler
 - Requires running Avro Server
- HBql
 - SQL like syntax for HBase
 - <http://www.hbql.com>

HBase Access

- PyHBase
 - python client for HBase Avro interface
- AsyncHBase
 - asynchronous, non-blocking, thread-safe, HBase client
- JPA/JPO access to HBase via DataNucleous
- HBase-DSL
 - Java Library that helps you build queries

HBase Access

- Native API is not the only option
 - REST Server
 - Complete client and admin APIs
 - Requires a REST gateway server
 - Supports many formats: text, xml, json, protocol buffers, raw binary
 - Thrift
 - Apache Thrift is a cross-language schema compiler
 - <http://thrift.apache.org>
 - Requires running Thrift Server

Chapter 7

HBASE SHELL

HBase Management Console

- HBase comes with web based management
 - <http://localhost:60010>
- Both Master and Region servers run web server
 - Browsing Master will lead you to region servers
 - Regions run on port 60030
- Firewall considerations
 - Opening <master_host>:60010 in firewall is not enough
 - Have to open up <region(s)_host>:60030 on every slave host

HBase Shell

- JRuby IRB (Interactive Ruby Shell)
 - HBase commands added
 - If you can do it in IRB you can do it in HBase shell
- To run simply

\$ <hbase_install>/bin/hbase shell

HBase Shell; enter 'help<RETURN>' for list of supported commands.

Type "exit<RETURN>" to leave the HBase Shell

Version 0.90.4-cdh3u2, r, Thu Oct 13 20:32:26 PDT 2011

hbase(main):001:0>

- Puts you into IRB
- Type 'help' to get a listing of commands
 - \$ help “command” e.g. > help “get”

HBase Shell

- Quote all names
 - Table and column names
 - Single quotes for text
 - `hbase> get 't1', 'myRowId'`
 - Double quotes for binary
 - Use hexadecimal representation of that binary value
 - `hbase> get 't1', "key\x03\x3f\xcd"`
- Uses ruby hashes to specify parameters
 - `{'key1' => 'value1', 'key2' => 'value2', ...}`

```
hbase> get 'UserTable', 'userId1', {COLUMN => 'address:str'}
```

HBase Shell

- HBase Shell supports various commands
 - General
 - status, version
 - Data Definition Language (DDL)
 - alter, create, describe, disable, drop, enable, exists, is_disabled, is_enabled, list
 - Data Manipulation Language (DML)
 - count, delete, deleteall, get, get_counter, incr, put, scan, truncate

HBase Shell (cont:)

- Cluster administration
 - balancer, close_region, compact, flush, major_compact, move, split, unassign, zk_dump, add_peer, disable_peer, enable_peer, remove_peer, start_replication, stop_replication

HBase Shell - Check Status

- Display cluster's status via status command
 - `hbase> status`
 - `hbase> status 'detailed'`
- Similar information can be found on HBase Web Management Console
 - `http://localhost:60010`

HBase Shell - Check Status

```
hbase> status
1 servers, 0 dead, 3.0000 average load

hbase> status 'detailed'
version 0.90.4-cdh3u2
0 regionsInTransition
1 live servers
    hadoop-laptop:39679 1326056194009
        requests=0, regions=3, usedHeap=30, maxHeap=998
        .META.,,1
            stores=1, storefiles=0, storefileSizeMB=0, ...
        -ROOT-,,0
            stores=1, storefiles=1, storefileSizeMB=0, ...
        Blog,,1326059842133.c1b865dd916b64a6228ecb4f743 ...
0 dead servers
```

Create Table

- Create table called 'Blog' with the following schema
 - 2 families
 - 'info' with 3 columns: 'title', 'author', and 'date'
 - 'content' with 1 column family: 'post'

Blog		
Family:	info:	Columns: title, author, date
	content:	Columns: post

Create Table (cont:)

- Various options to create tables and families
 - hbase> **create 't1', {NAME => 'f1', VERSIONS => 5}**
 - hbase> **create 't1', {NAME => 'f1', VERSIONS => 1, TTL => 2592000, BLOCKCACHE => true}**
 - hbase> **create 't1', {NAME => 'f1'}, {NAME => 'f2'}, {NAME => 'f3'}**
 - hbase> **create 't1', 'f1', 'f2', 'f3'**

```
hbase> create 'Blog', {NAME=>'info'}, {NAME=>'content'}
0 row(s) in 1.3580 seconds
```

Populate Table With Data Records

- Populate data with multiple records

Row Id	info:title	info:author	info:date	content:post
Matt-001	Elephant	Matt	2009.05.06	Do elephants like monkeys?
Matt-002	Monkey	Matt	2011.02.14	Do monkeys like elephants?
Bob-003	Dog	Bob	1995.10.20	People own dogs!
Michelle-004	Cat	Michelle	1990.07.06	I have a cat!
John-005	Mouse	John	2012.01.15	Mickey mouse.

- Put command format:

```
hbase> put 'table', 'row_id', 'family:column', 'value'
```

Populate Table With Data Records

```
# insert row 1
put 'Blog', 'Matt-001', 'info:title', 'Elephant'
put 'Blog', 'Matt-001', 'info:author', 'Matt'
put 'Blog', 'Matt-001', 'info:date', '2009.05.06'
put 'Blog', 'Matt-001', 'content:post', 'Do elephants like monkeys?'
...
...
...
# insert rows 2-4
...
...
#
# row 5
put 'Blog', 'John-005', 'info:title', 'Mouse'
put 'Blog', 'John-005', 'info:author', 'John'
put 'Blog', 'John-005', 'info:date', '1990.07.06'
put 'Blog', 'John-005', 'content:post', 'Mickey mouse.'
```

1 put statement per cell

Access data - count

- Access Data
 - count: display the total number of records
 - get: retrieve a single row
 - scan: retrieve a range of rows
- Count is simple
 - hbase> count 'table_name'
 - Will scan the entire table
 - Specify count to display every n rows. Default is 1000
 - hbase> count 't1', INTERVAL => 10

3. Access data - count

```
hbase> count 'Blog', {INTERVAL=>2}
Current count: 2, row: John-005
Current count: 4, row: Matt-002
5 row(s) in 0.0220 seconds
```

```
hbase> count 'Blog', {INTERVAL=>1}
Current count: 1, row: Bob-003
Current count: 2, row: John-005
Current count: 3, row: Matt-001
Current count: 4, row: Matt-002
Current count: 5, row: Michelle-004
```

Affects how often
count is displayed

Access data - get

- Select single row with 'get' command
 - `hbase> get 'table', 'row_id'`
 - Returns an entire row
 - Requires table name and row id
 - Optional: timestamp or time-range, and versions
- Select specific columns
 - `hbase> get 't1', 'r1', {COLUMN => 'c1'}`
 - `hbase> get 't1', 'r1', {COLUMN => ['c1', 'c2', 'c3']}`

Access data - get

- Select specific timestamp or time-range
 - hbase> get 't1', 'r1', {TIMERANGE => [ts1, ts2]}
 - hbase> get 't1', 'r1', {COLUMN => 'c1', TIMESTAMP => ts1}
- Select more than one version
 - hbase> get 't1', 'r1', {VERSIONS => 4}

Access data - get

```
hbase> get 'Blog', 'unknownRowId'  
COLUMN          CELL  
0 row(s) in 0.0250 seconds
```

Row Id doesn't exist

```
hbase> get 'Blog', 'Michelle-004'  
COLUMN          CELL  
content:post    timestamp=1326061625690, value=I have a cat!  
info:author     timestamp=1326061625630, value=Michelle  
info:date       timestamp=1326061625653, value=1990.07.06  
info:title      timestamp=1326061625608, value=Cat  
4 row(s) in 0.0420 seconds
```

Returns ALL the columns, displays 1 column per row!!!

Access data - get

```
hbbase> get 'Blog', 'Michelle-004',
          {COLUMN=>['info:author','content:post']}
```

COLUMN	CELL
content:post	timestamp=1326061625690, value=I have a cat!
info:author	timestamp=1326061625630, value=Michelle

2 row(s) in 0.0100 seconds

Narrow down to just two columns

```
hbbase> get 'Blog', 'Michelle-004',
          {COLUMN=>['info:author','content:post'],
           TIMESTAMP=>1326061625690}
```

COLUMN	CELL
content:post	timestamp=1326061625690, value=I have a cat!

1 row(s) in 0.0140 seconds

Only one timestamp matches

Access data - get

```
hbase> get 'Blog', 'Michelle-004',
          {COLUMN=>'info:date', VERSIONS=>2}
COLUMN           CELL
info:date        timestamp=1326071670471, value=1990.07.08
info:date        timestamp=1326071670442, value=1990.07.07
2 row(s) in 0.0300 seconds
```

Asks for the latest two versions

```
hbase> get 'Blog', 'Michelle-004',
          {COLUMN=>'info:date'}
COLUMN           CELL
info:date        timestamp=1326071670471, value=1990.07.08
1 row(s) in 0.0190 seconds
```

By default only the latest version is returned

Access data - Scan

- Scan entire table or a portion of it
- Load entire row or explicitly retrieve column families, columns or specific cells
- To scan an entire table
 - hbase> scan 'table_name'
- Limit the number of results
 - hbase> scan 'table_name', {LIMIT=>1}
- Scan a range
 - hbase> scan 'Blog', {STARTROW=>'startRow', STOPROW=>'stopRow'}
 - Start row is inclusive, stop row is exclusive

Access data - Scan

- Limit what columns are retrieved
 - hbase> scan 'table', {COLUMNS=>['col1', 'col2']}
- Scan a time range
 - hbase> scan 'table', {TIMERANGE => [1303, 13036]}
- Limit results with a filter
 - hbase> scan 'Blog', {FILTER =>
org.apache.hadoop.hbase.filter.ColumnPaginationFilter.new(
1, 0)}

Access data - Scan

Scan the entire table, grab ALL the columns

```
hbase (main):014:0> scan 'Blog'
ROW                  COLUMN+CELL
Bob-003 column=content:post, timestamp=1326061625569,
                  value=People own dogs!
Bob-003 column=info:author, timestamp=1326061625518, value=Bob
Bob-003 column=info:date, timestamp=1326061625546,
                  value=1995.10.20
Bob-003 column=info:title, timestamp=1326061625499, value=Dog
John-005      column=content:post, timestamp=1326061625820,
                  value=Mickey mouse.
John-005      column=info:author, timestamp=1326061625758,
                  value=John
...
Michelle-004    column=info:author, timestamp=1326061625630,
                  value=Michelle
Michelle-004    column=info:date, timestamp=1326071670471,
                  value=1990.07.08
Michelle-004    column=info:title, timestamp=1326061625608,
                  value=Cat
5 row(s) in 0.0670 seconds
```

Access data - Scan

Stop row is exclusive, row ids that start with John will not be included



```
hbase> scan 'Blog', {STOPROW=>'John' }
ROW      COLUMN+CELL
Bob-003  column=content:post, timestamp=1326061625569,
          value=People own dogs!
Bob-003  column=info:author, timestamp=1326061625518,
          value=Bob
Bob-003  column=info:date, timestamp=1326061625546,
          value=1995.10.20
Bob-003  column=info:title, timestamp=1326061625499,
          value=Dog
1 row(s) in 0.0410 seconds
```

Access data - Scan

Only retrieve 'info:title' column

```
hbase> scan 'Blog', {COLUMNS=>'info:title',
                      STARTROW=>'John', STOPROW=>'Michelle'}
ROW          COLUMN+CELL
John-005     column=info:title, timestamp=1326061625728,
                         value=Mouse
Matt-001      column=info:title, timestamp=1326061625214,
                         value=Elephant
Matt-002      column=info:title, timestamp=1326061625383,
                         value=Monkey
3 row(s) in 0.0290 seconds
```

Edit data

- Put command inserts a new value if row id doesn't exist
- Put updates the value if the row does exist
- But does it really update?
 - Inserts a new version for the cell
 - Only the latest version is selected by default
 - N versions are kept per cell
 - configured per family at creation:
 - 3 versions are kept by default

```
hbase> create 'table', {NAME => 'family', VERSIONS => 7}
```

Edit data

```
hbase> put 'Blog', 'Michelle-004', 'info:date', '1990.07.06'  
0 row(s) in 0.0520 seconds  
hbase> put 'Blog', 'Michelle-004', 'info:date', '1990.07.07'  
0 row(s) in 0.0080 seconds  
hbase> put 'Blog', 'Michelle-004', 'info:date', '1990.07.08'  
0 row(s) in 0.0060 seconds
```

Update the same exact row with a different value

```
hbase> get 'Blog', 'Michelle-004',  
          {COLUMN=>'info:date', VERSIONS=>3}  
COLUMN           CELL  
info:date       timestamp=1326071670471, value=1990.07.08  
info:date       timestamp=1326071670442, value=1990.07.07  
info:date       timestamp=1326071670382, value=1990.07.06  
3 row(s) in 0.0170 seconds
```

Keeps three versions of each cell by default

Edit data

```
hbase> get 'Blog', 'Michelle-004',  
          {COLUMN=>'info:date', VERSIONS=>2}  
COLUMN      CELL  
info:date   timestamp=1326071670471, value=1990.07.08  
info:date   timestamp=1326071670442, value=1990.07.07  
2 row(s) in 0.0300 seconds
```

Asks for the latest two versions

```
hbase> get 'Blog', 'Michelle-004',  
          {COLUMN=>'info:date'}  
COLUMN      CELL  
info:date   timestamp=1326071670471, value=1990.07.08  
1 row(s) in 0.0190 seconds
```

By default only the latest version is returned

Delete records

- Delete cell by providing table, row id and column coordinates
 - delete 'table', 'rowId', 'column'
 - Deletes all the versions of that cell
- Optionally add timestamp to only delete versions before the provided timestamp
 - delete 'table', 'rowId', 'column', timestamp

Delete records

```
hbase> get 'Blog', 'Bob-003', 'info:date'
COLUMN          CELL
info:date      timestamp=1326061625546, value=1995.10.20
1 row(s) in 0.0200 seconds

hbase> delete 'Blog', 'Bob-003', 'info:date'
0 row(s) in 0.0180 seconds

hbase> get 'Blog', 'Bob-003', 'info:date'
COLUMN          CELL
0 row(s) in 0.0170 seconds
```

Delete records

```
hbase> get 'Blog', 'Michelle-004',
          {COLUMN=>'info:date', VERSIONS=>3}
```

COLUMN	CELL
info:date	timestamp=1326254742846, value=1990.07.08
info:date	timestamp= 1326254739790 , value=1990.07.07
info:date	timestamp=1326254736564, value=1990.07.06

3 row(s) in 0.0120 seconds

3 versions

```
hbase> delete 'Blog', 'Michelle-004', 'info:date', 1326254739791
0 row(s) in 0.0150 seconds
```

1 millisecond after the second version

```
hbase> get 'Blog', 'Michelle-004',
          {COLUMN=>'info:date', VERSIONS=>3}
```

COLUMN	CELL
info:date	timestamp=1326254742846, value=1990.07.08

1 row(s) in 0.0090 seconds

After the timestamp provided at delete statement

Drop table

- Must disable before dropping
 - puts the table “offline” so schema based operations can be performed
 - `hbase> disable 'table_name'`
 - `hbase> drop 'table_name'`
- For a large table it may take a long time....

Drop table

```
hbase> list  
TABLE  
Blog  
1 row(s) in 0.0120 seconds
```

Take the table offline for schema modifications

```
hbase> disable 'Blog'  
0 row(s) in 2.0510 seconds
```

```
hbase> drop 'Blog'  
0 row(s) in 0.0940 seconds
```

```
hbase> list  
TABLE  
0 row(s) in 0.0200 seconds
```

Chapter 8

HBASE JAVA CLIENT API

Java Client API Overview

- HBase is written in Java
- Supports programmatic access to Data Manipulation Language (DML)
- Everything that you can do with HBase Shell and more....
- Java Native API is the fastest way to access HBase

Using Client API

- Create a Configuration object
 - Recall Configuration from HDFS object
 - Adds HBase specific props
- Construct HTable
 - Provide Configuration object
 - Provide table name
- Perform operations
- Close HTable instance
 - Flushes all the internal buffers
 - Releases all the resources

Configuration Object

- Share Configuration instance as much as possible
 - HTables created with the same Connection object will share the same underlying Connection
 - Connection to Zookeeper and HbaseMaster
 - Represented by HConnection class
 - Managed by HConnectionManager class
 - Internally connections are cached in a map that uses Configuration instances as a key

Configuration Object

- When re-using Configuration object for multiple HTable instances
 - Call `HTable.close` so `HConnectionManager` removes this particular instance from the list of HTables requiring `Hconnection`
- When all HTables closed for a particular Connection object then `HConnectionManager` can close the connection
 - If `close` is not called then Connection will be open until the client process ends

HTable

- `org.apache.hadoop.hbase.client.HTable`
 - Client interface to a single HBase table
 - Exposes CRUD operations
 - Simple by design and easy to use :)
- **HTable is NOT thread safe**
 - Create 1 instance per thread
- **HTable supports CRUD batch operations**
 - Not atomic
 - For performance and convenience

HTable (cont:)

- Operations that change data are atomic on per-row-basis
 - There is no built-in concept of a transaction for multiple rows or tables
 - 100% consistency per-row - a client will either write/read the entire row OR have to wait
 - Not a problem when having many readers for a given row - contention when lots of writers attempt to write to the same exact row
 - Doesn't matter on the number of columns written per request, the request will be fully atomic

HTable (cont:)

- Creating HTable instance is not free
 - Actually quite costly - scans catalog .META. Table
 - Checks that table exists and enabled
 - Create once (per thread) and re-use for as long as possible
 - If you find yourself constructing many instances consider using HTablePool (utility to re-use multiple HTable instances)

Create/Save Data to HBase

- Construct HTable instance
 - Create Put instance
- Add cell values and their coordinates
 - Specify family:column as a coordinate
- Call put on HTable instance
- Close HTable

Put Instance

- Put is a save operation for a single row
- Must provide a row id to the constructor
 - Row id is raw bytes: can be anything like number or UUID
 - You are responsible for converting the id to bytes
 - HBase comes with a helper class Bytes that provides static methods which handles various conversions from and to bytes
- See PutExample.java in HadoopSamples

Retrieving Data

- API supports
 - Get a single row by id
 - Get a set of rows by a set of row ids
 - Scan an entire table or a sub set of rows
 - To scan a portion of the table provide start and stop row ids
 - Recall that row-ids are ordered by raw byte comparison
 - In case of string based ids, the order is alphabetical
- Simple API

Retrieve a Single Row

- Construct HTable instance
- Create Get instance
- Optionally narrow down result
 - Specify family:column coordinate
 - Optionally add filters
- Request and get results
 - Call get on HTable
 - Result instance is returned and will contain the data
- Close HTable
- See `GetExample.java` in `HadoopSamples`

Deleting Data

- Deletes are per-row-basis
- Supports batching
 - Batching is not atomic, for performance and for convenience

Deleting Data

- Construct HTable instance
- Create and Initialize Delete
- Call delete on HTable
 - `htable.delete(delete);`
- Close Htable
- See `DeleteExample.java` in `HadoopSamples`

Chapter 9

HBASE JAVA ADMINISTRATIVE API

Java Admin API

- Just like HTable is for client API HBaseAdmin is for administrative tasks
 - `org.apache.hadoop.hbase.client.HBaseAdmin`
- Recall that only Table and Family names have to be pre-defined
 - Columns can be added/deleted dynamically
 - HBase scheme roughly equals table definitions and their column families

Create Table and Column Families

- Construct HBaseAdmin instance
- Create Table's schema
 - Represented by HTableDescriptor class
 - Add column families to table descriptor (HColumnDescriptor)
- Execute create via HBaseAdmin class
- See CreateTable.java in HadoopSamples

Drop Table

- Construct HBaseAdmin instance
- Disable table
 - Table must be taken offline in order to perform any schema modifications
- Delete table
- See DropTableExample.java in HadoopSamples

Chapter 10

ADVANCED JAVA CLIENT API

Scan Data Retrieval

- Utilizes HBase's sequential storage model
 - row ids are stored in sequence
- Allows you to scan
 - An entire table
 - Subset of a table by specifying start and/or stop key
 - Transfers limited amount of rows at a time from the server
 - 1 row at a time by default can be increased

Scan Data Retrieval (cont:)

- You can stop the scan any time
 - Evaluate at each row
 - Scans are similar to iterators

Scan Rows

- Construct HTable instance
 - Create and Initialize Scan
 - Retrieve ResultScanner from HTable
 - Scan through rows
 - Close ResultScanner
 - Close Htable
-
- See `ScanExample.java` in `HadoopSamples`

ResultScanner Lease

- HBase protects itself from Scanners that may hang indefinitely by implementing lease-based mechanism
- Scanners are given a configured lease
 - If they don't report within the lease time HBase will consider client to be dead
 - The scanner will be expired on the server side and it will not be usable
 - Default lease is 60 seconds

```
<property>
  <name>hbase.regionserver.lease.period</name>
  <value>120000</value>
</property>
```

Scanner Caching

- By default `next()` call equals to RPC (Remote Procedure Call) per row
 - Even in case of `next(int rows)`

```
int numOfRPCs = 0;
for ( Result result : scanner) {
    numOfRPCs++;
}
System.out.println("Remote Calls: " + numOfRPCs);
```

- Results in a bad performance for small cells
- Use Scanner Caching to fetch more than a single row per RPC

Scanner Caching

- Three Levels of control
 - HBase Cluster: change for ALL
 - HTable Instance: configure caching per table instance, will affect all the scans created for this table
 - ResultScanner Instance: configure caching per scan instance, will only affect the configured scan
- Can configure at multiple levels if you require the precision

Scanner Batching

- A single row with lots of columns may not fit memory
- HBase Batching allows you to page through columns on per row basis
- Limits the number of columns retrieved from each `ResultScanner.next()` RPC
 - Will not get multiple results
- Set the batch on Scan instance
 - No option on per table or cluster basis
- See `ScanBatchingExample.java` in `HadoopSamples`

Caching and Batching Example

****Batch = 2 and Caching = 9****

RPC

	c1	c2	c3	c4	c5	c6
row1	[]	[]	[]	[]	[]	[]
row2	[]	[]	[]	[]	[]	[]
row3	[]	[]	[]	[]	[]	[]

RPC

	c1	c2	c3	c4	c5	c6
row1	[]	[]	[]	[]	[]	[]
row2	[]	[]	[]	[]	[]	[]
row3	[]	[]	[]	[]	[]	[]

HTable

Source: Lars, George. [HBase The Definitive Guide](#). O'Reilly Media. 2011

Filters

- **get() and scan()** can limit the data retrieved/transferred back to the client
 - via Column families, columns, timestamps, row ranges, etc...
- Filters add further control to limit the data returned
 - For example: select by key or values via regular expressions
 - Optionally added to Get and Scan parameter

Filters

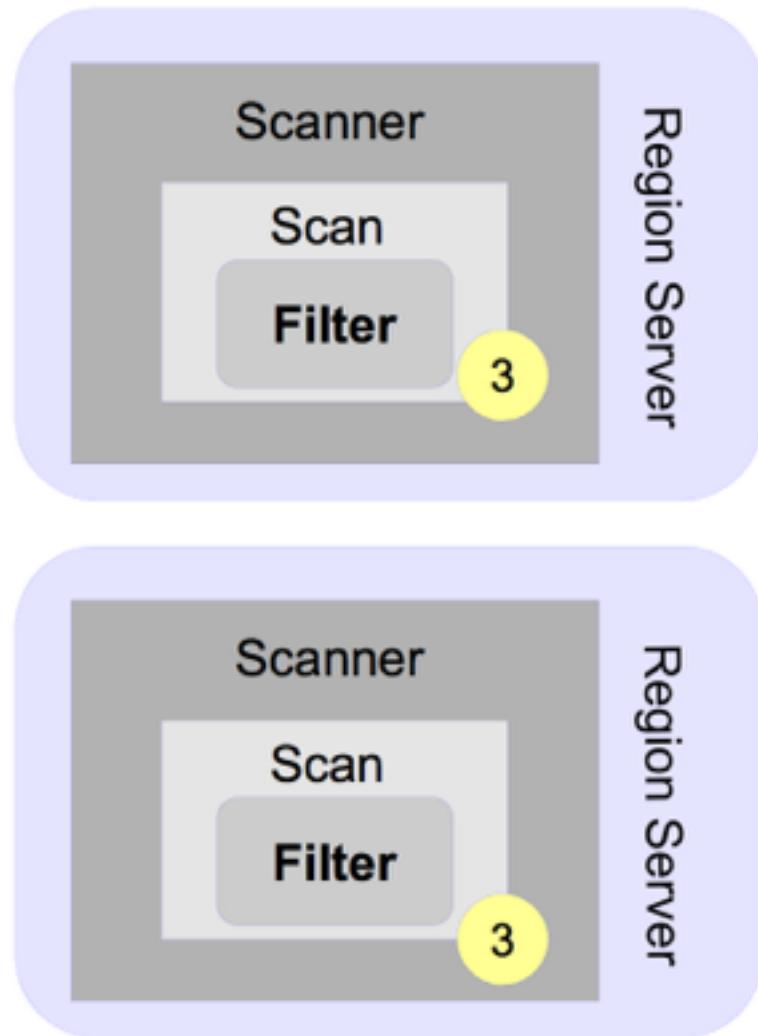
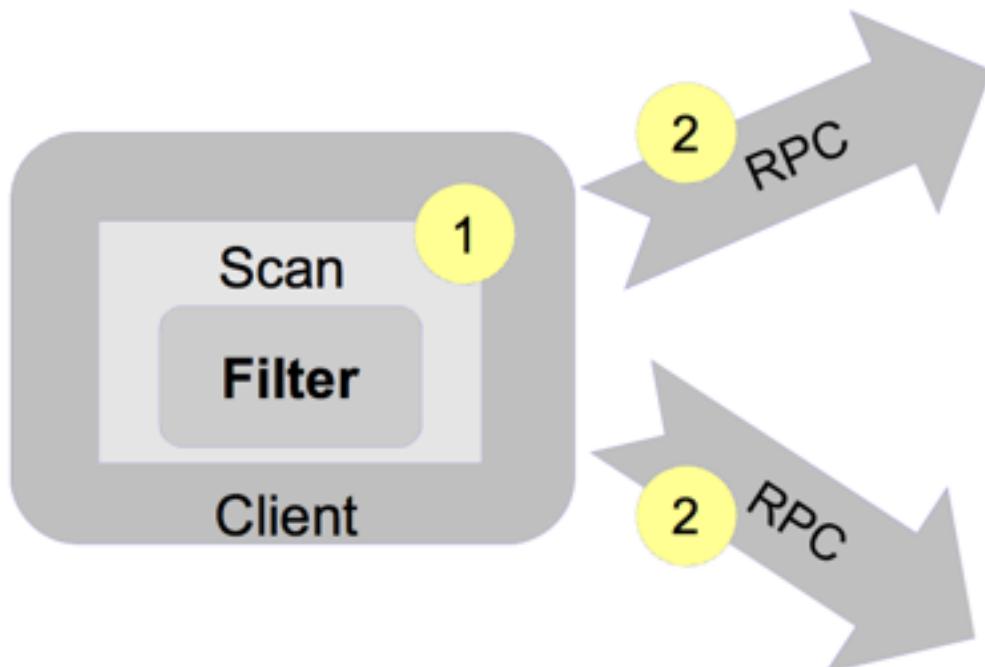
- Implemented by `org.apache.hadoop.hbase.filter.Filter`
 - Use HBase's provided concrete implementations
 - Can implement your own
- See `ValueFilterExample.java` and `FilterListExample.java` in `HadoopSamples`

Filters

- Filters are applied on the server side
 - Reducing amount of data transmitted over the wire
 - Still involves scanning rows
- Execution with filters
 - constructed on the client side
 - serialized and transmitted to the server
 - executed on the server side
- Must exist both on client's and server's CLASSPATH

Execution of a Request with Filter(s)

1. constructed on the client side
2. serialized and transmitted to the server
3. applied on the server side



Sampling of HBase Provided Filters

Filter	Description from HBase API
ColumnPrefixFilter	This filter is used for selecting only those keys with columns that matches a particular prefix.
FilterList	Implementation of Filter that represents an ordered List of Filters
FirstKeyOnlyFilter	A filter that will only return the first KV from each row.
KeyOnlyFilter	A filter that will only return the key component of each KV
PrefixFilter	This filter is used for selecting only those keys with columns that matches a particular prefix.
QualifierFilter	This filter is used to filter based on the column qualifier.
RowFilter	This filter is used to filter based on the key
SkipFilter	A wrapper filter that filters an entire row if any of the KeyValue checks do not pass.
ValueFilter	This filter is used to filter based on column value.

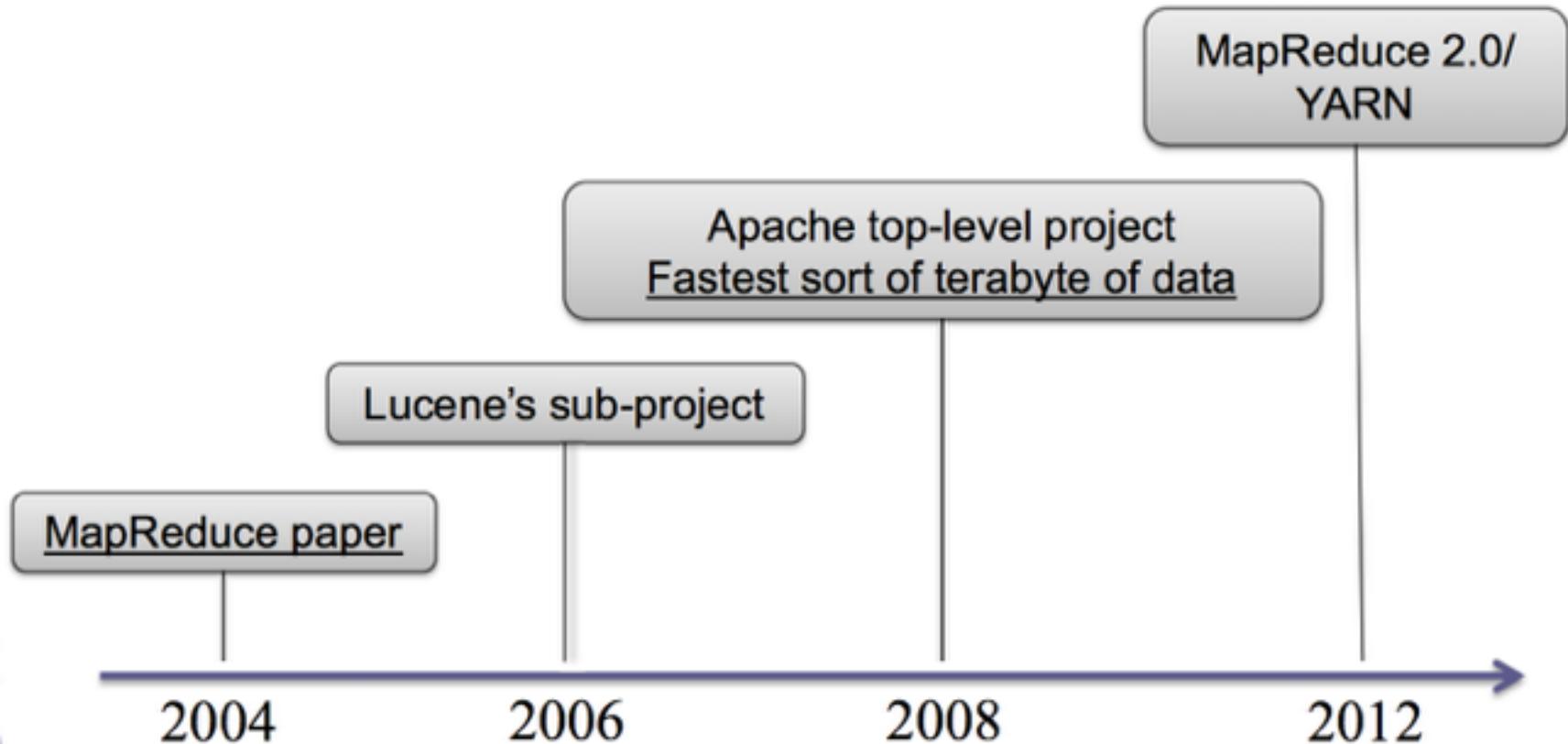
Chapter 11

MAP REDUCE ON YARN OVERVIEW

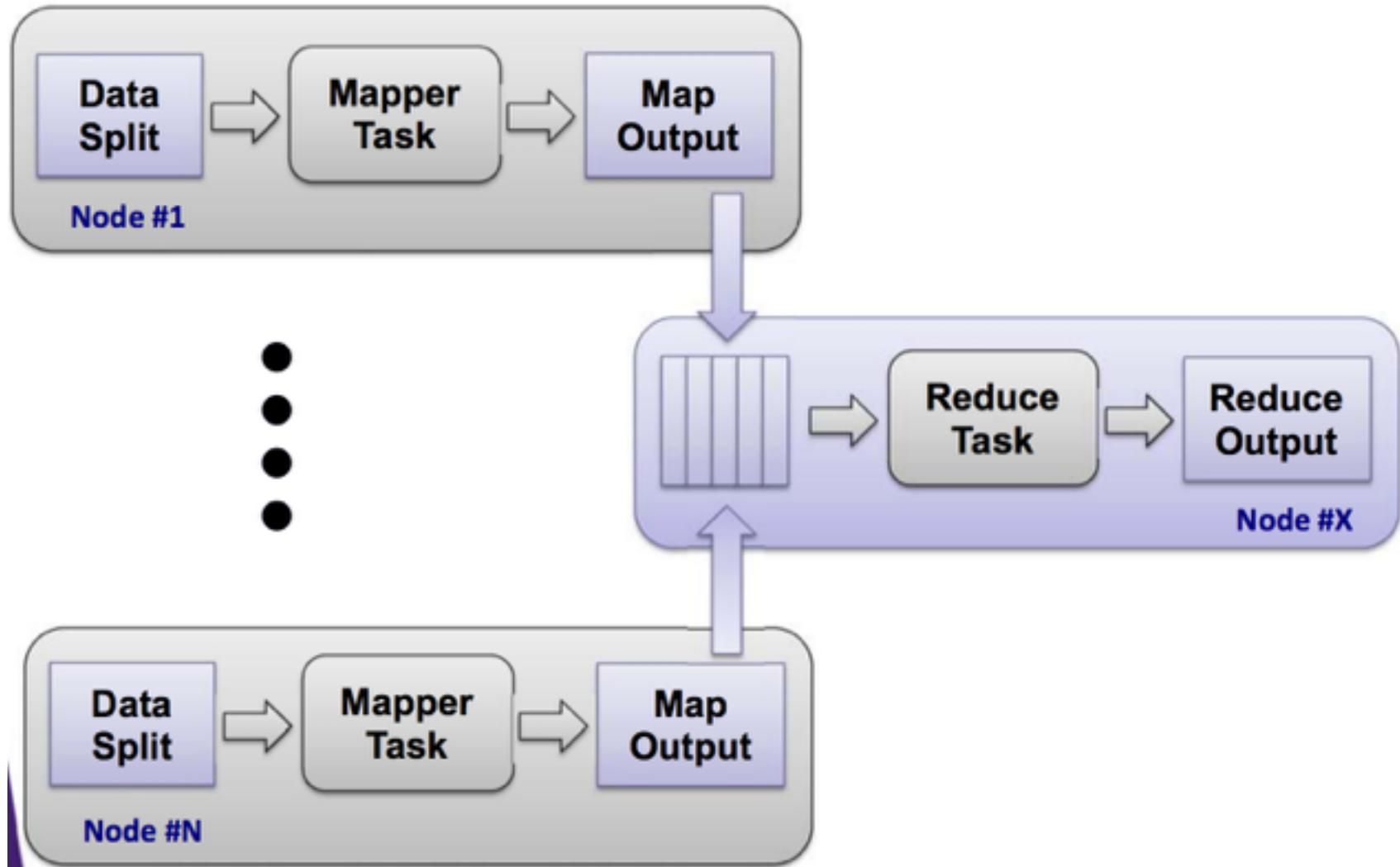
Hadoop MapReduce

- Model for processing large amounts of data in parallel
 - On commodity hardware
 - Lots of nodes
- Derived from functional programming
 - Map and reduce functions
- Can be implemented in multiple languages
 - Java, C++, Ruby, Python (etc...)

Hadoop MapReduce History



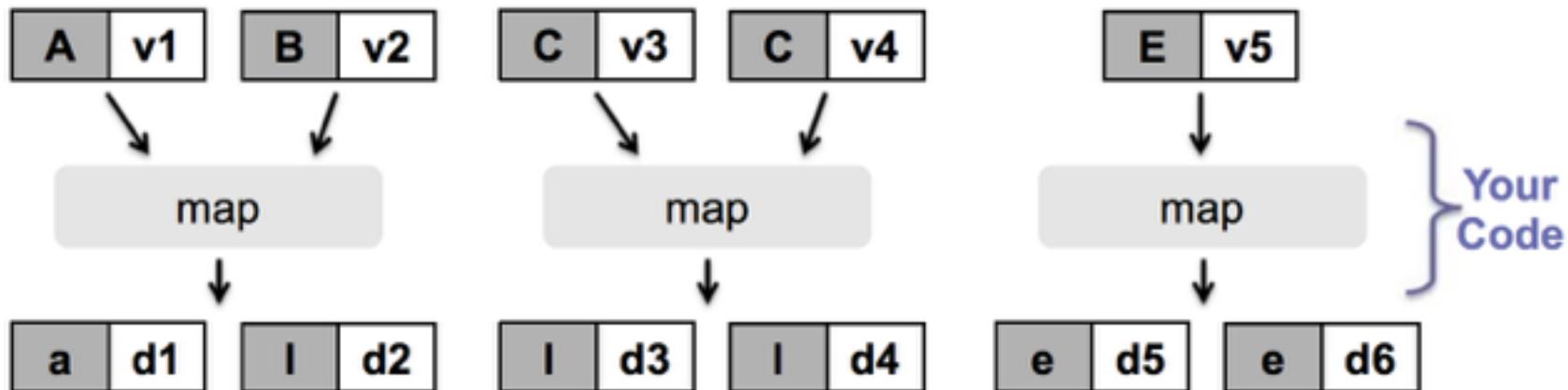
Map Reduce Flow of Data



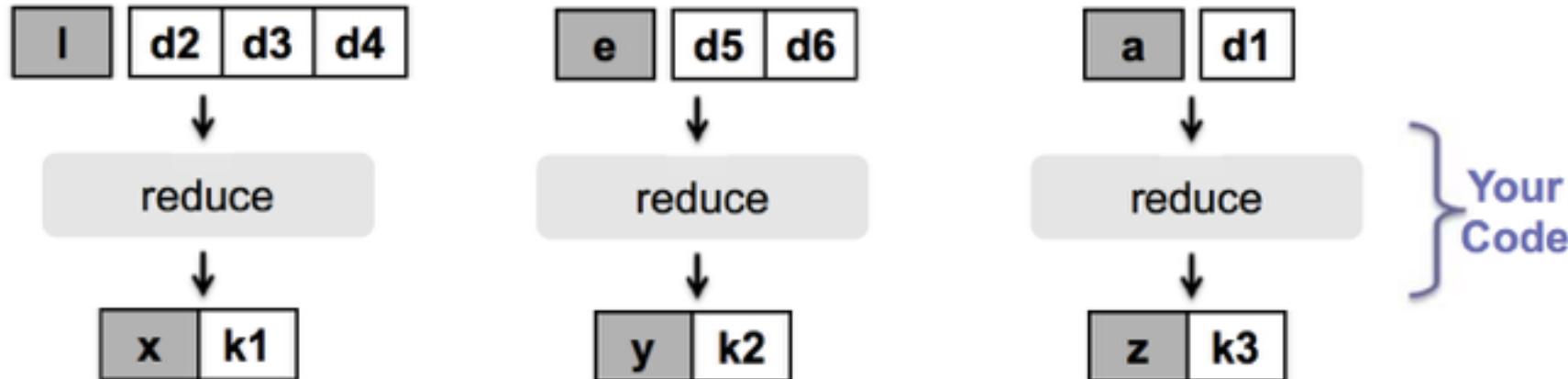
MapReduce Model

- Defines map and reduce functions
 - map: $(K_1, V_1) \rightarrow \text{list } (K_2, V_2)$
 - reduce: $(K_2, \text{list}(V_2)) \rightarrow \text{list } (K_3, V_3)$
 - Map function is applied to every input key-value pair
 - Map function generates intermediate key-value pairs
 - Intermediate key-values are sorted and grouped by key
 - Reduce is applied to sorted and grouped intermediate key-values
 - Reduce emits result key-values

MapReduce Model/Framework



MapReduce Shuffle and Sort: sort and group by output key



MapReduce Framework

- Takes care of distributed processing and coordination
- Scheduling
 - Jobs are broken down into smaller chunks called tasks. These tasks are scheduled
- Task Localization with Data
 - Framework strives to place tasks on the nodes that host the segment of data to be processed by that specific task
 - Code is moved to where the data is

MapReduce Framework

- Error Handling
 - Failures are an expected behavior so tasks are automatically re-tried on other machines
- Data Synchronization
 - Shuffle and Sort barrier re-arranges and moves data between machines
 - Input and output are coordinated by the framework

Map Reduce 2.0 on YARN

- Yet Another Resource Negotiator (YARN)
- Various applications can run on YARN
 - MapReduce is just one choice (the main choice at this point)
 - <http://wiki.apache.org/hadoop/PoweredByYarn>
- YARN was designed to address issues with MapReduce1
 - Scalability issues (max ~4,000 machines)
 - Inflexible Resource Management
 - MapReduce1 had slot based model

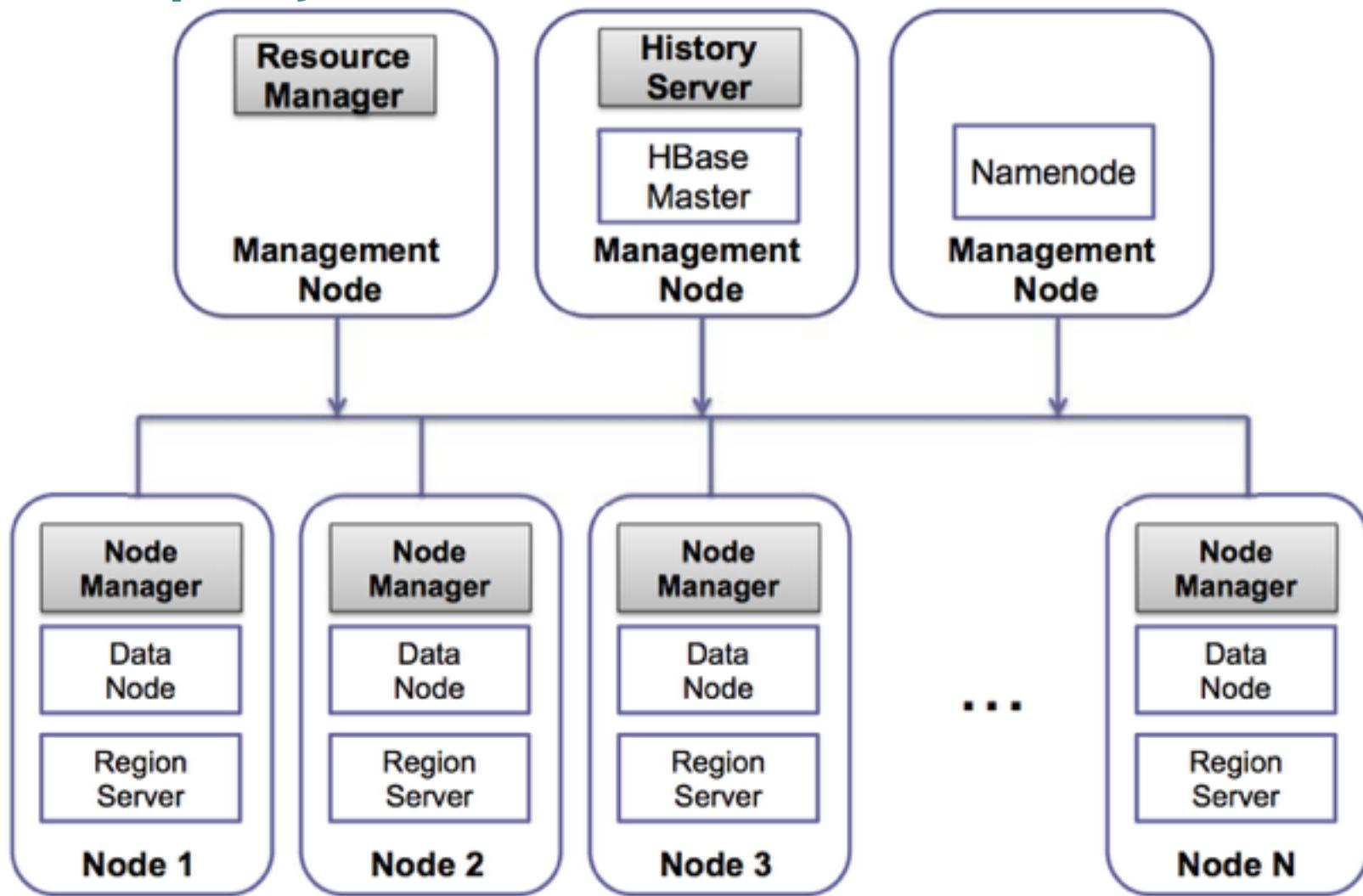
Daemons

- YARN Daemons
 - Node Manager
 - Manages resources of a single node
 - There is one instance per node in the cluster
 - Resource Manager
 - Manages Resources for a Cluster
 - Instructs Node Manager to allocate resources
 - Application negotiates for resources with Resource Manager
 - Only one instance of Resource Manager

Daemons (cont:)

- MapReduce Specific Daemon
 - MapReduce History Server
 - Archives Jobs' metrics and meta-data

Sample YARN Daemons Deployments with HDFS/HBase



YARN Configuration

Config File	Description
yarn-env.sh	A bash script where YARN environment variables are specified. For example, configure log directory here.
yarn-site.xml	Hadoop configuration file where majority of properties are specified for YARN daemons. Configures Resource Manager, Node Manager and History Server.
slaves	A list of nodes where Node Manager daemons are started; one host per line.
mapred-site.xml	MapReduce specific properties go here. This is the application specific configuration file; an application is MapReduce in this case.

Note: YARN will also utilize core-site.xml and hadoop-env.sh which were covered in HDFS lecture

YARN Web-UI

- Resource Manager Web-UI
 - Cluster resource usage, job scheduling, and current running jobs
 - Runs on port 8088 by default
- Application Proxy Web-UI
 - Provides information about the current job
 - Runs as a part of Resource Manager Web-UI by default
 - After completion, jobs get exposed by History Server

YARN Web-UI

- Node Manager Web-UI
 - Single Node information and current containers being executed
 - Runs on port 8042 by default
- MapReduce History Server Web-UI
 - Provides history and details of past MapReduce jobs
 - Runs on port 19888 by default

Command Line Tools

- <hadop_install>/bin/yarn
 - Execute code with a jar
 - \$yarn jar jarFile [mainClass] args...
 - Print out CLASSPATH: \$yarn classpath
 - Resource Manager admin: \$yarn rmadmin
- <hadop_install>/bin/mapred
 - \$mapred job
 - Get information about jobs
 - Kill Jobs

\$ yarn jar jarFile [mainClass] args...

```
$ yarn jar
```

```
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-  
mapreduce-examples-2.0.0-cdh4.0.0.jar pi 5 5
```

Examples jar files
shipped with hadoop

pi is the program that
computes pi

Specify number of
mappers

Number of
samples; artifact
of pi application

\$ yarn rmadmin

- Runs ResourceManager admin client
- Allows to refresh and clear resources

```
$ yarn rmadmin -refreshNodes
```



Resource Manager will refresh its information
about all the Node Managers

\$mapred job

- Command line interface to view job's attributes
- Most of the information is available on Web-UI

```
$ mapred job -list
```

List Jobs that are currently running

```
$ mapred job -status job_1340417316008_0001
```

Retrieve job's status by Job ID

Chapter 12

DEVELOPING FIRST MAPREDUCE JOB

MapReduce

- **Job** - execution of map and reduce functions to accomplish a task
 - Equal to Java's main
- **Task** - single Mapper or Reducer
 - Performs work on a fragment of data

First Map Reduce Job

- StartsWithCount Job
 - Input is a body of text from HDFS
 - In this case hamlet.txt
 - Split text into tokens
 - For each first letter sum up all occurrences
 - Output to HDFS

Hadoop's InputFormats

- Hadoop eco-system is packaged with many InputFormats
 - TextInputFormat
 - NLineInputFormat
 - DBInputFormat
 - TableInputFormat (HBASE)
 - StreamInputFormat
 - SequenceFileInputFormat
- Configured on a Job object
 - `job.setInputFormatClass(XXInputFormat.class)`

TextInputFormat

- Plain Text Input
- Default format

Split: Single HDFS block (can be configured)

Record: Single line of text; linefeed or carriage-return used to locate end of line

Key: LongWritable - Position in the file

Value: Text - line of text

** Please see StartsWithCountJob for sample usage

NLineInputFormat

- Same as TextInputFormat but splits equal to configured N lines

Split:	N lines; configured via <i>mapred.line.input.format</i> or <i>NLineInputFormat.setNumLinesPerSplit(job, 100);</i>
Record:	Single line of text
Key:	LongWritable - Position in the file
Value:	Text - line of text

** Please see StartsWithCountJob_NLineInput for sample usage

TableInputFormat

- Converts data in HTable to format consumable to MapReduce
- Mapper must accept proper key/values

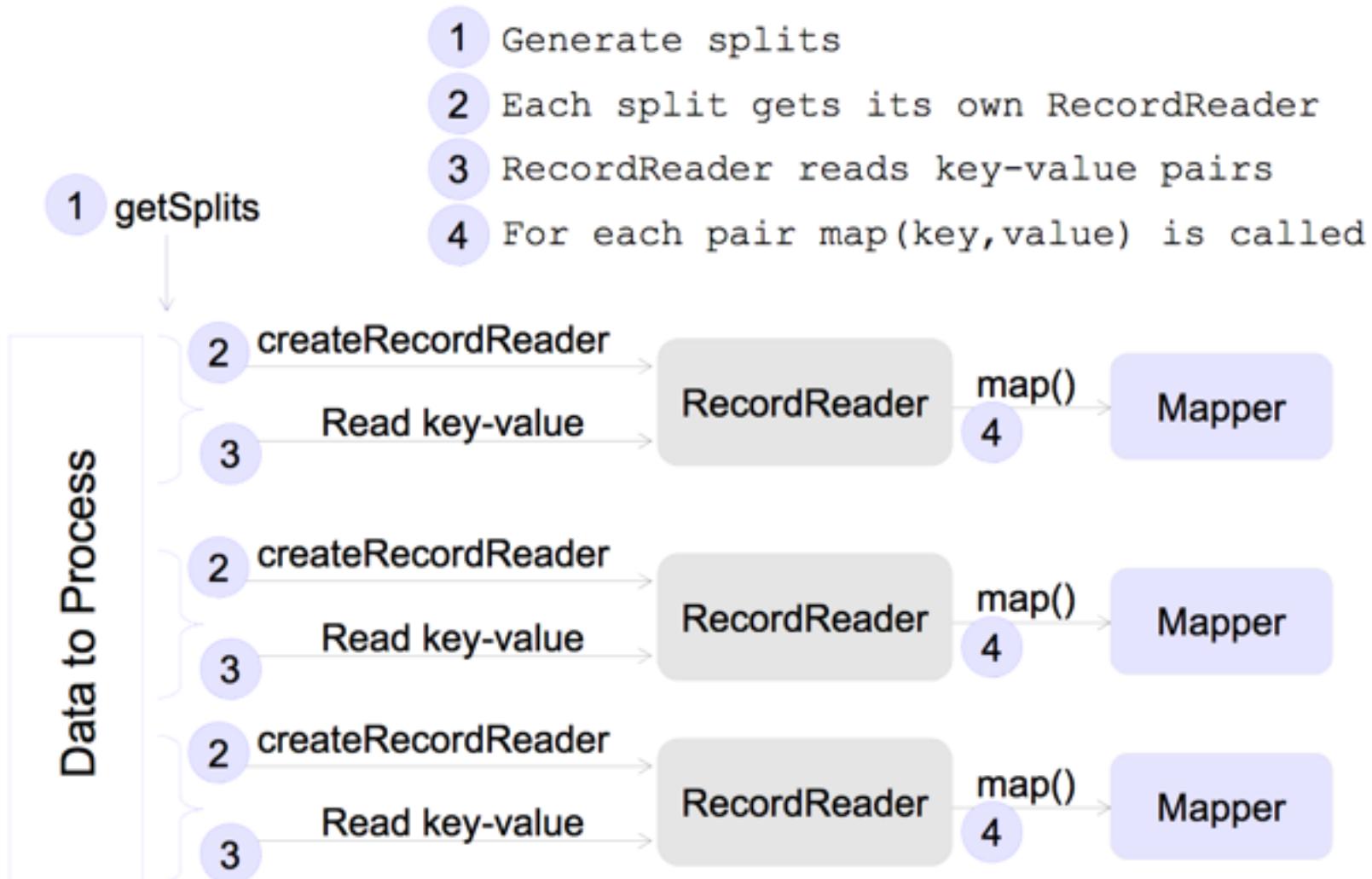
Split: Rows in one HBase Region (provided Scan may narrow down the result)

Record: Row, returned columns are controlled by a provided scan

Key: ImmutableBytesWritable

Value: Result (HBase class)

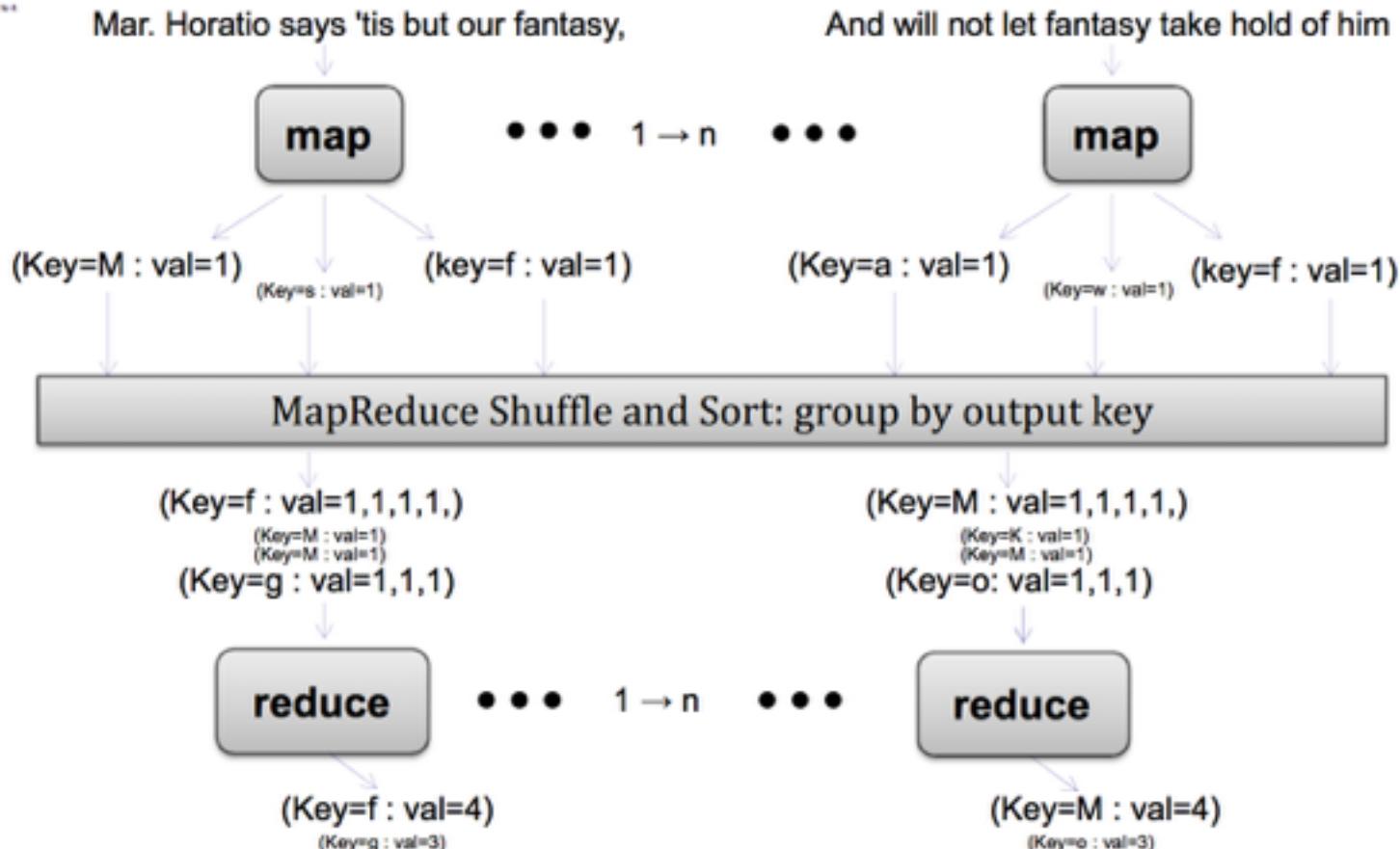
Framework's Usage of InputFormat Implementation



Word Count Job

Mar. What, has this thing appear'd again to-night?
Or, is it my fancy? 'tis but our fantasy,
And will not let itself take hold of him.
Touching this dreadful sight, I have seen of us,
Therefore I have shamed him long,
Until he will not mention of his right,
That I appear'd this apparition-sight.
Hor. Fie, fie, fie, 'tis not appear'd.

MapReduce breaks text into lines feeding each line into map functions



StartsWithCount Job

- Configure the Job
 - Specify Input, Output, Mapper, Reducer and Combiner
- Implement Mapper
 - Input is text - a line from hamlet.txt
 - Tokenize the text and emit first character with a count of 1 - <token, 1>
- Implement Reducer
 - Sum up counts for each letter
 - Write out the result to HDFS
- Run the job

Configure Job

- Job class
 - Encapsulates information about a job
 - Controls execution of the job
- A job is packaged within a jar file
 - Hadoop Framework distributes the jar on your behalf
 - Needs to know which jar file to distribute
 - The easiest way to specify the jar that your job resides in is by calling `job.setJarByClass`
 - Hadoop will locate the jar file that contains the provided class

Configure Job - Specify Input

- Can be a file, directory or a file pattern
 - Directory is converted to a list of files as an input
- Input is specified by implementation of `InputFormat` - in this case `TextInputFormat`
 - Responsible for creating splits and a record reader
 - Controls input types of key-value pairs, in this case `LongWritable` and `Text`
 - File is broken into lines, mapper will receive 1 line at a time

Side Note - Hadoop IO Classes

- Hadoop uses it's own serialization mechanism for writing data in and out of network, database or files
 - Optimized for network serialization
 - A set of basic types is provided
 - Easy to implement your own
- org.apache.hadoop.io package
 - LongWritable for Long
 - IntWritable for Integer
 - Text for String
 - Etc...

Configure Job - Specify Output

- **OutputFormat** defines specification for outputting data from Map/Reduce job
- Count job utilizes an implemenation of OutputFormat - **TextOutputFormat**
 - Define output path where reducer should place its output
 - If path already exists then the job will fail
 - Each reducer task writes to its own file
 - By default a job is configured to run with a single reducer
 - Writes key-value pair as plain text

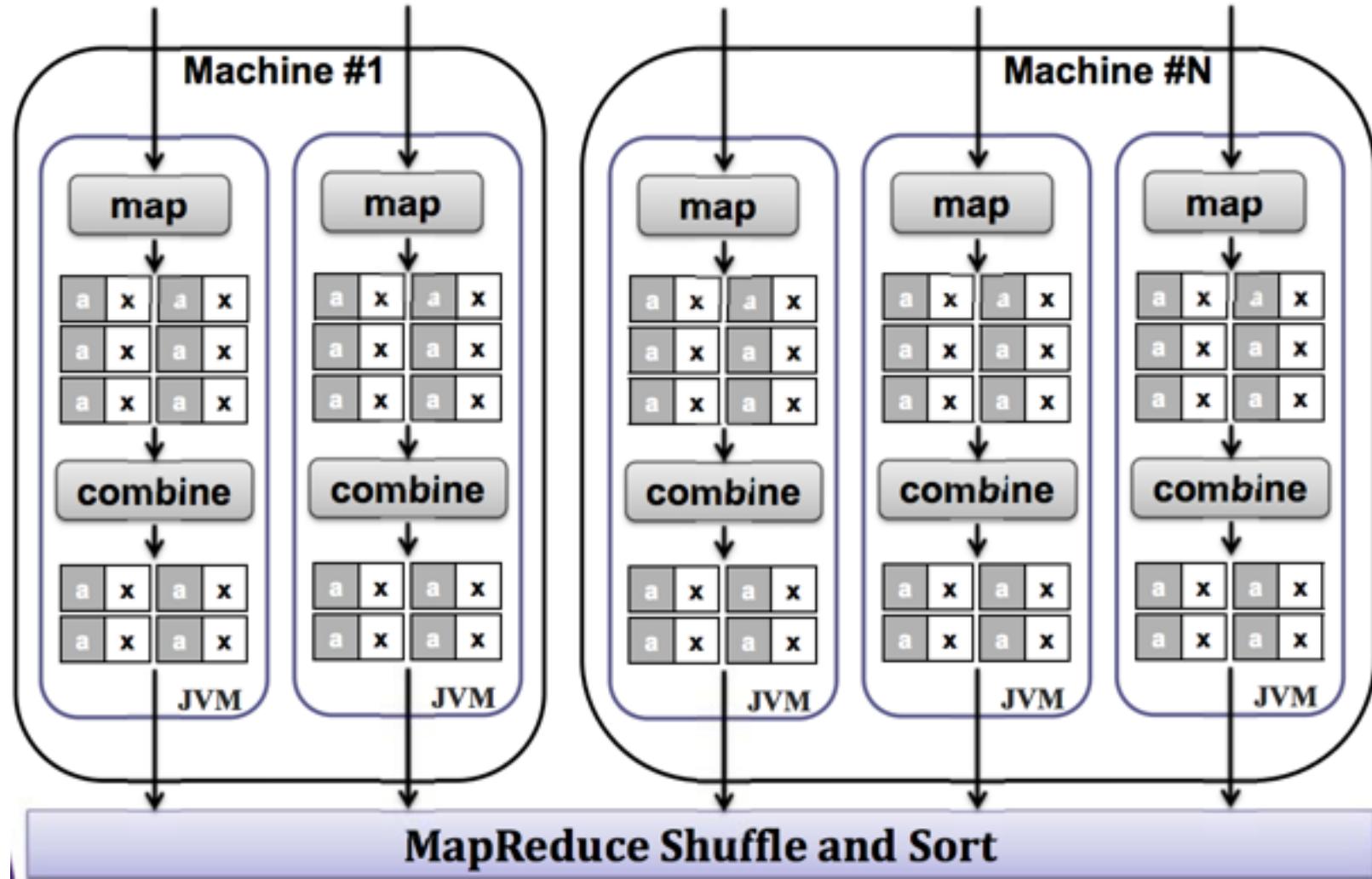
Configure Job - Specify Output

- Specify the output key and value types for both mapper and reducer functions
 - Many times the same type
 - If types differ then use
 - `setMapOutputKeyClass()`
 - `setMapOutputValueClass()`

Configure Job

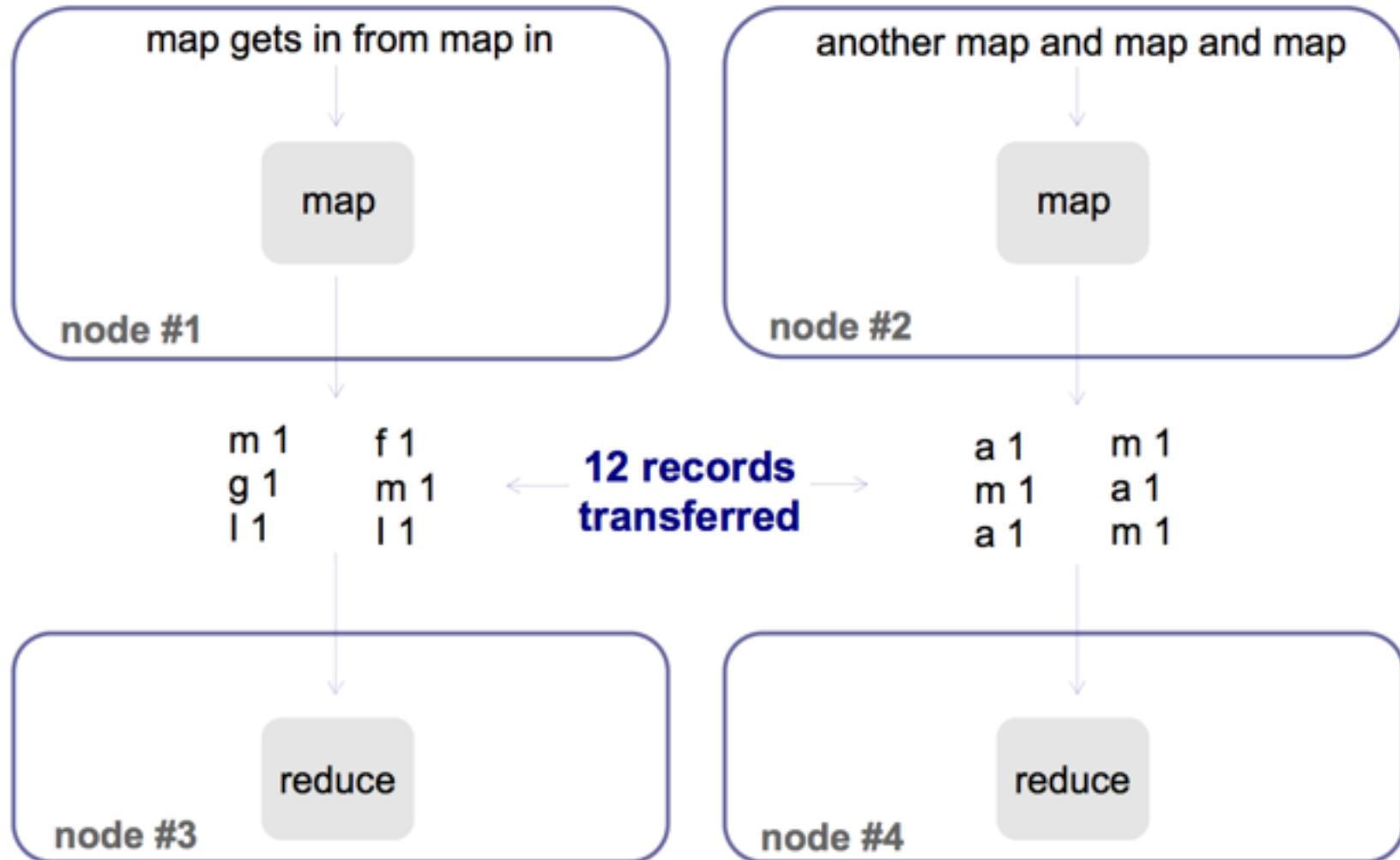
- Specify Mapper, Reducer and Combiner
 - At a minimum will need to implement these classes
 - Mappers and Reducer usually have same output key

Combiner Data Flow



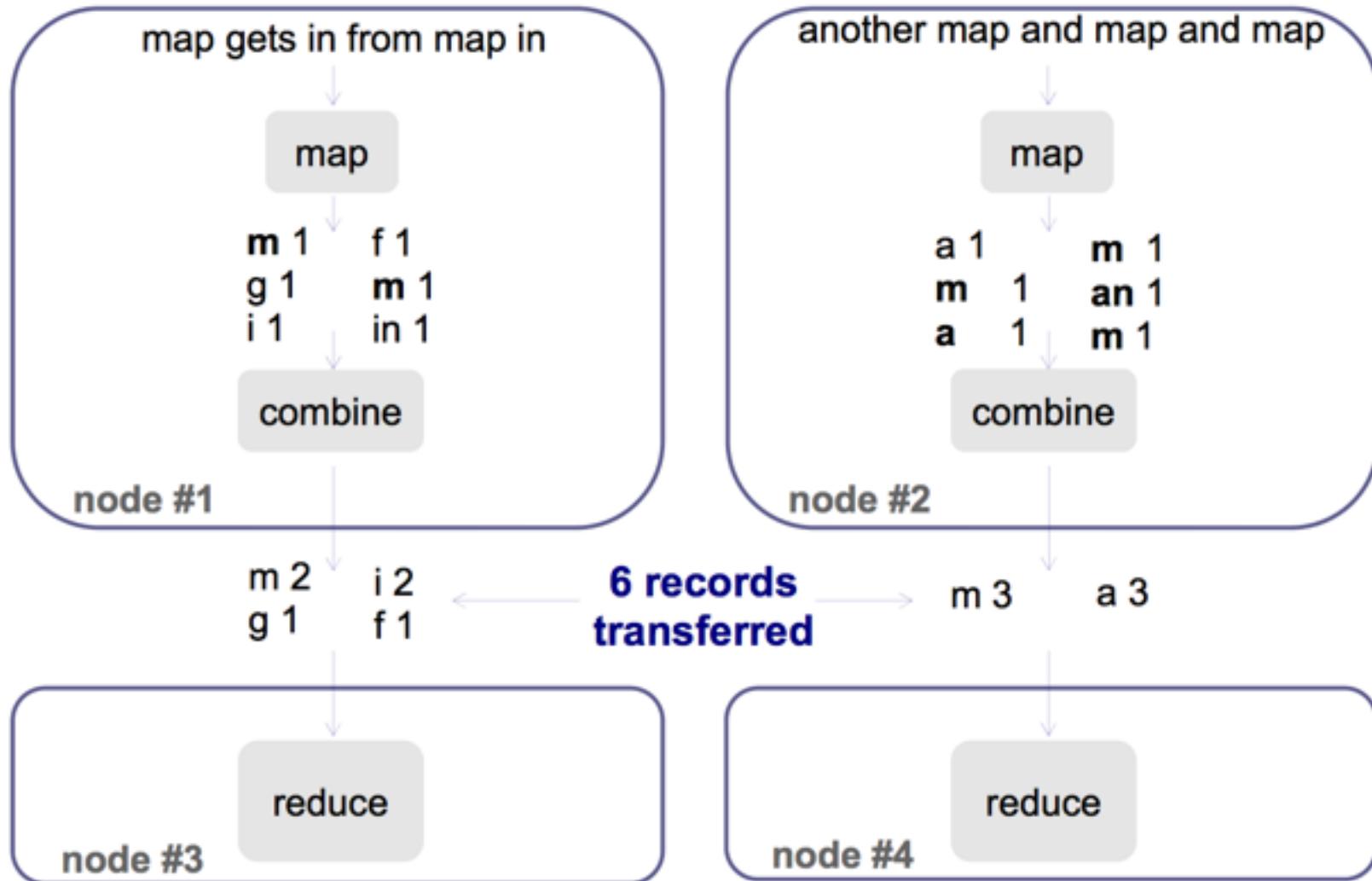
Sample StartsWithCountJob

Run without Combiner



Sample StartsWithCountJob

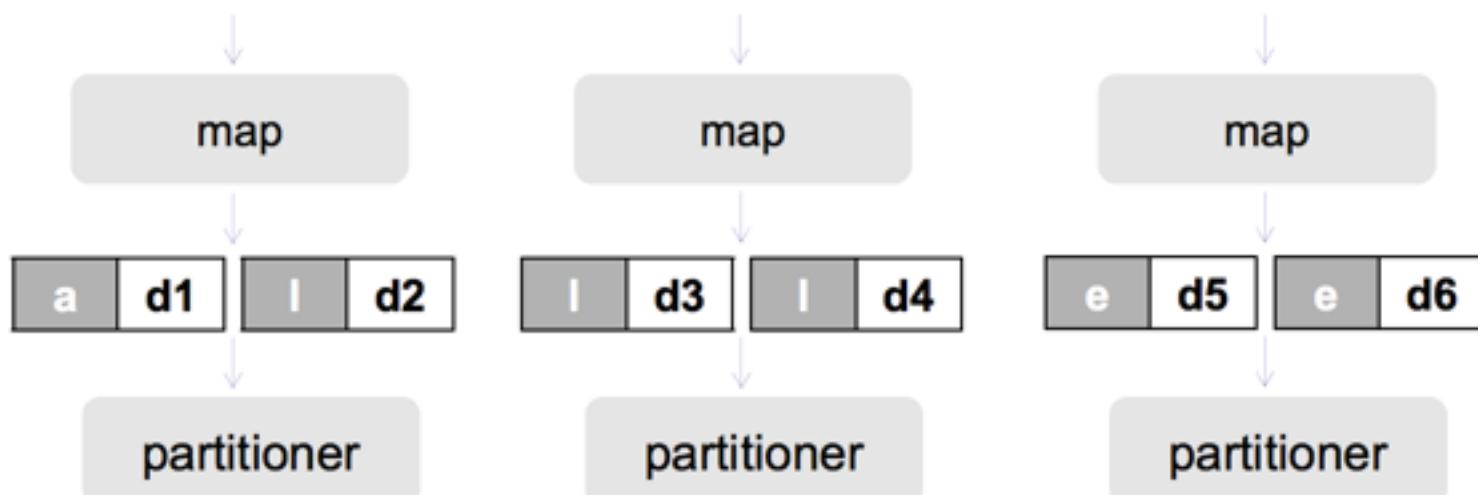
Run with Combiner



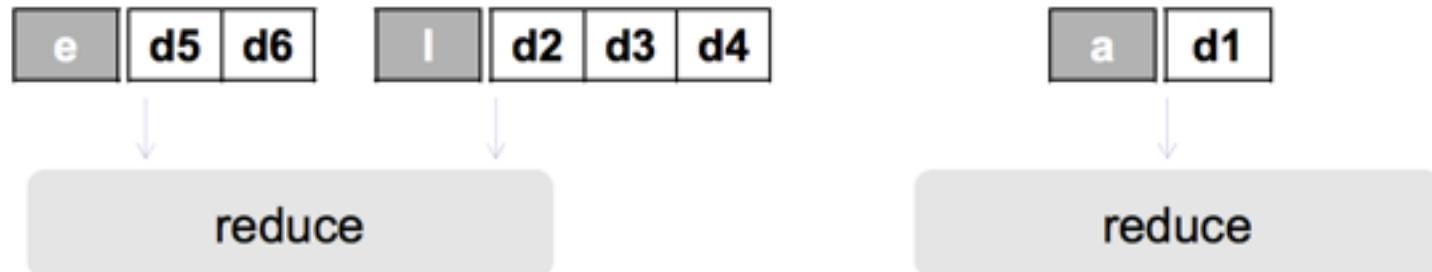
Reducer

- Can configure more than 1 reducer
 - `job.setNumReduceTasks(10);`
 - `mapreduce.job.reduces` property
- Partitioner implementation directs key-value pairs to the proper reducer task
 - A partition is processed by a reduce task
 - # of partitions = # of reduce tasks
 - Default strategy is to hash key to determine partition implemented by `HashPartitioner<K, V>`

Partitioner Data Flow



MapReduce Shuffle and Sort



Source: Jimmy Lin, Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool. 2010

Configure Job

- `job.waitForCompletion(true)`
 - Submits and waits for completion
 - The boolean parameter flag specifies whether output should be written to console
 - If the job completes successfully ‘true’ is returned, otherwise ‘false’ is returned

Our Count Job is configured to..

- Chop up text files into lines
- Send records to mappers as key-value pairs
 - Line number and the actual value
- Mapper class is StartsWithCountMapper
 - Receives key-value of <IntWritable, Text>
 - Outputs key-value of <Text, IntWritable>
- Reducer class is StartsWithCountReducer
 - Receives key-value of <Text, IntWritable>
 - Outputs key-values of <Text, IntWritable> as text
- Combiner class is StartsWithCountReducer

Implement Mapper class

- Class has 4 Java Generics** parameters
 - (1) input key (2) input value (3) output key (4) output value
 - Input and output utilizes hadoop's IO framework
 - org.apache.hadoop.io
- Your job is to implement map() method
 - Input key and value
 - Output key and value
 - Logic is up to you

Implement Mapper class (cont:)

- **map()** method injects Context object, use to:
 - Write output
 - Create your own counters

Implement Reducer

- Analogous to Mapper- generic class with four types
 - (1) input key (2) input value (3) output key (4) output value
 - The output types of map functions must match the input types of reduce function
 - In this case Text and IntWritable

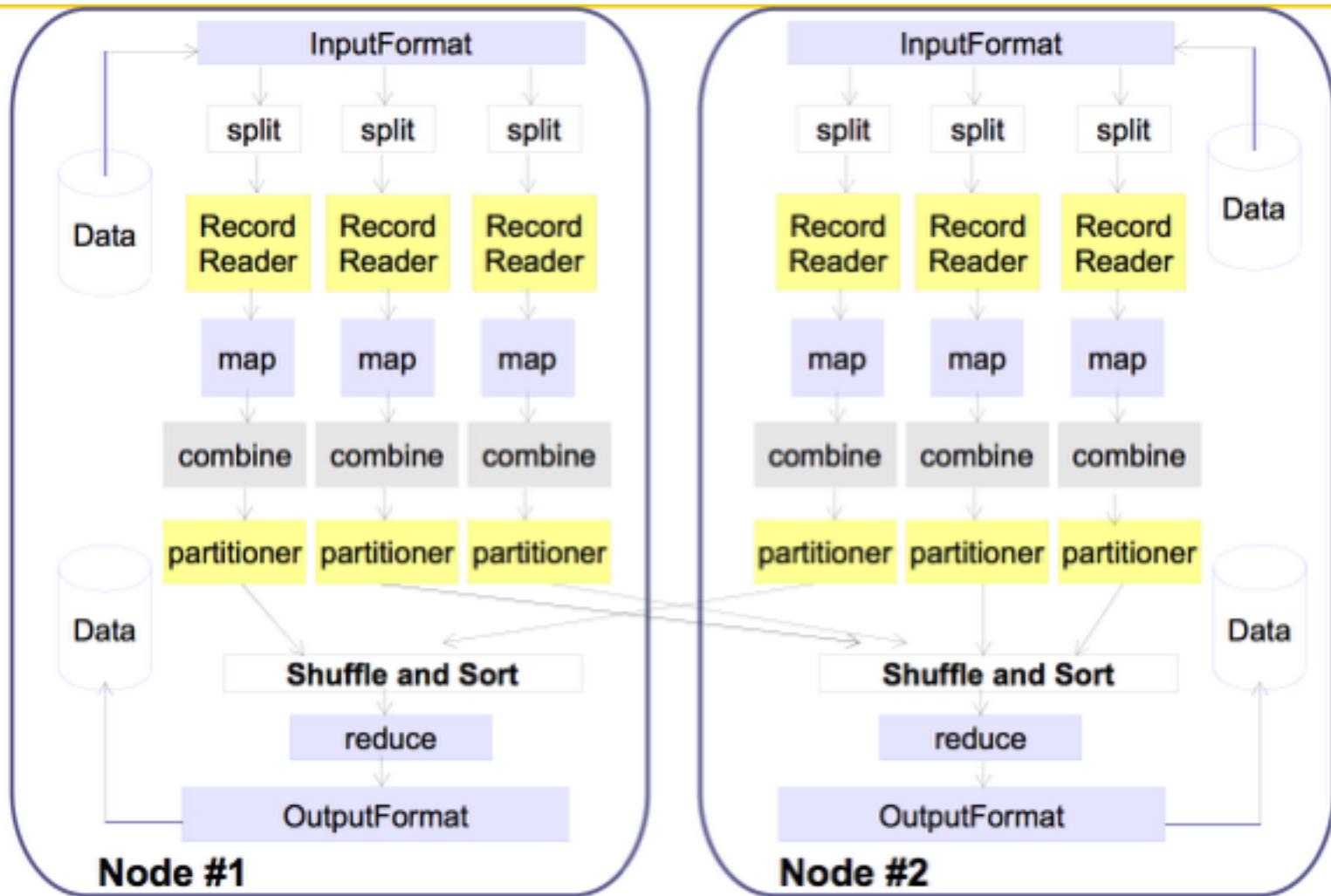
Implement Reducer

- Map/Reduce framework groups key-value pairs produced by mapper by key
 - For each key there is a set of one or more values
 - Input into a reducer is sorted by key
 - Known as Shuffle and Sort
- Reduce function accepts key->Reduce function accepts key-value pairs
 - Also utilizes Context object (similar to Mapper)

Reducer as a Combiner

- Combine data per Mapper task to reduce amount of data transferred to reduce phase
- Reducer can very often serve as a combiner
 - Only works if reducer's output key-value pair types are the same as mapper's output types
- Combiners are not guaranteed to run
 - Optimization only
 - Not for critical logic
- More about combiners later

Component Overview



Source: Yahoo! Inc. "Hadoop Tutorial from Yahoo!", 2012

Output From Your Job

- Provides job id
 - Used to identify, monitor and manage the job
- Shows number of generated splits
- Reports the Progress
- Displays Counters - statistics for the job
 - Sanity check that the numbers match what you expected

\$yarn command

- **yarn script with a class argument command launches a JVM and executes the provided Job**
- You could use straight java but yarn script is more convenient
 - Adds hadoop's libraries to CLASSPATH
 - Adds hadoop's configurations to Configuration object

Chapter 13

MAPREDUCE ON YARN JOB EXECUTION

YARN

- Various applications can run on YARN
 - MapReduce is just one choice
 - Also referred to as MapReduce2.0, NextGen MapReduce

MapReduce on YARN Components

- Client - submits MapReduce Job
- Resource Manager - controls the use of resources across the Hadoop cluster
- Node Manager - runs on each node in the cluster; creates execution container, monitors container's usage
- MapReduce Application Master - Coordinates and manages MapReduce Jobs; negotiates with Resource Manager to schedule tasks; the tasks are started by NodeManager(s)
- HDFS - shares resources and jobs' artifacts between YARN components

Distributed Cache

- A mechanism to distribute files
- Make them available to MapReduce task code
- yarn command provides several options to add distributed files
- Can also use Java API directly
- Supports
 - Simple text files
 - Jars
 - Archives: zip, tar, tgz/tar.gz

Distributed Cache Inner-Workings

- Accepts two types: files and archives
 - Archives are unarchived on the local node
- Items specified to the \$yarn command via -files, -libjars and -archives are copied to HDFS
- Prior to task execution these files are copied locally from HDFS
 - Files now reside on a local disk - local cache
- Files provided to the -libjars are appended to task's CLASSPATH
- Locally cached files become qualified to be deleted after all tasks utilizing cache complete

Distributed Cache Inner-Workings

- Files in the local cache are deleted after a 10GB threshold is reached
 - Allow space for new files
 - Configured via `yarn.nodemanager.localizer.cache.target-size-mb` property
- See `StartsWithCountJob_DistCache.java` in `HadoopSamples` (Mappers)

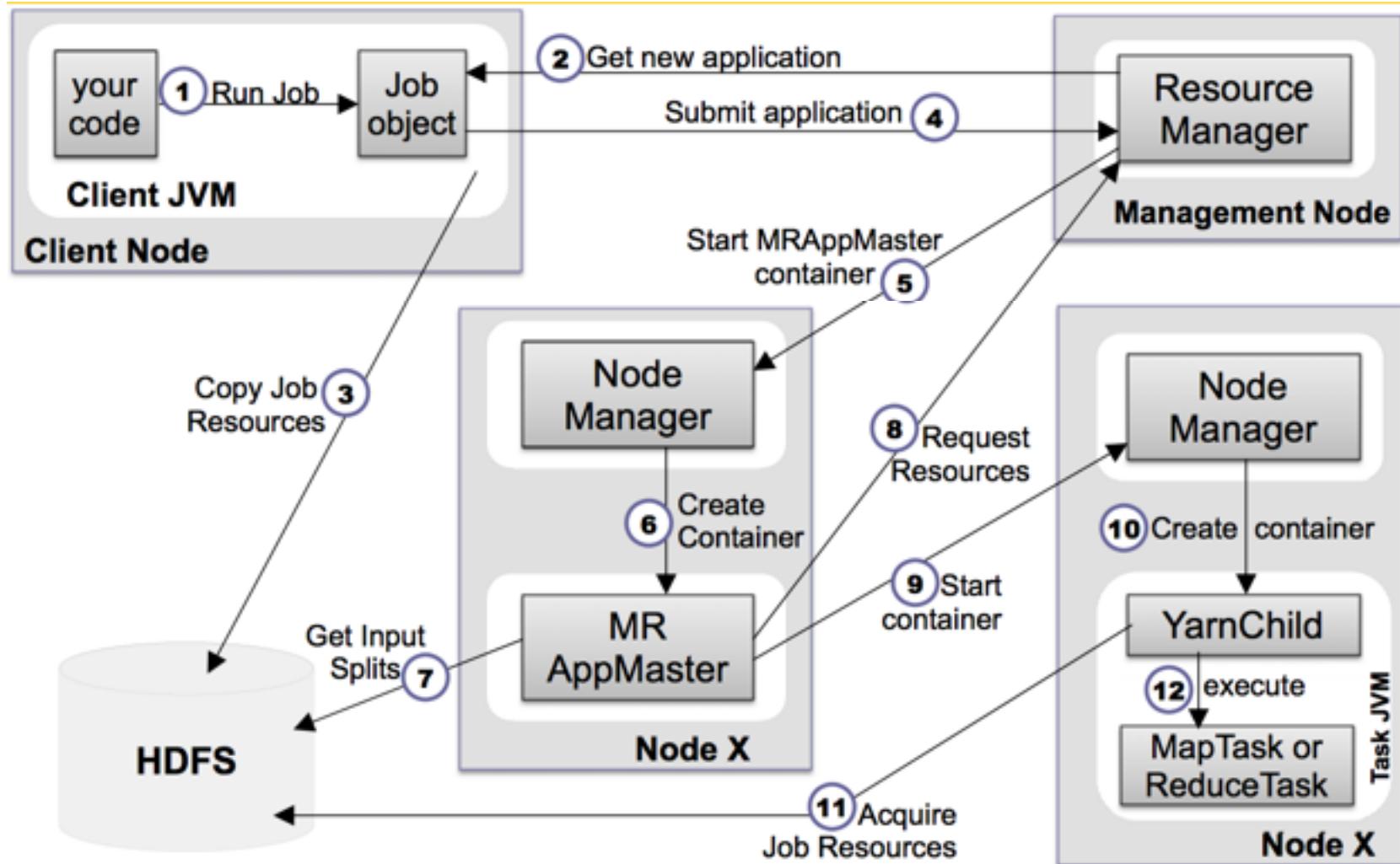
Distributed Cache Inner-Workings

- Local cache is stored under

`${yarn.nodemanager.local-dirs}/usercache/$user/filecache`

- Task code is not aware of the location
 - Symbolic link is created for each file, that's why we were able to reference a file without the path

MapReduce Job Execution on YARN



Source: Tom White, Hadoop: The Definitive Guide, O'Reilly Media, 2012

MapReduce on YARN Job Execution

- Client submits MapReduce job by interacting with Job objects; Client runs in it's own JVM
- Job's code interacts with Resource Manager to acquire application meta-data, such as application id
- Job's code moves all the job related resources to HDFS to make them available for the rest of the job

MapReduce on YARN Job Execution

- Job's code submits the application to Resource Manager
- Resource Manager chooses a Node Manager with available resources and requests a container for MRAppMaster
- Node Manager allocates container for MRAppMaster; MRAppMaster will execute and coordinate MapReduce job
- MRAppMaster grabs required resource from HDFS, such as Input Splits; these resources were copied there in step 3

MapReduce Job Execution on YARN

- MRAppMaster negotiates with Resource Manager for available resources; Resource Manager will select Node Manager that has the most resources
- MRAppMaster tells selected NodeManager to start Map and Reduce tasks
- NodeManager creates YarnChild containers that will coordinate and run tasks

MapReduce Job Execution on YARN

- **YarnChild acquires job resources from HDFS that will be required to execute Map and Reduce tasks**
- **YarnChild executes Map and Reduce tasks**

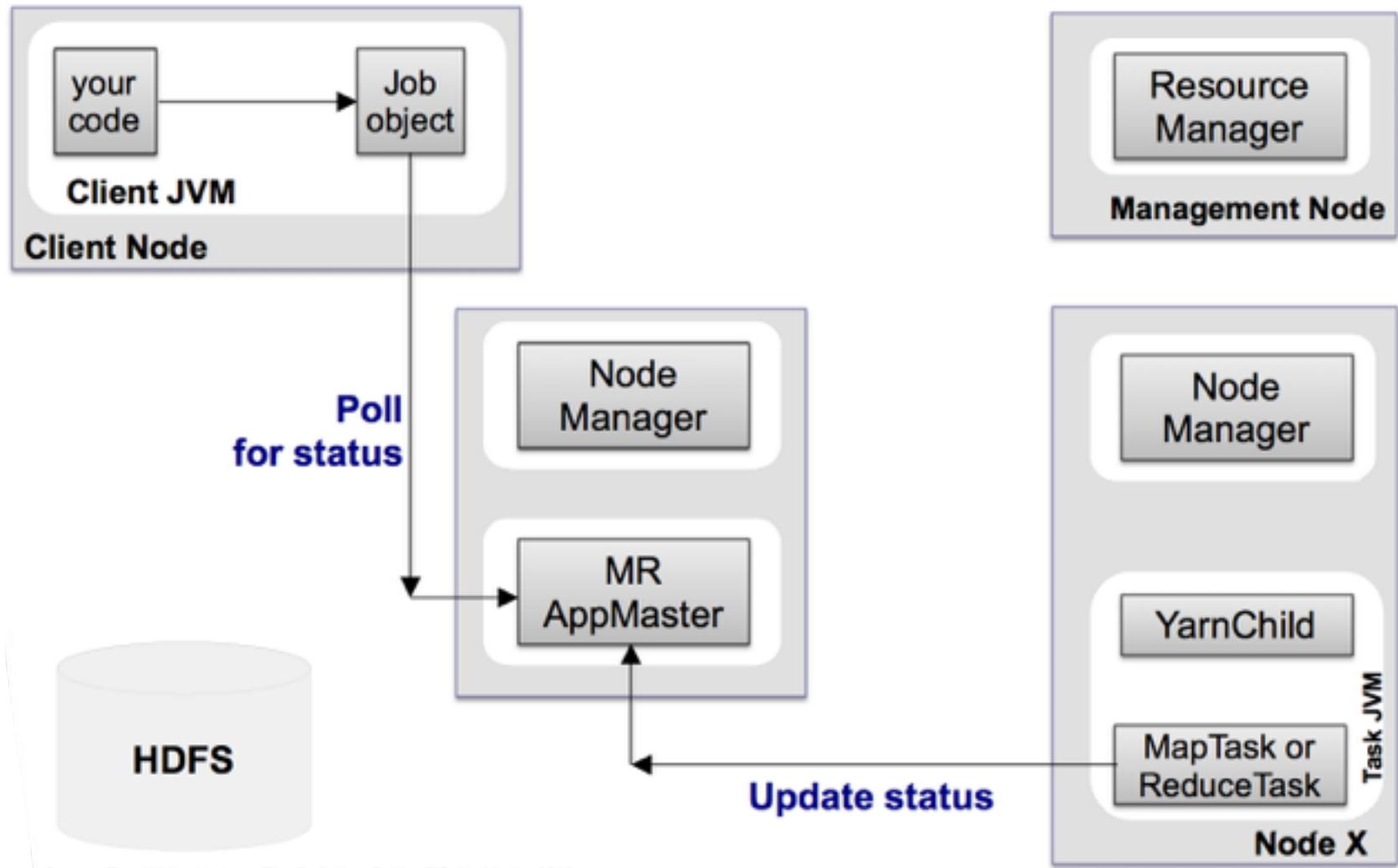
MRAppMaster Initialization Steps

- Decides how to run the tasks
 - In case of a small job, it will run all tasks in MRAppMaster's JVM; this job is called “uberized” or “uber”
 - Execute tasks on Node Manager
- Execute the tasks

MRAppMaster and Uber Job

- Will Uberize if all of these conditions are met:
 - Less than 10 mappers
(mapreduce.job.ubertask.maxmapsproperty)
 - A single Reducer
(mapreduce.job.ubertask.maxreducesproperty)
 - Input size less than 1 HDFS block
(mapreduce.job.ubertask.maxbytes property)
- Execution of Jobs as Uber can be disabled
 - Set mapreduce.job.ubertask.enable property to false

MapReduce Status Updates



Source: Tom White. Hadoop: The Definitive Guide. O'Reilly Media. 2012

Failures

- Failures can occur in
 - Tasks
 - Application Master - MRAppMaster
 - Node Manager
 - Resource Manager

Speculative Execution

- Job is decomposed into small tasks
- Job is as fast as the slowest task
- Given 100s or even 1000s of tasks
 - Few tasks may run very slowly (hardware issues, other activity on that machine, configuration, etc...)
- MapReduce framework strives to resolve slow running tasks by spawning the same task on a different machine
 - Doesn't start speculative tasks immediately

Speculative Execution

- Will spawn a speculative task when
 - All the tasks have been started
 - Task has been running for an extended period of time
 - over a minute
 - Did not make significant progress as compared to the rest of the running tasks
- After task's completion duplicates are killed
- Just an optimization

Speculative Execution

- Can be turned off by setting these properties to false
 - mapred.map.tasks.speculative.execution
 - mapred.reduce.tasks.speculative.execution
- When should I disable Speculative Execution?
 - Task is outputting directly to a shared resource; then starting a duplicate task may cause unexpected results
 - Minimize cluster and bandwidth usage; duplicate tasks use up resources

Task Failures

- Most likely offender and easiest to handle
- Task's exceptions and JVM crashes are propagated to MRAppMaster
- Hanging tasks will be noticed, killed
 - Attempt is marked as failed
 - Control via mapreduce.task.timeout property
- Task is considered to be failed after 4 attempts
 - Set for map tasks via mapreduce.map.maxattempts
 - Set for reduce tasks via mapreduce.reduce.maxattempts

Application Master Failures - MRAppMaster

- MRAppMaster Application can be re-tried
 - By default will not re-try and will fail after a single application failure
- Enable re-try by increasing `yarn.resourcemanager.am.max-retries` property
- Resource Manager receives heartbeats from MRAppMaster and can restart in case of failure(s)

Application Master Failures - MRAppMaster

- Restarted MRAppMaster can recover latest state of the tasks
 - Completed tasks will not need to be re-run
 - To enable set `yarn.app.mapreduce.am.job.recovery.enable` property to true

Node Manager Failure

- Failed Node Manager will not send heartbeat messages to Resource Manager
- Resource Manager will black list a Node Manager that hasn't reported within 10 Minutes
 - Configure via property:
 - `yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms`
 - Usually there is no need to change this setting
- Tasks on a failed Node Manager are recovered and placed on healthy Node Managers

Node Manager Blacklisting by MRAppMaster

- MRAppMaster may blacklist Node Managers if the number of failures is high on that node
- MRAppMaster will attempt to reschedule tasks on a blacklisted Node Manager onto Healthy Nodes
- Blacklisting is per Application/Job therefore doesn't affect other Jobs
- By default blacklisting happens after 3 failures on the same node
 - Adjust default via mapreduce.job.maxtaskfailures.per.tracker

Resource Manager Failures

- The most serious failure = downtime
- Resource Manager was designed to automatically recover
 - Saves state into persistent store by configuring `yarn.resourcemanager.store.class` property

Job Scheduling

- By default Capacity scheduler is used
 - CapacityScheduler
 - Supports basic priority model: VERY_LOW, LOW, NORMAL, HIGH, and VERY_HIGH
 - Two ways to specify priority
 - mapreduce.job.priority property
 - job.setPriority(JobPriority.HIGH)
- Specify scheduler via yarn.resourcemanager.scheduler.class property
 - First In First Out
 - FairScheduler

Job Completion

- After MapReduce application completes (or fails)
 - MRAppMaster and YARN Child JVMs are shut down
 - Management and metrics information is sent from MRAppMaster to the MapReduce History Server

Job Completion

- History server has a similar Web UI to YARN Resource Manager and MRAppMaster
 - By default runs on port 19888
 - <http://localhost:19888/jobhistory>
 - Resource Manager UI auto-magically proxies to the proper location, MRAppMaster if an application is running and History Server after application's completion
 - May get odd behavior (blank pages) if you access an application as it's moving its management from MRAppMaster to History Server

Chapter 14

HADOOP STREAMING

Hadoop Streaming

- Develop MapReduce jobs in practically any language
- Uses Unix Streams as communication mechanism between Hadoop and your code
 - Any language that can read standard input and write standard output will work
- Few good use-cases:
 - Text processing
 - scripting languages do well in text analysis
 - Utilities and/or expertise in languages other than Java

Streaming and MapReduce

- Map input passed over standard input
- Map processes input line-by-line
- Map writes output to standard output
 - Key-value pairs separate by tab ('\t')
- Reduce input passed over standard input
 - Same as mapper output - key-value pairs separated by tab
 - Input is sorted by key
- Reduce writes output to standard output

Implementing Streaming Job

- Choose a language
 - Examples are in Python
- Implement Map function
 - Read from standard input
 - Write to standard out - keys-value pairs separated by tab
- Implement Reduce function
 - Read key-value from standard input
 - Write out to standard output
- Run via Streaming Framework
 - Use \$yarn command

Implement Reduce Code

- Reduce is a little different from Java MapReduce framework
 - Each line is a key-value pair
 - Differs from Java API
 - Values are already grouped by key
 - Iterator is provided for each key
- MapReduce Streaming will still sort by key
- See countMap.py & countReduce.py in HadoopSamples

Run via Streaming Framework

- Before running on a cluster it's very easy to express MapReduce Job via unit pipes

```
$ cat testText.txt | countMap.py | sort | countReduce.py
a      1
h      1
i      4
s      1
• t      5
v      1
```

Run via Streaming Framework

```
yarn jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming*.jar \
  -D mapred.job.name="Count Job via Streaming" \← Name the job
  -files $HADOOP_SAMPLES_SRC/scripts/countMap.py, \
          $HADOOP_SAMPLES_SRC/scripts/countReduce.py \
  -input /training/data/hamlet.txt \
  -output /training/playArea/wordCount/ \
  -mapper countMap.py \
  -combiner countReduce.py \
  -reducer countReduce.py
```

-files options makes scripts available on the cluster for MapReduce

Python vs. Java Map Implementation

Python

```
#!/usr/bin/python
import sys

for line in sys.stdin:
    for token in line.strip().split(" "):
        if token: print token[0] + '\t1'
```

Java

```
package mr.wordcount;

import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;

public class StartsWithCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable countOne = new IntWritable(1);
    private final Text reusableText = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        StringTokenizer tokenizer = new StringTokenizer(value.toString());
        while (tokenizer.hasMoreTokens()) {
            reusableText.set(tokenizer.nextToken().substring(0, 1));
            context.write(reusableText, countOne);
        }
    }
}
```

Python vs. Java Reduce Implementation

Python

```
#!/usr/bin/python

import sys

(lastKey, sum)=(None, 0)
for line in sys.stdin:
    (key, value) = line.strip().split("\t")

    if lastKey and lastKey != key:
        print lastKey + '\t' + str(sum)
        (lastKey, sum) = (key, int(value))
    else:
        (lastKey, sum) = (key, sum + int(value))

if lastKey:
    print lastKey + '\t' + str(sum)
```

Java

```
package mr.wordcount;

import java.io.IOException;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;

public class StartsWithCountReducer extends
    Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    protected void reduce(Text token, Iterable<IntWritable> counts,
        Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable count : counts) {
            sum+== count.get();
        }
        context.write(token, new IntWritable(sum));
    }
}
```

Chapter 15

MAP REDUCE WORKFLOWS

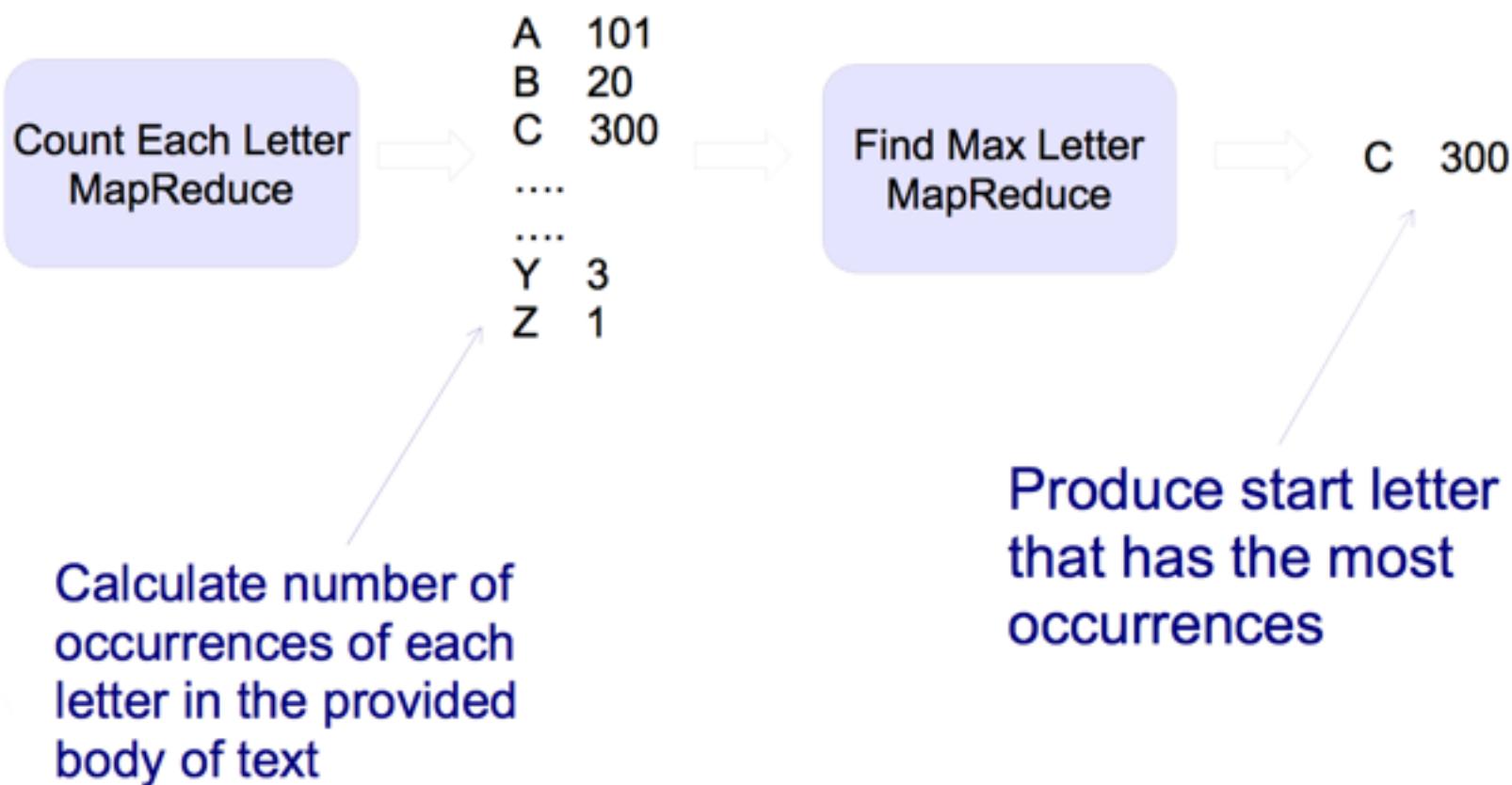
MapReduce Workflows

- We've looked at single MapReduce job
- Complex processing requires multiple steps
 - Usually manifest in multiple MapReduce jobs
- Higher-level MapReduce abstractions
 - Pig, Hive, Cascading, Cascalog, Crunch
 - Focus on business logic rather than MapReduce translation

Decomposing Problems into MapReduce Jobs

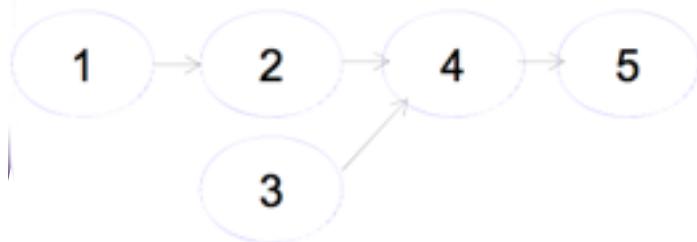
- Small map-reduce jobs are usually better
 - Easier to implement, test and maintain
 - Easier to scale and re-use
- Problem:
 - Find a letter that occurs the most in the provided body of text

Decomposing the Problem

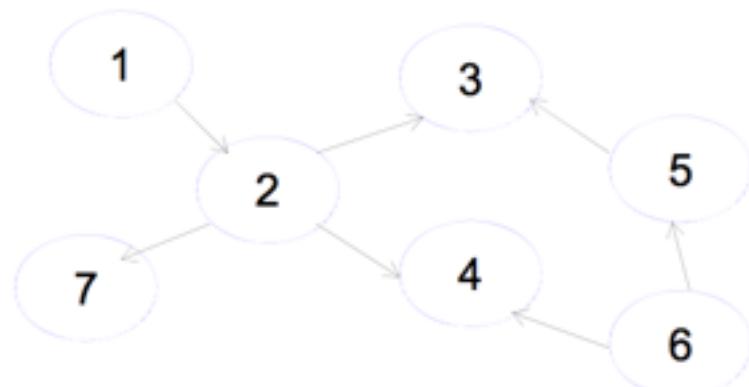


MapReduce Workflows

- Your choices can depend on complexity of workflows
 - Linear chain or simple set of dependent Jobs vs. Directed Acyclic Graph (DAG)
 - http://en.wikipedia.org/wiki/Directed_acyclic_graph



vs.



Simple Dependency
or
Linear chain

Directed Acyclic Graph (DAG)

MapReduce Workflows

- JobControl class
 - Create simple workflows
 - Represents a graph of Jobs to run
 - Specify dependencies in code
- Oozie
 - An engine to build complex DAG workflows
 - Runs in its own daemon
 - Describe workflows in XML
 - Coordinator engine that schedules workflows based on time and incoming data
 - Ability to re-run failed portions of the workflow

Workflow with JobControl

- Create JobControl
 - Implements `java.lang.Runnable`, will need to execute within a Thread
- For each Job in the workflow Construct ControlledJob
 - Wrapper for Job instance
 - Constructor takes in dependent jobs
- Add each ControlledJob to JobControl

Workflow with JobControl

- Execute JobControl in a Thread
 - Recall JobControl implements Runnable
- Wait for JobControl to complete and report results
 - Clean-up in case of a failure
- See `MostSeenStartLetterJobControl.java` in `HadoopSamples`

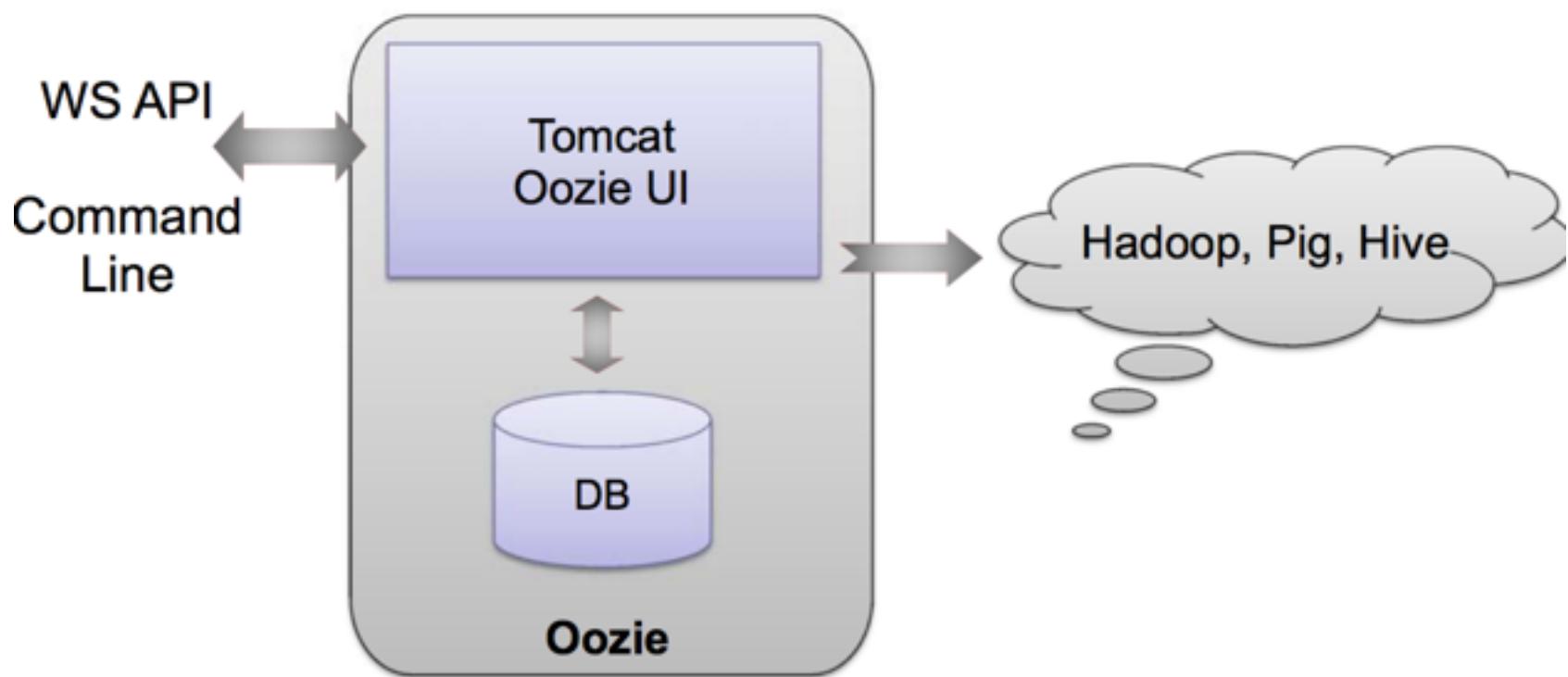
Chapter 16

OOZIE

Oozie

- Workflow scheduler for Hadoop
 - Java MapReduce Jobs
 - Streaming Jobs
 - Pig
- Top level Apache project
 - Comes packaged in major Hadoop Distributions
- Provides workflow management and coordination of those workflows
- Manages Directed Acyclic Graph (DAG) of actions

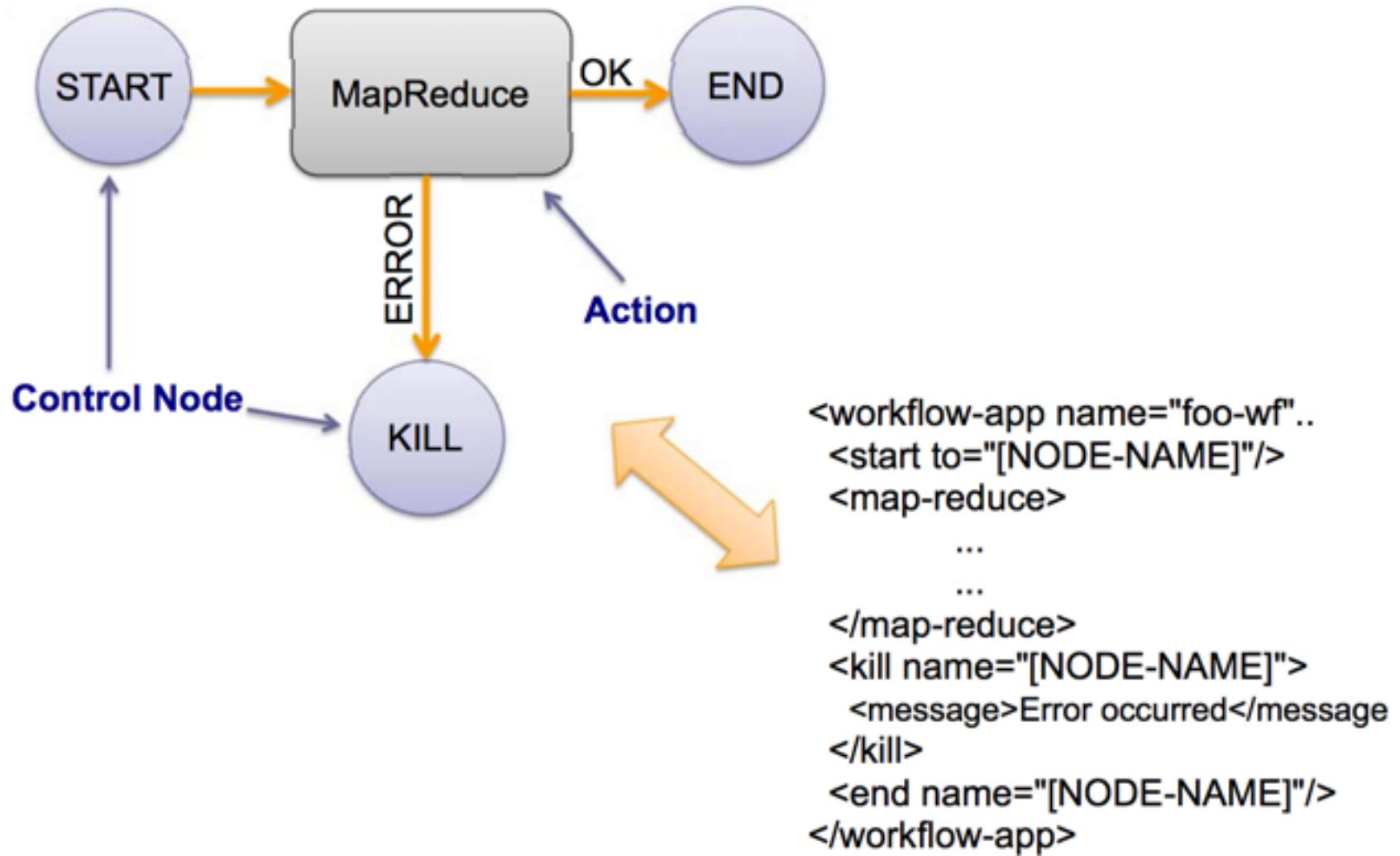
Oozie



Oozie

- Runs HTTP service
 - Clients interact with the service by submitting workflows
 - Workflows are executed immediately or later
- Workflows are defined via XML
 - Instead of writing Java code that implements Tool interface and extending Configured class

Action and Control Nodes



Action and Control Nodes

- **Control Flow**
 - **start, end, kill**
 - **decision**
 - **fork, join**
- **Actions**
 - **map-reduce**
 - **java**
 - **pig**
 - **hdfs**

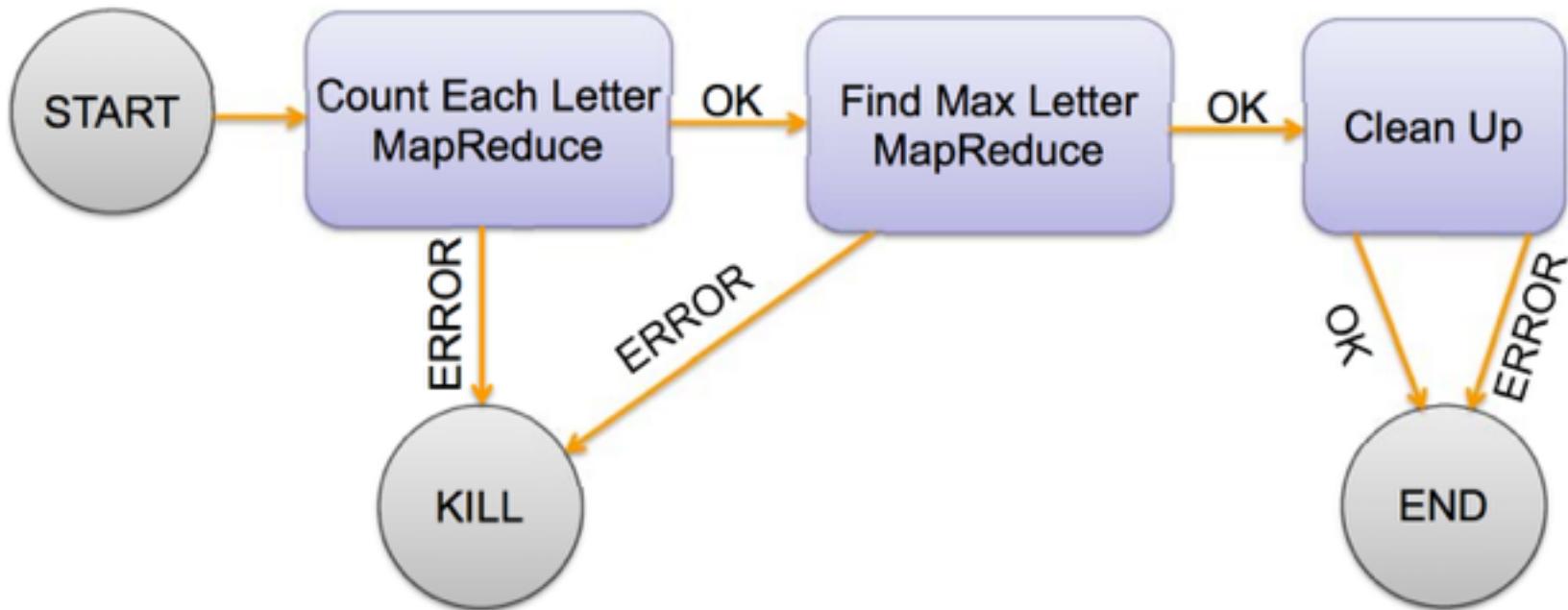
Oozie Coordination Engine

- Oozie Coordination Engine can trigger workflows by
 - Time (Periodically)
 - Data Availability (Data appears in a directory)
- Defined in XML
- Uses Process Definition Language

Oozie Workflows

- Workflows consist of
 - Action nodes
 - MapReduce, Pig, Hive
 - Streaming, Java, etc...
 - Control flow nodes
 - Logic decisions between action nodes
 - Execute actions based on conditions or in parallel
- Workflows begin with START node
- Workflows succeed with END node
- Workflows fail with KILL node
- Several actions support JSP Expression Language (EL)

Most Occurrences Workflows



Count Each Letter
MapReduce

- Action Node



- Control Flow Node

END

- Control Node

This source is in *HadoopSamples*
project under
`/src/main/resources/mr/workflows`

Most Occurrences Workflows

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="most-seen-letter">
  <start to="count-each-letter"/>
  <action name="count-each-letter">
    <map-reduce>
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <prepare>
        <delete path="${nameNode}${outputDir}"/>
        <delete path="${nameNode}${intermediateDir}"/>
      </prepare>
      <configuration>
        ...
        <property>
          <name>mapreduce.job.map.class</name>
          <value>mr.wordcount.StartsWithCountMapper</value>
        </property>
        ...
      </configuration>
    </map-reduce>
    <ok to="find-max-letter"/>
    <error to="fail"/>
  </action>
  ...

```

MapReduce have optional Prepare section

START Action Node to count-each-letter MapReduce action

Pass property that will be set on MapReduce job's Configuration object

In case of success, go to the next job; in case of failure go to fail node

First map-reduce Action

```
<map-reduce>
  <job-tracker>${jobTracker}</job-tracker>
  <name-node>${nameNode}</name-node>
  <prepare>
    <delete path="${nameNode}${outputDir}"/>
    <delete path="${nameNode}${intermediateDir}"/>
  </prepare>
  <configuration>
    <property>
      <name>mapred.mapper.new-api</name>
      <value>true</value>
    </property>
    <property>
      <name>mapred.reducer.new-api</name>
      <value>true</value>
    </property>
    <property>
      <name>mapred.job.queue.name</name>
      <value>${queueName}</value>
    </property>
  ...
  ...

```

Administrative items to indicate where namenode and resource manager is

Optional prepare section; allows to execute command prior running the job

By default “old api” is used; specify to use new api

Specify which queue to submit this job to Resource Manager

First map-reduce Action

```
...  
<property>  
  <name>mapreduce.job.map.class</name>  
  <value>mr.wordcount.StartsWithCountMapper</value>  
</property>  
<property>  
  <name>mapreduce.job.combine.class</name>  
  <value>mr.wordcount.StartsWithCountReducer</value>  
</property>  
<property>  
  <name>mapreduce.job.reduce.class</name>  
  <value>mr.wordcount.StartsWithCountReducer</value>  
</property>  
<property>  
  <name>mapreduce.job.inputformat.class</name>  
  <value>org.apache.hadoop.mapreduce.lib.input.TextInputFormat</value>  
</property>  
<property>  
  <name>mapreduce.job.outputformat.class</name>  
  <value>org.apache.hadoop.mapreduce.lib.output.TextOutputFormat</value>  
</property>  
...  
  
This action will produce a file of tab separated key-value pairs as specified by TextOutputFormat
```

Specify Mapper, Reducer, Input and Output formats; this is instead of Tool implementation

Most Occurrences Workflows

```
<action name="find-max-letter"> ← Second MapReduce job
  <map-reduce>
    <job-tracker>${jobTracker}</job-tracker>
    <name-node>${nameNode}</name-node> ← Namenode and Yarn
    <configuration>                         Resource Manager
      ...                                     Location
      ...                                     Token substituted from
      <property>                           application properties file
        <name>mapreduce.job.map.class</name>
        <value>mr.workflows.MostSeenStartLetterMapper</value>
      </property>
      <property>
        <name>mapreduce.job.combine.class</name>
        <value>mr.workflows.MostSeendStartLetterReducer</value>
      ...
      ...
    </configuration> ← Control Flow Node
  </map-reduce>
  <ok to="clean-up"/>
  <error to="fail"/>
</action>
```

Package and Run Your Workflow

- Create application directory structure with workflow definitions and resources
 - Workflow.xml, jars, etc..
- Copy application directory to HDFS
- Create application configuration file
 - specify location of the application directory on HDFS
 - specify location of the namenode and resource manager
- Submit workflow to Oozie
 - Utilize oozie command line
- Monitor running workflow(s)

Oozie Application Directory

- Must comply to directory structure spec

```
mostSeenLetter-oozieWorkflow  
|--lib/  
|   |--HadoopSamples.jar  
|--workflow.xml
```

Application Workflow Root

Libraries should be placed under lib directory

Workflow.xml defines workflow

Create Application Configuration File

- **job.properties** - Needs to exist locally, required for submission

```
nameNode=hdfs://localhost:8020  
jobTracker=localhost:8021  
queueName=default
```

Properties for required locations such as namenode and resource manager

```
inputFile=/training/data/hamlet.txt  
intermediateDir=/training/playArea/mostSeenLetter-oozieWorkflow-tmp  
outputDir=/training/playArea/oozieWorkflow
```

Properties needed for the MapReduce actions in the workflow

```
oozie.wf.application.path=${nameNode}/user/${user.name}/mostSeenLetter-oozieWorkflow
```

Most importantly HDFS location of the application must be specified

Submit Workflow to Oozie

- Use oozie command line tool
 - For usage: \$oozie help

```
$ oozie job -config job.properties -run  
job: 0000001-120711224224630-oozie-hado-W
```

Application configuration file

Application ID; use this ID to get status

Monitor Running Workflow(s)

- Two options
 - Command line (\$oozie)
 - Web Interface (<http://localhost:11000/oozie>)

5: Monitor Running Workflow(s)

- Web Interface

<http://localhost:11000/oozie>

The screenshot shows the Oozie Web Console interface in Mozilla Firefox. The title bar reads "Oozie Web Console - Mozilla Firefox". The address bar shows the URL "http://localhost:11000/oozie/". The main content area displays a table of workflow jobs. The columns are labeled: Job Id, Name, Status, Run, User, Group, Created, and Started. Arrows point from the text labels "Application ID" and "Status of the Workflow" to the "Job Id" column and the "Status" column respectively.

Job Id	Name	Status	Run	User	Group	Created	Started
1 0000001-120711224224630-oozie-hado-W	most-seen-letter	SUCCEEDED	0	hadoop		Fri, 13 Jul 2012 03:08:16 GMT	Fri, 13 Ju
2 0000000-120711224224630-oozie-hado-W	most-seen-letter	SUCCEEDED	0	hadoop		Fri, 13 Jul 2012 02:30:22 GMT	Fri, 13 Ju
3 0000026-120623200723222-oozie-hado-W	most-seen-letter	SUCCEEDED	0	hadoop		Sun, 24 Jun 2012 04:31:58 GMT	Sun, 24 J
4 0000025-120623200723222-oozie-hado-W	most-seen-letter	SUCCEEDED	0	hadoop		Sun, 24 Jun 2012 04:23:32 GMT	Sun, 24 J
5 0000024-120623200723222-oozie-hado-W	most-seen-letter	KILLED	0	hadoop		Sun, 24 Jun 2012 04:20:32 GMT	Sun, 24 J
6 0000023-120623200723222-oozie-hado-W	most-seen-letter	KILLED	0	hadoop		Sun, 24 Jun 2012 04:17:42 GMT	Sun, 24 J
7 0000022-120623200723222-oozie-hado-W	most-seen-letter	SUCCEEDED	0	hadoop		Sun, 24 Jun 2012 04:12:05 GMT	Sun, 24 J

Chapter 17

APACHE PIG

Pig

“is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. “

- **Top Level Apache Project**
 - <http://pig.apache.org>
- **Pig is an abstraction on top of Hadoop**
 - Provides high level programming language designed for data processing
 - Converted into MapReduce and executed on Hadoop Clusters
- **Pig is widely accepted and used**
 - Yahoo!, Twitter, Netflix, etc...

Pig and MapReduce

- MapReduce requires programmers
 - Must think in terms of map and reduce functions
 - More than likely will require Java programmers
- Pig provides high-level language that can be used by
 - Analysts
 - Data Scientists
 - Statisticians
- Originally implemented at Yahoo! to allow analysts to access data

Pig's Features

- Join Datasets
- Sort Datasets
- Filter
- Data Types
- Group By
- User Defined Functions
- Etc..

Pig's Use Cases

- Extract Transform Load (ETL)
 - Ex: Processing large amounts of log data
 - clean bad entries, join with other data-sets
- Research of “raw” information
 - Ex. User Audit Logs
 - Schema maybe unknown or inconsistent
 - Data Scientists and Analysts may like Pig’s data transformation paradigm

Pig Components

- **Pig Latin**
 - Command based language
- **Execution Environment**
 - Currently there is support for Local and Hadoop modes
- **Pig compiler converts Pig Latin to MapReduce**
 - Compiler strives to optimize execution
 - You automatically get optimization improvements with Pig updates

Execution Modes

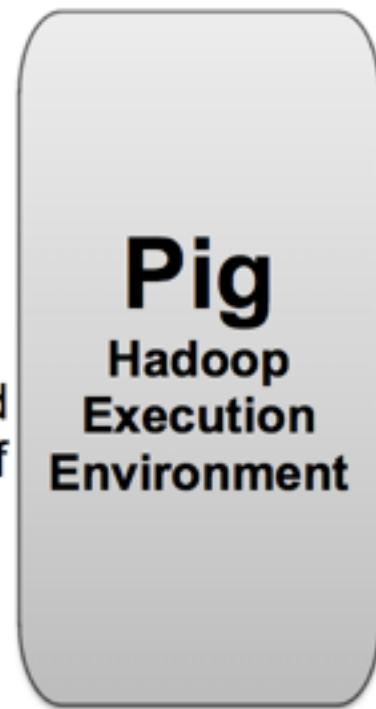
- Local
 - Executes in a single JVM
 - Works exclusively with local file system
 - Great for development, experimentation and prototyping
- Hadoop Mode
 - Also known as MapReduce mode
 - Pig renders Pig Latin into MapReduce jobs and executes them on the cluster
 - Can execute against semi-distributed or fully-distributed hadoop installation

Hadoop Mode

```
-- 1: Load text into a bag, where a row is a line of  
text:  
lines = LOAD '/training/playArea/hamlet.txt' AS  
(line:chararray);  
-- 2: Tokenize the provided text:  
tokens = FOREACH lines GENERATE  
flatten(TOKENIZE(line)) AS token:chararray;
```

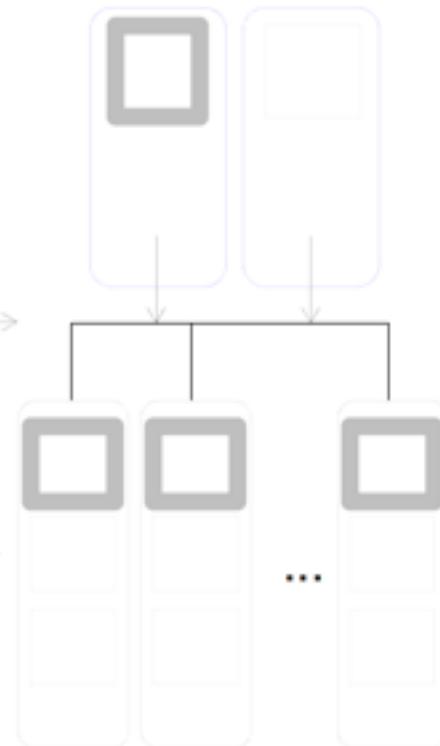
PigLatin.pig

Parse Pig script and
compile into a set of
MapReduce jobs



Execute on
Hadoop Cluster

Monitor/Report



**Hadoop
Cluster**

Running Pig

- **Script**
 - Execute commands in a file
 - `$pig scriptFile.pig`
- **Grunt**
 - Interactive Shell for executing Pig Commands
 - Started when script file is NOT provided
 - Can execute scripts from Grunt via run or exec commands

Running Pig

- **Embedded**
 - Execute Pig commands using `PigServer` class
 - Just like JDBC to execute SQL
 - Can have programmatic access to Grunt via `PigRunner` class

Pig Latin Concepts

- Building blocks
 - Field - piece of data
 - Tuple - ordered set of fields, represented with “(“ and “)”
 - (10.4, 5, word, 4, field1)
 - Bag - collection of tuples, represented with “{“ and “}”
 - { (10.4, 5, word, 4, field1), (this, 1, blah) }

Pig Latin Concepts (cont:)

- Similar to Relational Database
 - Bag is a table in the database
 - Tuple is a row in a table
 - Bags do not require that all tuples contain the same number
 - Unlike relational table

Simple Pig Latin Example

```
$ pig
grunt> cat /training/playArea/pig/a.txt      Start Grunt with default
a      1                                         MapReduce mode
d      4                                         Grunt supports file
c      9                                         system commands
k      6                                         Load contents of text files
                                              into a Bag named records
grunt> records = LOAD '/training/playArea/pig/a.txt' as
(letter:chararray, count:int);
grunt> dump records;                         Display records bag to
...                                           the screen
                                             
org.apache.pig.backend.hadoop.executionengine.mapReduceLayer
.MapReduceLauncher - 50% complete
2012-07-14 17:36:22,040 [main] INFO
org.apache.pig.backend.hadoop.executionengine.mapReduceLayer
.MapReduceLauncher - 100% complete
...
(a,1)
(d,4)                                         Results of the bag named records
(c,9)                                         are printed to the screen
(k,6)
```

DUMP and STORE statements

- No action is taken until DUMP or STORE commands are encountered
 - Pig will parse, validate and analyze statements but not execute them
- DUMP - displays the results to the screen
- STORE - saves results (typically to a file)

Nothing is executed;
Pig will optimize this entire chunk of script

```
records = LOAD '/training/playArea/pig/a.txt' as  
(letter:chararray, count:int);  
...  
...  
...  
...  
...  
...  
DUMP final_bag;
```

The fun begins here

Large Data

- Hadoop data is usually quite large and it doesn't make sense to print it to the screen
- The common pattern is to persist results to Hadoop (HDFS, HBase)
 - This is done with STORE command
- For information and debugging purposes you can print a small sub-set to the screen

```
grunt> records = LOAD '/training/playArea/pig/excite-small.log'  
AS (userId:chararray, timestamp:long, query:chararray);  
grunt> toPrint = LIMIT records 5;  
grunt> DUMP toPrint;
```

Only 5 records will be displayed

LOAD Command

```
LOAD 'data' [USING function] [AS schema];
```

- **data** - name of the directory or file
 - Must be in single quotes
- **USING** - specifies the load function to use
 - By default uses PigStorage which parses each line into fields using a delimiter
 - Default delimiter is tab ('\t')
- **AS** - assign a schema to incoming data
 - Assigns names to fields
 - Declares types to fields

LOAD Command Example

```
records =  
    LOAD '/training/playArea/pig/excite-small.log'  
    USING PigStorage()  
    AS (userId:chararray, timestamp:long, query:chararray);
```

Data

Schema

User selected Load Function, there
are a lot of choices or you can
implement your own

Schema Data Types

Type	Description	Example
Simple		
int	Signed 32-bit integer	10
long	Signed 64-bit integer	10L or 10l
float	32-bit floating point	10.5F or 10.5f
double	64-bit floating point	10.5 or 10.5e2 or 10.5E2
Arrays		
chararray	Character array (string) in Unicode UTF-8	hello world
bytearray	Byte array (blob)	
Complex Data Types		
tuple	An ordered set of fields	(19,2)
bag	An collection of tuples	{(19,2), (18,1)}
map	An collection of tuples	[open#apache]

Pig Latin - Diagnostic Tools

- Display the structure of the Bag
 - `grunt> DESCRIBE <bag_name>;`
- Display Execution Plan
 - Produces Various reports
 - Logical Plan
 - MapReduce Plan
 - `grunt> EXPLAIN <bag_name>;`
- Illustrate how Pig engine transforms the data
 - `grunt> ILLUSTRATE <bag_name>;`

Pig Latin - Grouping

```
grunt> chars = LOAD '/training/playArea/pig/b.txt' AS  
(c:chararray);  
grunt> describe chars;  
chars: {c: chararray}  
grunt> dump chars;  
(a)  
(k)      Creates a new bag with element named  
...       group and element named chars  
...  
(k)  
(c)  
(k)
```

```
grunt> charGroup = GROUP chars by c;  
grunt> describe charGroup;  
charGroup: {group: chararray,chars: { (c: chararray) }}  
grunt> dump charGroup;  
(a,{(a),(a),(a)})  
(c,{(c),(c)})  
(i,{(i),(i),(i)})  
(k,{(k),(k),(k),(k)})  
(l,{(l),(l)})
```

The chars bag is grouped by "c"; therefore 'group' element will contain unique values

'chars' element is a bag itself and contains all tuples from 'chars' bag that match the value form 'c'

ILLUSTRATE Command

```
grunt> chars = LOAD '/training/playArea/pig/b.txt' AS (c:chararray);  
grunt> charGroup = GROUP chars by c;  
grunt> ILLUSTRATE charGroup;
```

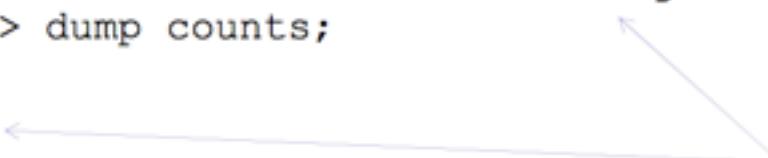
chars	c:chararray
	c
	c

charGroup	group:chararray	chars:bag{:tuple(c:chararray)}
	c	{(c), (c)}

Pig Latin - FOREACH

- FOREACH <bag> GENERATE <data>
 - Iterate over each element in the bag and produce a result
 - Ex: **grunt> result = FOREACH bag GENERATE f1;**

```
grunt> records = LOAD 'data/a.txt' AS (c:chararray, i:int);
grunt> dump records;
(a,1)
(d,4)
(c,9)
(k,6)
grunt> counts = foreach records generate i;
grunt> dump counts;
(1)
(4)
(9)
(6)
```



For each row emit 'i' field

FOREACH with Functions

- FOREACH B GENERATE group, FUNCTION(A);
 - Pig comes with many functions including COUNT, FLATTEN, CONCAT, etc...
 - Can implement a custom function

```
grunt> chars = LOAD 'data/b.txt' AS (c:chararray);
grunt> charGroup = GROUP chars by c;
grunt> dump charGroup;
(a,{{(a),(a),(a)})}
(c,{{(c),(c)})}
(i,{{(i),(i),(i)})}
(k,{{(k),(k),(k),(k)}})
(l,{{(l),(l)}})
grunt> describe charGroup;
charGroup: {group: chararray,chars: {{c: chararray}}}
grunt> counts = FOREACH charGroup GENERATE group, COUNT(chars);
grunt> dump counts;
(a,3)
(c,2)
(i,3)
(k,4)
(l,2)
```

For each row in 'charGroup' bag, emit group field and count the number of items in 'chars' bag

TOKENIZE Function

- Splits a string into tokens and outputs as a bag of tokens
 - Separators are: space, double quote("), comma(,), parenthesis(()), star(*)

```
grunt> linesOfText = LOAD 'data/c.txt' AS (line:chararray);
grunt> dump linesOfText;
(this is a line of text)
(yet another line of text)
(third line of words)
```

Split each row line by space
and return a bag of tokens

```
grunt> tokenBag = FOREACH linesOfText GENERATE TOKENIZE(line);
```

```
grunt> dump tokenBag;
({(this),(is),(a),(line),(of),(text)}) 
({(yet),(another),(line),(of),(text)}) 
({(third),(line),(of),(words)})
```

Each row is a bag of
words produced by
TOKENIZE function

```
grunt> describe tokenBag;
tokenBag: {bag_of_tokenTuples: {tuple_of_tokens: (token: chararray)}}
```

FLATTEN Operator

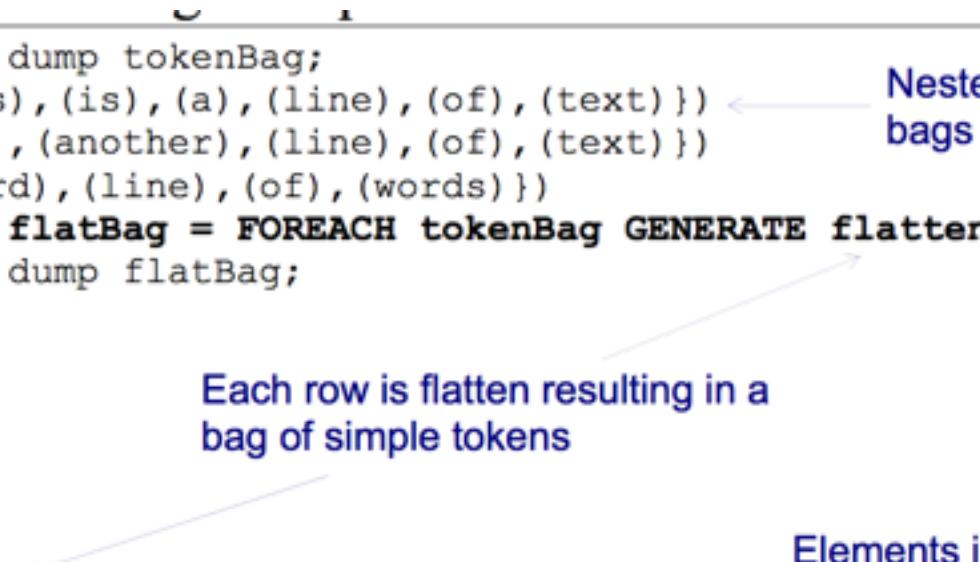
- Flattens nested bags and data types
- FLATTEN is not a function, it's an operator
 - Re-arranges output

```
grunt> dump tokenBag;
((this), (is), (a), (line), (of), (text))
((yet), (another), (line), (of), (text))
((third), (line), (of), (words))
grunt> flatBag = FOREACH tokenBag GENERATE flatten($0);
grunt> dump flatBag;
(this)
(is)
(a)
...
...
(text)
(third)
(line)
(of)
(words)
```

Nested structure: bag of bags of tuples

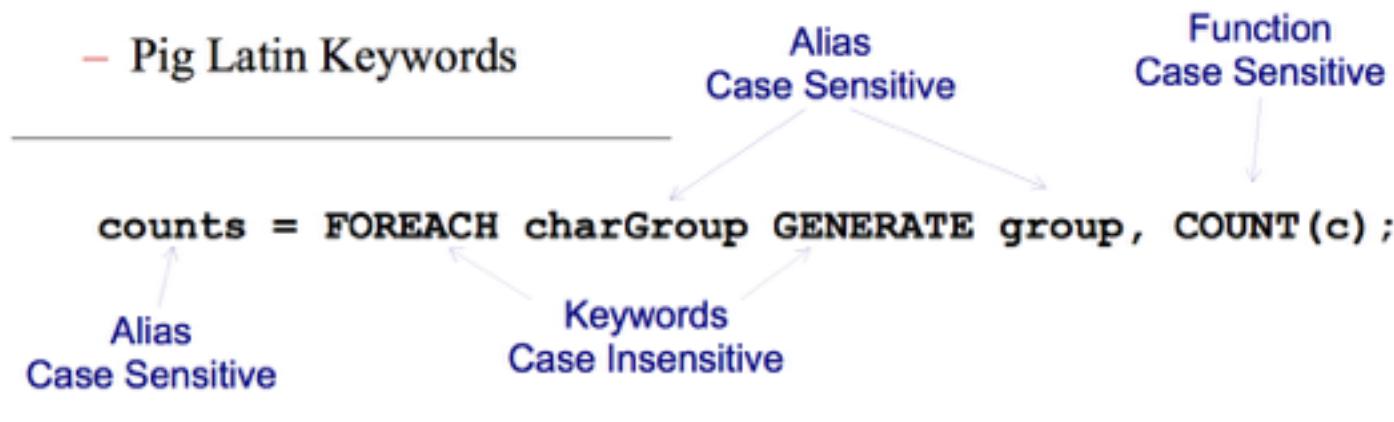
Each row is flatten resulting in a bag of simple tokens

Elements in a bag can be referenced by index



Conventions and Case Sensitivity

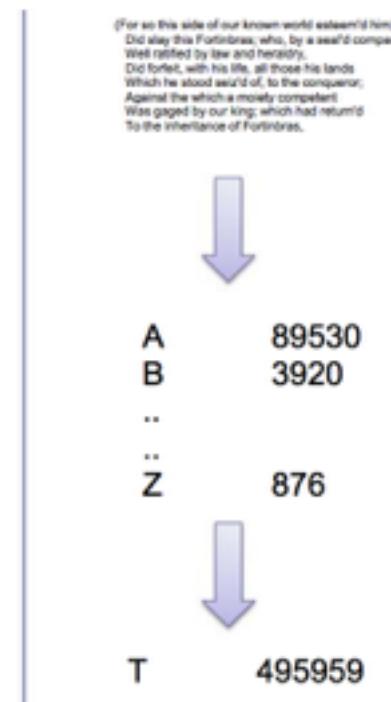
- Case Sensitive
 - Alias names
 - Pig Latin Functions
- Case Insensitive
 - Pig Latin Keywords



- **General conventions**
 - Upper case is a system keyword
 - Lowercase is something that you provide

Problem: Locate Most Occurred Start Letter

- Calculate number of occurrences of each letter in the provided body of text
- Traverse each letter comparing occurrence count
- Produce start letter that has the most occurrences



‘Most Occurred Start Letter’ Pig Way

- Load text into a bag (named ‘lines’)
- Tokenize the text in the ‘lines’ bag
- Retain first letter of each token
- Group by letter
- Count the number of occurrences in each group
- Descending order the group by the count
- Grab the first element => Most occurring letter
- Persist result on a file system

Load Text Into a Bag

```
grunt> lines = LOAD '/training/data/hamlet.txt'  
AS (line:chararray);
```

Load text file into a bag, stick entire line into
element 'line' of type 'chararray'

INSPECT lines bag:

```
grunt> describe lines;  
lines: {line: chararray}  
grunt> toDisplay = LIMIT lines 5;  
grunt> dump toDisplay;  
(This Etext file is presented by Project Gutenberg, in)  
(This etext is a typo-corrected version of Shakespeare's Hamlet,)  
(cooperation with World Library, Inc., from their Library of the)  
(*This Etext has certain copyright implications you should read!*)  
(Future and Shakespeare CDROMS. Project Gutenberg often releases
```

Each row is a line of text

Tokenize the Text in the ‘Lines’ Bag

```
grunt> tokens = FOREACH lines GENERATE  
flatten(TOKENIZE(line)) AS token:chararray;
```

For each line of text (1) tokenize that line
(2) flatten the structure to produce 1 word per row

INSPECT tokens bag:

```
grunt> describe tokens  
tokens: {token: chararray}  
grunt> toDisplay = LIMIT tokens 5;  
grunt> dump toDisplay;  
(a)  
(is)  
(of)  
(This)  
(etext)
```

Each row is now a token

Retain First Letter of Each Token

```
grunt> letters = FOREACH tokens GENERATE  
SUBSTRING(token,0,1) AS letter:chararray;
```

For each token grab the first letter; utilize
SUBSTRING function

INSPECT letters bag:

```
grunt> describe letters;  
letters: {letter: chararray}  
grunt> toDisplay = LIMIT letters 5;  
grunt> dump toDisplay;  
(a)  
(i)  
(T)  
(e)  
(t)
```

What we have no is 1
character per row

Group by Letter

```
grunt> letterGroup = GROUP letters BY letter;
```

Create a bag for each unique character; the “grouped” bag will contain the same character for each occurrence of that character

INSPECT letterGroup bag:

```
grunt> describe letterGroup;
letterGroup: {group: chararray, letters: {(letter: chararray)}}
grunt> toDisplay = LIMIT letterGroup 5;
grunt> dump toDisplay;
(0, {(0), (0), (0)})
(a, {(a), (a)})
(2, {(2), (2), (2), (2), (2)}) ←
(3, {(3), (3), (3)})
(b, {(b)})
```

Next we'll need to convert characters occurrences into counts; Note this display was modified as there were too many characters to fit on the screen

Count the Number of Occurrences in Each Group

```
grunt> countPerLetter = FOREACH letterGroup  
GENERATE group, COUNT(letters);
```

For each row, count occurrence of the letter

INSPECT countPerLetter bag:

```
grunt> describe countPerLetter;  
countPerLetter: {group: chararray, long}  
grunt> toDisplay = LIMIT countPerLetter 5;  
grunt> dump toDisplay;  
(A, 728)  
(B, 325)  
(C, 291)  
(D, 194)  
(E, 264)
```

Each row now has the character and the number of times it was found to start a word. All we have to do is find the maximum

Descending Order the Group by the Count

```
grunt> orderedCountPerLetter = ORDER  
countPerLetter BY $1 DESC;
```

Simply order the bag by the first element, a number of occurrences for that element

INSPECT orderedCountPerLetter bag:

```
grunt> describe orderedCountPerLetter;  
orderedCountPerLetter: {group: chararray, long}  
grunt> toDisplay = LIMIT orderedCountPerLetter 5;  
grunt> dump toDisplay;  
(t,3711)  
(a,2379)  
(s,1938)  
(m,1787)  
(h,1725)
```

All we have to do now is just grab the first element

Grab the First Element

```
grunt> result = LIMIT orderedCountPerLetter 1;
```

The rows were already ordered in descending order, so simply limiting to one element gives us the result

INSPECT **orderedCountPerLetter** bag:

```
grunt> describe result;
result: {group: chararray, long}
grunt> dump result;
(t,3711)
```

There it is

Persist Result on a File System

```
grunt> STORE result INTO  
'/training/playArea/pig/mostSeenLetterOutput';
```

Result is saved under the provided directory

INSPECT result

```
$ hdfs dfs -cat  
/training/playArea/pig/mostSeenLetterOutput/part-r-00000  
t      3711
```

result

Notice that result was stored int part-r-0000, the regular artifact of a MapReduce reducer; Pig compiles Pig Latin into MapReduce code and executes.

MostSeenStartLetter.pig Script

```
-- 1: Load text into a bag, where a row is a line of text
lines = LOAD '/training/data/hamlet.txt' AS (line:chararray);
-- 2: Tokenize the provided text
tokens = FOREACH lines GENERATE flatten(TOKENIZE(line)) AS token:chararray;
-- 3: Retain first letter of each token
letters = FOREACH tokens GENERATE SUBSTRING(token,0,1) AS letter:chararray;
-- 4: Group by letter
letterGroup = GROUP letters BY letter;
-- 5: Count the number of occurrences in each group
countPerLetter = FOREACH letterGroup GENERATE group, COUNT(letters);
-- 6: Descending order the group by the count
orderedCountPerLetter = ORDER countPerLetter BY $1 DESC;
-- 7: Grab the first element => Most occurring letter
result = LIMIT orderedCountPerLetter 1;
-- 8: Persist result on a file system
STORE result INTO '/training/playArea/pig/mostSeenLetterOutput';
```

- Execute the script:
 - \$ pig MostSeenStartLetter.pig

Chapter 18

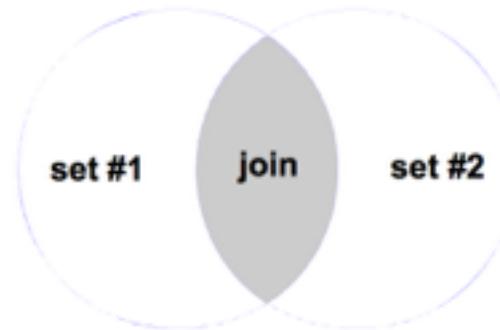
JOINING DATA-SETS

Joins Overview

- Critical Tool for Data Processing
- Will probably be used in most of your Pig scripts
- Pigs supports
 - Inner Joins
 - Outer Joins
 - Full Joins

How to Join in Pig

- **Join Steps**
 - Load records into a bag from input #1
 - Load records into a bag from input #2
 - Join the 2 data-sets (bags) by provided join key
- **Default Join is Inner Join**
 - Rows are joined where the keys match
 - Rows that do not have matches are not included in the result



Simple Inner Join Example

```
--InnerJoin.pig
```

1: Load records into a bag from input #1

```
posts = load '/training/data/user-posts.txt' using PigStorage(',')  
as (user:chararray,post:chararray,date:long);
```

1: Load records into a bag from input #2

Use comma as a separator

```
likes = load '/training/data/user-likes.txt' using PigStorage(',')  
as (user:chararray,likes:int,date:long);
```

```
userInfo = join posts by user, likes by user;
```

3: Join the 2 data-sets

```
dump userInfo;
```

When a key is equal in both data-sets then the rows are joined into a new single row; In this case when user name is equal

Execute InnerJoin.pig

```
$ hdfs dfs -cat /training/data/user-posts.txt  
user1,Funny Story,1343182026191  
user2,Cool Deal,1343182133839  
user4,Interesting Post,1343182154633  
user5,Yet Another Blog,13431839394
```

```
$ hdfs dfs -cat /training/data/user-likes.txt  
user1,12,1343182026191  
user2,7,1343182139394  
user3,0,1343182154633  
user4,50,1343182147364
```

```
$ pig $PLAY_AREA/pig/scripts-samples/InnerJoin.pig  
(user1,Funny Story,1343182026191,user1,12,1343182026191)  
(user2,Cool Deal,1343182133839,user2,7,1343182139394)  
(user4,Interesting Post,1343182154633,user4,50,1343182147364)
```



user1, user2 and user4 are id that exist in both data-sets; the values for these records have been joined.

Field Names After Join

- Join re-uses the names of the input fields and prepends the name of the input bag
 - `<bag_name>::<field_name>`

```
grunt> describe posts;
posts: {user: chararray,post: chararray,date: long}
grunt> describe likes;
likes: {user: chararray,likes: int,date: long}
```

```
grunt> describe userInfo; ← Schema of the resulting Bag
UserInfo: {
    posts::user: chararray,
    posts::post: chararray,
    posts::date: long,
    likes::user: chararray,
    likes::likes: int,
    likes::date: long}
```

Fields that were joined from 'posts' bag

Fields that were joined from 'likes' bag

Join By Multiple Keys

- Must provide the same number of keys
 - Each key must be of the same type
-

```
--InnerJoinWithMultipleKeys.pig
posts = load '/training/data/user-posts.txt'
    using PigStorage(',')
        as (user:chararray,post:chararray,date:long);

likes = load '/training/data/user-likes.txt'
    using PigStorage(',')
        as (user:chararray,likes:int,date:long);

userInfo = join posts by (user,date), likes by (user,date);
dump userInfo;
```

Only join records whose
user **and** date are equal

Execute InnerJoinWithMultipleKeys.pig

```
$ hdfs dfs -cat /training/data/user-posts.txt
user1,Funny Story,1343182026191
user2,Cool Deal,1343182133839
user4,Interesting Post,1343182154633
user5,Yet Another Blog,13431839394
```

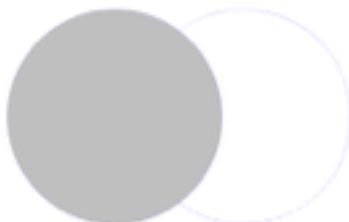
```
$ hdfs dfs -cat /training/data/user-likes.txt
user1,12,1343182026191
user2,7,1343182139394
user3,0,1343182154633
User4,50,1343182147364
```

```
$ pig $PLAY_AREA/pig/scripts/InnerJoinWithMultipleKeys.pig
(user1,Funny Story,1343182026191,user1,12,1343182026191)
```

There is only 1 record in each data-set
where both user and date are equal

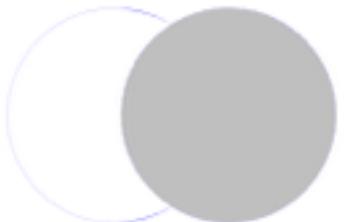
Outer Join

- Records which will not join with the ‘other’ record-set are still included in the result



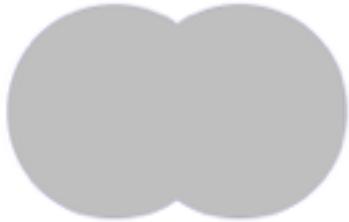
Left Outer

- Records from the first data-set are included whether they have a match or not. Fields from the unmatched (second) bag are set to null.



Right Outer

- The opposite of Left Outer Join: Records from the second data-set are included no matter what. Fields from the unmatched (first) bag are set to null.



Full Outer

- Records from both sides are included. For unmatched records the fields from the ‘other’ bag are set to null.

Left Outer Join Example

```
--LeftOuterJoin.pig
posts = load '/training/data/user-posts.txt'
    using PigStorage(',')
        as (user:chararray,post:chararray,date:long);

likes = load '/training/data/user-likes.txt'
    using PigStorage(',')
        as (user:chararray,likes:int,date:long);

userInfo = join posts by user left outer, likes by user;
dump userInfo;
```



Records in the posts bag will be in the result-set even if there isn't a match by user in the likes bag

Execute LeftOuterJoin.pig

```
$ hdfs dfs -cat /training/data/user-posts.txt  
user1,Funny Story,1343182026191  
user2,Cool Deal,1343182133839  
user4,Interesting Post,1343182154633  
user5,Yet Another Blog,13431839394  
  
$ hdfs dfs -cat /training/data/user-likes.txt  
user1,12,1343182026191  
user2,7,1343182139394  
user3,0,1343182154633  
User4,50,1343182147364  
  
$ pig $PLAY_AREA/pig/scripts/LeftOuterJoin.pig  
(user1,Funny Story,1343182026191,user1,12,1343182026191)  
(user2,Cool Deal,1343182133839,user2,7,1343182139394)  
(user4,Interesting Post,1343182154633,user4,50,1343182147364)  
(user5,Yet Another Blog,13431839394,,)
```

User5 is in the posts data-set
but NOT in the likes data-set



Right Outer and Full Join

```
--RightOuterJoin.pig
posts = LOAD '/training/data/user-posts.txt'
        USING PigStorage(',')
        AS (user:chararray,post:chararray,date:long);
likes = LOAD '/training/data/user-likes.txt'
        USING PigStorage(',')
        AS (user:chararray,likes:int,date:long);
userInfo = JOIN posts BY user RIGHT OUTER, likes BY user;
DUMP userInfo;

--FullOuterJoin.pig
posts = LOAD '/training/data/user-posts.txt'
        USING PigStorage(',')
        AS (user:chararray,post:chararray,date:long);
likes = LOAD '/training/data/user-likes.txt'
        USING PigStorage(',')
        AS (user:chararray,likes:int,date:long);
userInfo = JOIN posts BY user FULL OUTER, likes BY user;
DUMP userInfo;
```

Cogroup

- Joins data-sets preserving structure of both sets
- Creates tuple for each key
 - Matching tuples from each relationship become fields

```
--Cogroup.pig
posts = LOAD '/training/data/user-posts.txt'
        USING PigStorage(',')
        AS (user:chararray,post:chararray,date:long);
likes = LOAD '/training/data/user-likes.txt'
        USING PigStorage(',')
        AS (user:chararray,likes:int,date:long);
userInfo = COGROU P posts BY user, likes BY user;
DUMP userInfo;
```

Execute Cogroup.pig

```
$ hdfs dfs -cat /training/data/user-posts.txt  
user1,Funny Story,1343182026191  
user2,Cool Deal,1343182133839  
user4,Interesting Post,1343182154633  
user5,Yet Another Blog,13431839394
```

```
$ hdfs dfs -cat /training/data/user-likes.txt  
user1,12,1343182026191  
user2,7,1343182139394  
user3,0,1343182154633  
User4,50,1343182147364
```

```
$ pig $PLAY_AREA/pig/scripts/Cogroup.pig  
(user1,{{user1,Funny Story,1343182026191}},{{user1,12,1343182026191}})  
(user2,{{user2,Cool Deal,1343182133839}},{{user2,7,1343182139394}})  
(user3,{},{{user3,0,1343182154633}})  
(user4,{{user4,Interesting Post,1343182154633}},{{user4,50,1343182147364}})  
(user5,{{user5,Yet Another Blog,13431839394}},{})
```

Tuple per key

First field is a bag which came from posts bag (first data-set); second bag is from the likes bag (second data-set)

Cogroup with INNER

- Cogroup by default is an OUTER JOIN
 - You can remove empty records with empty bags by performing INNER on each bag
 - ‘INNER JOIN’ like functionality
-

```
--CogroupInner.pig
posts = LOAD '/training/data/user-posts.txt'
        USING PigStorage(',')
        AS (user:chararray,post:chararray,date:long);
likes = LOAD '/training/data/user-likes.txt'
        USING PigStorage(',')
        AS (user:chararray,likes:int,date:long);
userInfo = COGROU P posts BY user INNER, likes BY user INNER;
DUMP userInfo;
```

Execute CogroupInner.pig

```
$ hdfs dfs -cat /training/data/user-posts.txt  
user1,Funny Story,1343182026191  
user2,Cool Deal,1343182133839  
user4,Interesting Post,1343182154633  
user5,Yet Another Blog,13431839394
```

```
$ hdfs dfs -cat /training/data/user-likes.txt  
user1,12,1343182026191  
user2,7,1343182139394  
user3,0,1343182154633  
User4,50,1343182147364
```

```
$ pig $PLAY_AREA/pig/scripts/CogroupInner.pig  
(user1,{{user1,Funny Story,1343182026191}},{{user1,12,1343182026191}})  
(user2,{{user2,Cool Deal,1343182133839}},{{user2,7,1343182139394}})  
(user4,{{user4,Interesting Post,1343182154633}},{{user4,50,1343182147364}})
```

Records with empty bags are removed

Chapter 19

HIVE

Hive

- Data Warehousing Solution built on top of Hadoop
- Provides SQL-like query language named HiveQL
 - Minimal learning curve for people with SQL expertise
 - Data analysts are target audience
- Early Hive development work started at Facebook in 2007
- Today Hive is an Apache project under Hadoop
 - <http://hive.apache.org>



Hive Provides

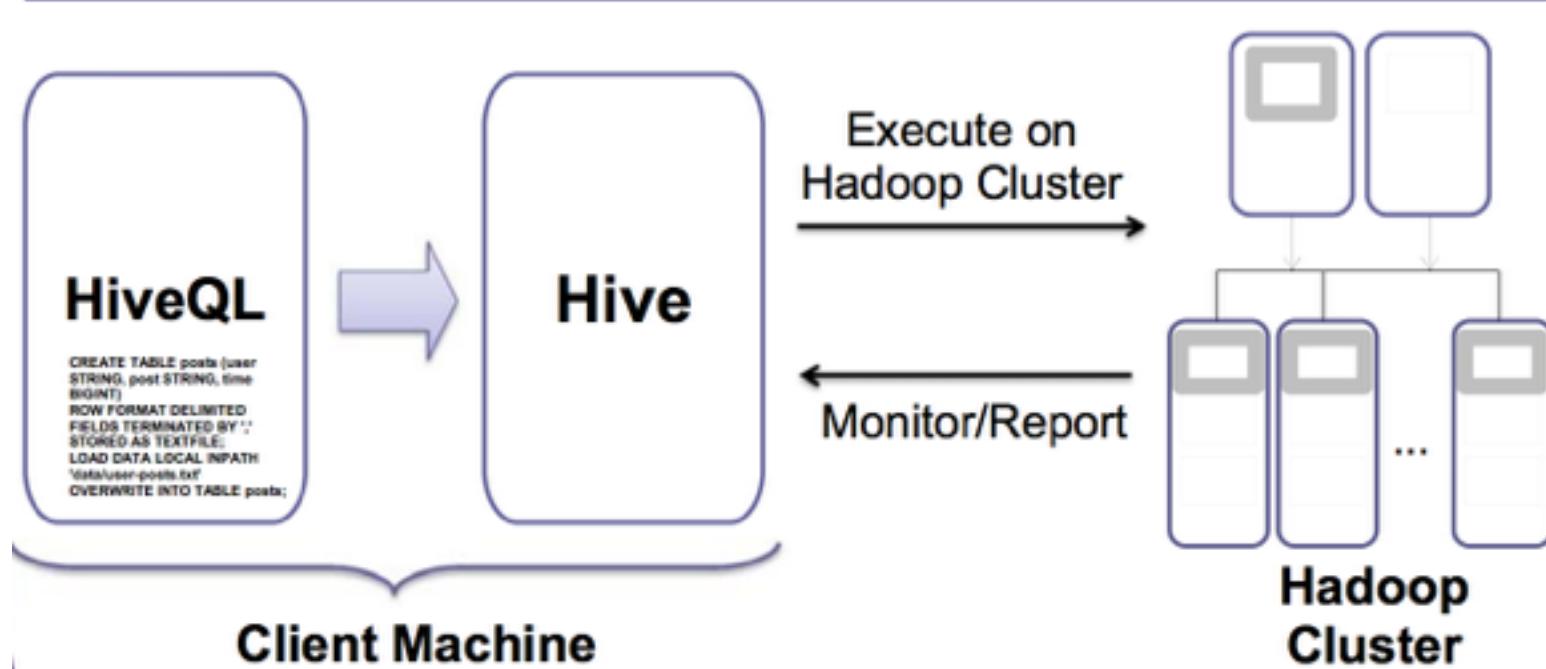
- Ability to bring structure to various data formats
- Simple interface for ad hoc querying, analyzing and summarizing large amounts of data
- Access to files on various data stores such as HDFS and HBase

Hive

- Hive does NOT provide low latency or real- time queries
- Even querying small amounts of data may take minutes
- Designed for scalability and ease-of-use rather than low latency responses

Hive

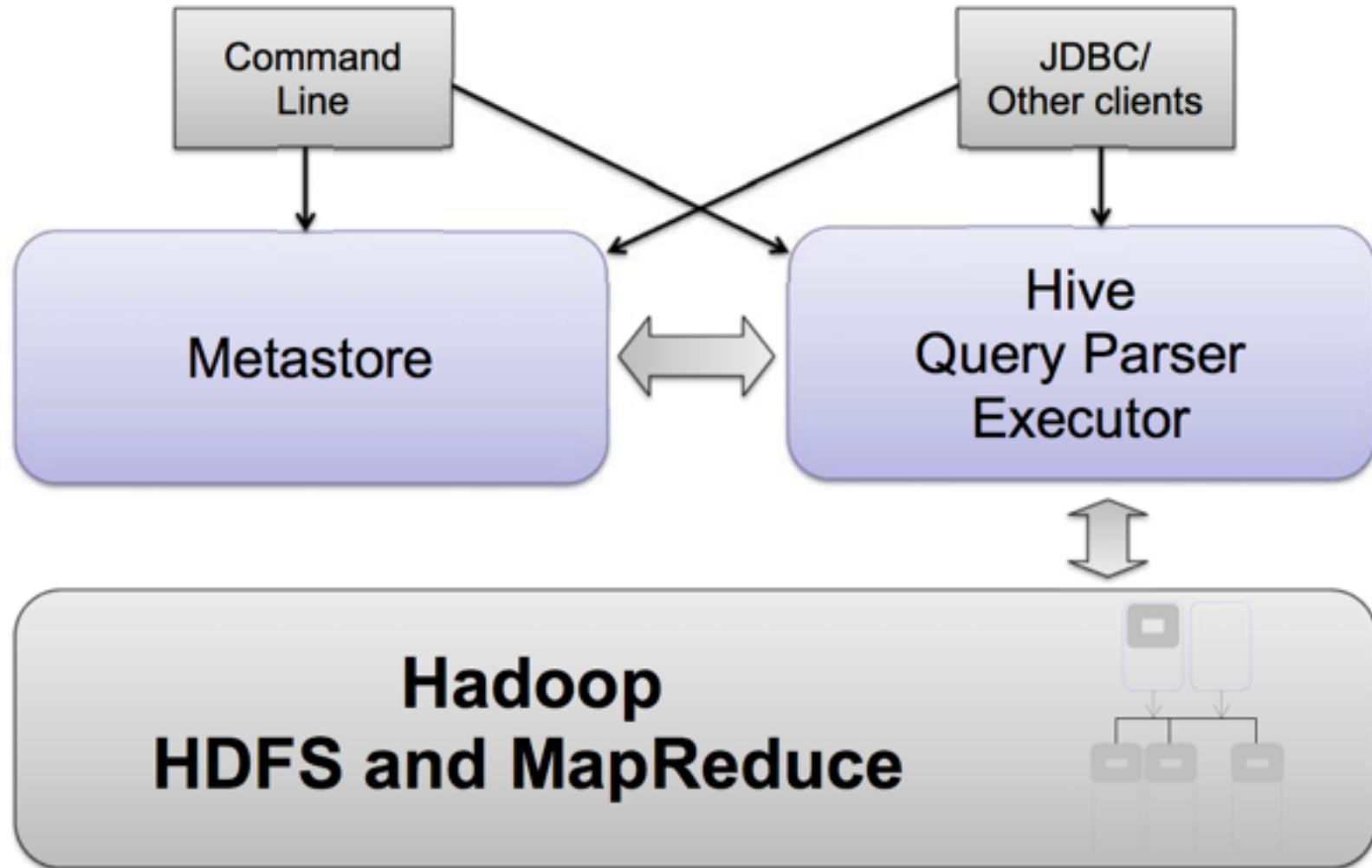
- Translates HiveQL statements into a set of MapReduce Jobs which are then executed on a Hadoop Cluster



Hive Metastore

- To support features like schema(s) and data partitioning Hive keeps its metadata in a Relational Database
 - Packaged with Derby, a lightweight embedded SQL DB
 - Schema is not shared between users
 - Stored in `metastore_db` directory which resides in the directory that hive was started from
 - Can easily switch another SQL installation such as MySQL

Hive Architecture

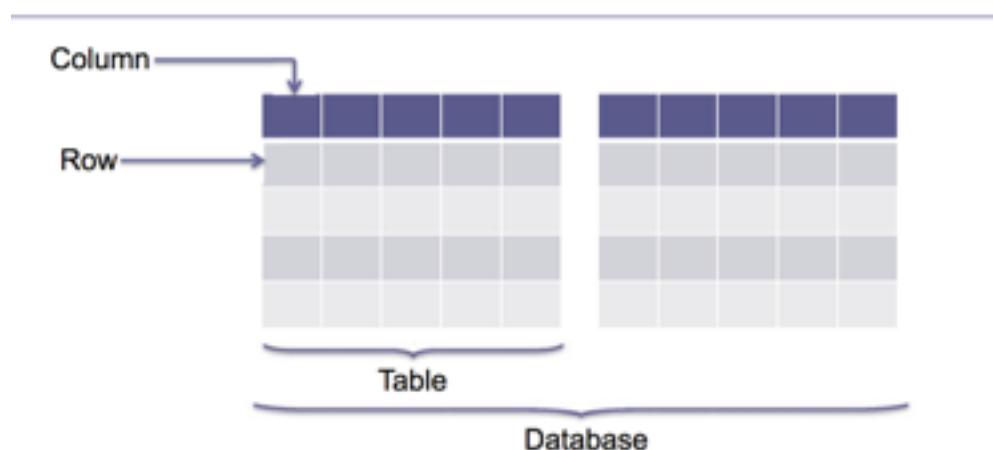


Hive Interface Options

- **Command Line Interface (CLI)**
 - Will use exclusively in these slides
- **Hive Web Interface**
 - <https://cwiki.apache.org/confluence/display/Hive/HiveWebInterface>
- **Java Database Connectivity (JDBC)**
 - <https://cwiki.apache.org/confluence/display/Hive/HiveClient>

Hive Concepts

- Re-used from Relational Databases
 - Database: Set of Tables, used for name conflicts resolution
 - Table: Set of Rows that have the same schema (same columns)
 - Row: A single record; a set of columns
 - Column: provides value and type for a single value



Run Hive

- HDFS and YARN need to be up and running

```
$ hive
Hive history file=/tmp/hadoop/hive_job_log_hadoop_201207312052_1402761030.txt
hive>
```

Hive's Interactive Command Line Interface (CLI)

Create a Table

- Let's create a table to store data from \$PLAY_AREA/data/user-posts.txt

```
$ cd $PLAY_AREA  
$ hive  
Hive history file=/tmp/hadoop/hive_job_log_hadoop_201208022144_2014345460.txt  
  
hive> !cat data/user-posts.txt;  
user1,Funny Story,1343182026191  
user2,Cool Deal,1343182133839  
user4,Interesting Post,1343182154633  
user5,Yet Another Blog,13431839394  
hive>
```

Launch Hive Command Line Interface (CLI)

Location of the session's log file

Can execute local commands within CLI, place a command in between ! and ;

Values are separate by ',' and each row represents a record; first value is user name, second is post content and third is timestamp

Create a Table

```
hive> CREATE TABLE posts (user STRING, post STRING, time BIGINT)
      > ROW FORMAT DELIMITED
      > FIELDS TERMINATED BY ','
      > STORED AS TEXTFILE;
```

OK

Time taken: 10.606 seconds

1st line: creates a table with 3 columns
2nd and 3rd line: how the underlying file should be parsed
4th line: how to store data

Statements must end with a semicolon and can span multiple rows

```
hive> show tables;
```

OK

posts

Time taken: 0.221 seconds

Display all of the tables

Result is displayed between 'OK' and 'Time taken...'

```
hive> describe posts;
```

OK

```
user    string
post    string
time    bigint
```

Time taken: 0.212 seconds

Display schema for posts table

Load Data Into a Table

```
hive> LOAD DATA LOCAL INPATH 'data/user-posts.txt'  
> OVERWRITE INTO TABLE posts;
```

Copying data from file:/home/hadoop/Training/play_area/data/user-posts.txt

Copying file: file:/home/hadoop/Training/play_area/data/user-posts.txt

Loading data to table default.posts

Deleted /user/hive/warehouse/posts

OK

Time taken: 5.818 seconds

hive>

Existing records in the table *posts* are deleted; data in *user-posts.txt* is loaded into Hive's *posts* table

Query Data

```
hive> select count (1) from posts;          ← Count number of records in posts table  
Total MapReduce jobs = 1                   ← Transformed HiveQL into 1 MapReduce Job  
Launching Job 1 out of 1  
...  
Starting Job = job_1343957512459_0004, Tracking URL =  
http://localhost:8088/proxy/application_1343957512459_0004/  
Kill Command = hadoop job -Dmapred.job.tracker=localhost:10040 -kill  
job_1343957512459_0004  
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1  
2012-08-02 22:37:24,962 Stage-1 map = 0%, reduce = 0%  
2012-08-02 22:37:30,497 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 0.87 sec  
2012-08-02 22:37:31,577 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 0.87 sec  
2012-08-02 22:37:32,664 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 2.64 sec  
MapReduce Total cumulative CPU time: 2 seconds 640 msec  
Ended Job = job_1343957512459_0004  
MapReduce Jobs Launched:  
Job 0: Map: 1 Reduce: 1 Accumulative CPU: 2.64 sec HDFS Read: 0 HDFS Write: 0  
SUCES  
Total MapReduce CPU Time Spent: 2 seconds 640 msec  
OK  
4 ← Result is 4 records  
Time taken: 14.204 seconds
```

Query Data

```
hive> select * from posts where user="user2";
```

```
...
```

```
...
```

```
OK
```

```
user2 Cool Deal 1343182133839
```

```
Time taken: 12.184 seconds
```

Select records for "user2"

Select records whose
timestamp is less or equals
to the provided value

```
hive> select * from posts where time<=1343182133839 limit 2;
```

```
...
```

```
...
```

```
OK
```

```
user1 Funny Story 1343182026191
```

```
user2 Cool Deal 1343182133839
```

```
Time taken: 12.003 seconds
```

```
hive>
```

Usually there are too
many results to display,
then one could utilize
limit command to
bound the display

Drop the Table

```
hive> DROP TABLE posts; ← Remove the table; use with caution
OK
Time taken: 2.182 seconds

hive> exit;
```

Loading Data

- Several options to start using data in HIVE
 - Load data from HDFS location

```
hive> LOAD DATA INPATH '/training/hive/user-posts.txt'  
> OVERWRITE INTO TABLE posts;
```

- File is copied from the provided location to /user/hive/warehouse/
- Load data from a local file system

```
hive> LOAD DATA LOCAL INPATH 'data/user-posts.txt'  
> OVERWRITE INTO TABLE posts;
```

- File is copied from the provided location to /user/hive/warehouse/
- Utilize an existing location on HDFS
 - Just point to an existing location

Re-Use Existing HDFS Location

```
hive> CREATE EXTERNAL TABLE posts  
> (user STRING, post STRING, time BIGINT)  
> ROW FORMAT DELIMITED  
> FIELDS TERMINATED BY ','  
> STORED AS TEXTFILE  
> LOCATION '/training/hive/' ;
```

OK

Time taken: 0.077 seconds

hive>

Hive will load all the files under
/training/hive directory in posts table

Schema Violations

```
hive> !cat data/user-posts-inconsistentFormat.txt;
user1,Funny Story,1343182026191
user2,Cool Deal,2012-01-05 ←
user4,Interesting Post,1343182154633
user5,Yet Another Blog,13431839394
```

```
hive> describe posts;
OK
user  string
post  string ←
time  bigint
Time taken: 0.289 seconds
```

Third Column 'post' is of type bigint;
will not be able to convert
'2012-01-05' value

Schema Violations

```
hive> LOAD DATA LOCAL INPATH  
      > 'data/user-posts-inconsistentFormat.txt'  
      > OVERWRITE INTO TABLE posts;  
OK  
Time taken: 0.612 seconds
```

```
hive> select * from posts;  
OK  
user1 Funny Story 1343182026191  
user2 Cool Deal    NULL ←  
user4 Interesting Post 1343182154633  
user5 Yet Another Blog 13431839394  
Time taken: 0.136 seconds  
hive>
```

null is set for any value that
violates pre-defined schema

Partitions

- To increase performance Hive has the capability to partition data
 - The values of partitioned column divide a table into segments
 - Entire partitions can be ignored at query time
 - Similar to relational databases' indexes but not as granular
- Partitions have to be properly created by users
 - When inserting data must specify a partition
- At query time, whenever appropriate, Hive will automatically filter out partitions

Creating Partitioned Table

```
hive> CREATE TABLE posts (user STRING, post STRING, time BIGINT)
      > PARTITIONED BY(country STRING)
      > ROW FORMAT DELIMITED
      > FIELDS TERMINATED BY ','
      > STORED AS TEXTFILE;
```

Partition table based on the value of a country.

```
OK
Time taken: 0.116 seconds
```

```
hive> describe posts;
```

```
OK
user    string
post    string
time    bigint
countrystring
Time taken: 0.111 seconds
```

There is no difference in schema between "partition" columns and "data" columns

```
hive> show partitions posts;
```

```
OK
Time taken: 0.102 seconds
hive>
```

Load Data Into Partitioned Table

```
hive> LOAD DATA LOCAL INPATH 'data/user-posts-US.txt'  
> OVERWRITE INTO TABLE posts;
```

FAILED: Error in semantic analysis: Need to specify partition columns because the destination table is partitioned

Since the posts table was defined to be partitioned
any insert statement must specify the partition

```
hive> LOAD DATA LOCAL INPATH 'data/user-posts-US.txt'  
> OVERWRITE INTO TABLE posts PARTITION(country='US');
```

OK

Time taken: 0.225 seconds

```
hive> LOAD DATA LOCAL INPATH 'data/user-posts-AUSTRALIA.txt'  
> OVERWRITE INTO TABLE posts PARTITION(country='AUSTRALIA');
```

OK

Time taken: 0.236 seconds

Each file is loaded into separate partition;
data is separated by country

Partitioned Table

- Partitions are physically stored under separate directories
-

```
hive> show partitions posts;
```

```
OK
```

```
country=AUSTRALIA
```

```
country=US
```

```
Time taken: 0.095 seconds
```

```
hive> exit;
```

```
$ hdfs dfs -ls -R /user/hive/warehouse/posts
```

```
/user/hive/warehouse/posts/country=AUSTRALIA
```

```
/user/hive/warehouse/posts/country=AUSTRALIA/user-posts-AUSTRALIA.txt
```

```
/user/hive/warehouse/posts/country=US
```

```
/user/hive/warehouse/posts/country=US/user-posts-US.txt
```



There is a directory for each partition value



Querying Partitioned Table

- There is no difference in syntax
- When partitioned column is specified in the where clause entire directories/partitions could be ignored

Only "COUNTRY=US" partition will be queried,
"COUNTRY=AUSTRALIA" partition will be ignored

```
hive> select * from posts where country='US' limit 10;
OK
user1 Funny Story 1343182026191      US
user2 Cool Deal   1343182133839      US
user2 Great Interesting Note 13431821339485      US
user4 Interesting Post 1343182154633      US
user1 Humor is good   1343182039586      US
user2 Hi I am user #2 1343182133839      US
Time taken: 0.197 seconds
```

Bucketing

- Mechanism to query and examine random samples of data
- Break data into a set of buckets based on a hash function of a "bucket column"
 - Capability to execute queries on a sub-set of random data
- Doesn't automatically enforce bucketing
 - User is required to specify the number of buckets by setting # of reducer

```
hive> mapred.reduce.tasks = 256;  
OR  
hive> hive.enforce.bucketing = true;
```

Either manually set the # of reducers to be the number of buckets or you can use 'hive.enforce.bucketing' which will set it on your behalf

Create and Use Table with Buckets

```
hive> CREATE TABLE post_count (user STRING, count INT)
      > CLUSTERED BY (user) INTO 5 BUCKETS;          ← Declare table with 5
                                                    buckets for user column
OK
Time taken: 0.076 seconds

hive> set hive.enforce.bucketing = true; ← # of reducer will get set 5
hive> insert overwrite table post_count
      > select user, count(post) from posts group by user;
Total MapReduce jobs = 2
Launching Job 1 out of 2
...
Launching Job 2 out of 2
...
OK
Time taken: 42.304 seconds
hive> exit;
$ hdfs dfs -ls -R /user/hive/warehouse/post_count/
/user/hive/warehouse/post_count/000000_0
/user/hive/warehouse/post_count/000001_0
/user/hive/warehouse/post_count/000002_0
/user/hive/warehouse/post_count/000003_0
/user/hive/warehouse/post_count/000004_0          ← A file per bucket is
                                                    created; now only a
                                                    sub-set of buckets can
                                                    be sampled
```

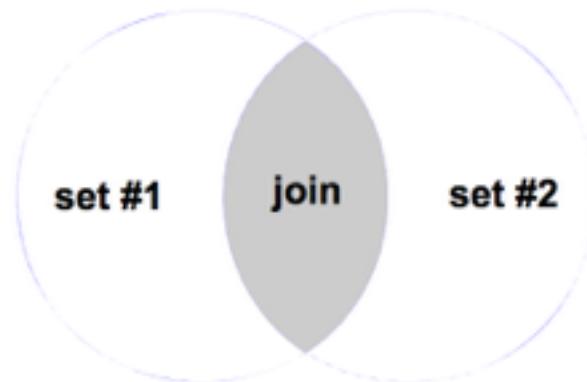
Random Sample of Bucketed Table

```
hive> select * from post_count TABLESAMPLE(BUCKET 1 OUT OF 2);  
OK  
user5 1  
user1 2  
Time taken: 11.758 seconds  
hive>
```

Sample approximately 1 for every 2 buckets

Joins

- Joins in Hive are trivial
- Supports outer joins
 - left, right and full joins
- Can join multiple tables
- Default Join is Inner Join
 - Rows are joined where the keys match
 - Rows that do not have matches are not included in the result



Simple Inner Join

- Let's say we have 2 tables: posts and likes

```
hive> select * from posts limit 10;
```

OK

user1	Funny Story	1343182026191
user2	Cool Deal	1343182133839
user4	Interesting Post	1343182154633
user5	Yet Another Blog	1343183939434

Time taken: 0.108 seconds

```
hive> select * from likes limit 10;
```

OK

user1	12	1343182026191
user2	7	1343182139394
user3	0	1343182154633
user4	50	1343182147364

Time taken: 0.103 seconds

```
hive> CREATE TABLE posts_likes (user STRING, post STRING, likes_count INT);
```

OK

Time taken: 0.06 seconds

We want to join these 2 data-sets
and produce a single table that
contains user, post and count of
likes



Simple Inner Join

```
hive> INSERT OVERWRITE TABLE posts_likes  
> SELECT p.user, p.post, l.count  
> FROM posts p JOIN likes l ON (p.user = l.user);  
OK  
Time taken: 17.901 seconds
```

Two tables are joined based on user column; 3 columns are selected and stored in posts_likes table

```
hive> select * from posts_likes limit 10;  
OK  
user1 Funny Story      12  
user2 Cool Deal        7  
user4 Interesting Post  50  
Time taken: 0.082 seconds  
hive>
```

Outer Join

- Rows which will not join with the ‘other’ table are still included in the result
- **Left Outer**
- **Right Outer**
- **Full Outer**

Outer Join Examples

```
SELECT p.*, l.*  
FROM posts p LEFT OUTER JOIN likes l ON (p.user = l.user)  
limit 10;
```

```
SELECT p.*, l.*  
FROM posts p RIGHT OUTER JOIN likes l ON (p.user = l.user)  
limit 10;
```

```
SELECT p.*, l.*  
FROM posts p FULL OUTER JOIN likes l ON (p.user = l.user)  
limit 10;
```

THANK YOU!