Welcome to:

# Unit 2 Hadoop Fundamentals

# Unit objectives

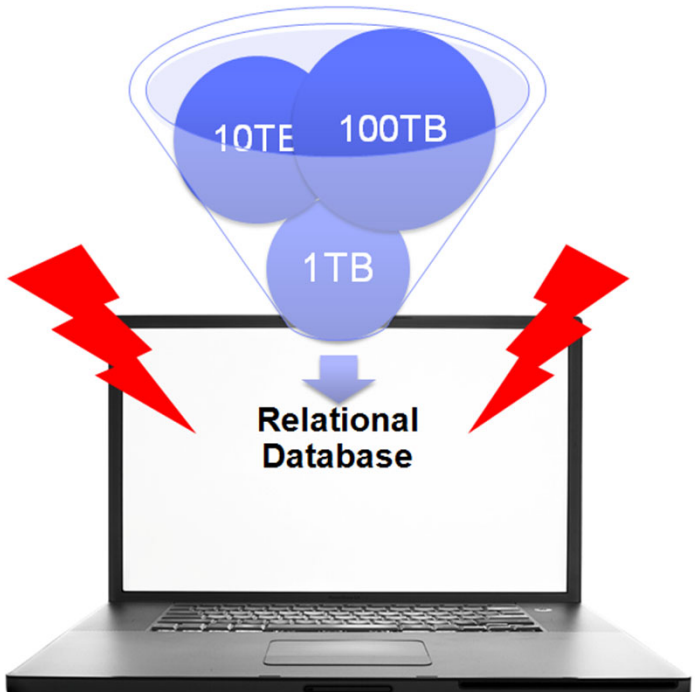


After completing this unit, you should be able to:

- Define Hadoop concept and Hadoop File System (HDFS)
- Administer HDFS on server system
- Define the concept of Map / Reduce
- Setup an Hadoop Cluster
- Manage Job Execution using HDFS
- Understand JAQL
- Load the data
- Overview the workflow engine

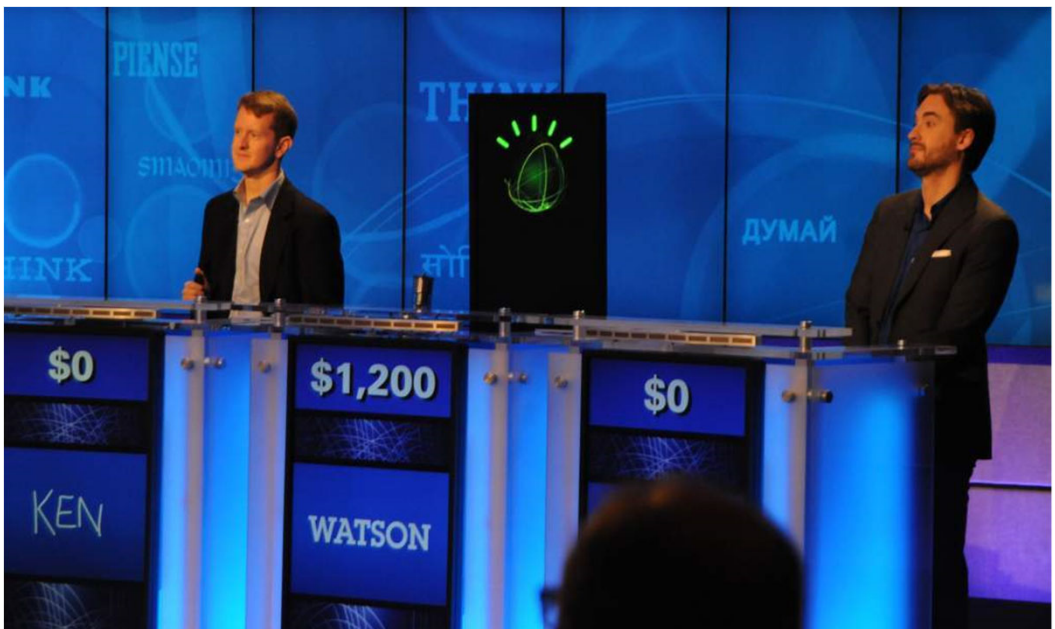
# What is Hadoop?





# Examples of Hadoop in action – IBM Watson





# Examples of Hadoop in action



- In the telecommunication industry
- In the media
- In the technology industry

# Introduction to Hadoop



- Hadoop is designed to efficiently handle large volumes of information, connecting many computers' consumption together to work in parallel.
- Hadoop's unique simplified programming model allows the user to write quickly and test distributed systems, and its efficiency for automatic distribution of data.
- In a Hadoop cluster, the data is distributed to all the nodes in the cluster as it is being loaded
- Hadoop ecosystem do the batch processing at large scale with the help of multiple nodes .
- In Hadoop system, big problem are usually broken and distributed in smaller elements to achieve the quick and cost effective processing. Smaller elements can be processed in parallel and then they regroup themselves to produce desired results.
- At its core, Hadoop has two primary components:
- **Hadoop Distributed File System:**
- **Map Reduce engine**

# Data Distribution



- The Hadoop Distributed File System (HDFS) will split up large data files into fragments that are managed by different nodes of the cluster.
- Each fragment is replicated in multiple computers, so that a single failure in a machine will not make the data unavailable.
- Content is universally accessible through a single namespace.
- Data is record-oriented in the Hadoop framework programming.
- Input files are divided into lines or other specific formats of the application logic. Each process running on a cluster node processes a subset of these records.
- The Hadoop framework programs processes in the proximity of the location of data and records using the knowledge of the distributed file system.
- Since the files are distributed throughout the distributed file system as fragments, each process in a computer node works with a subset of the data. The data operated on by a node is chosen based on their locality to the node.

# Flat Scalability



- Major advantage of using Hadoop is its scalability flat curve
- Running Hadoop in a limited amount of data in a small number of nodes cannot prove its particular stellar performance, as the overhead involved in the start programs of Hadoop is relatively high
- Another parallel/distributed programming paradigm such as MPI (Message Passing Interface) can operate much better in two, four, or maybe a dozen machines.
- If a program is written in Hadoop and runs on ten nodes, it is not necessary to work for the same program that runs on a much larger amount of hardware
- Orders of magnitude of growth can be managed with a little re-work required for their applications
- The Hadoop platform manages data and hardware resources and offers a performance increase proportional to the number of available machines

# HDFS-Hadoop Distributed File System



- HDFS was invented in Yahoo by Doug Cutting
  - It processes the internet scale data
  - It save costs by distributing workload on massive parallel system
- Tolerate high component failure rate
- Replication provides reliability to HDFS.
- HDFS follows the block system.
- Individual files are divided into blocks of a fixed size. These blocks are stored in a cluster of one or more computers with data storage capacity.
- Each machines in the cluster is called DataNodes. A file can consist of several blocks, and are not necessarily stored in the same machine.
- The target computers for each block are chosen at random in a block by block fashion. Therefore, access to a file may require the cooperation of several machines.
- If multiple machines are involved in carrying out a file, and the file becomes unavailable at the loss of any of these machines then HDFS combats this problem by playing each block through a series of machines (3 by default).

# Name Nodes



- Under HDFS, extremely large files are broken down into small pieces known as blocks.
- All the blocks are stored on data nodes. With the help of namenode only you can access the access the correct file on data node.
- Read, write and access right to the files are managed by the namenode only. Creation and replication of datablocks can be done with the help of namenode.
- File system ‘namespace’ refers to all the collection of all the files on cluster thus working as directory on the system.
- Namenode manages all the namespace.
- Genrally we will see that one name node and one datanode will be operating out of the one single server rack machine.
- Namenode and datanode does not always sits on single server rack machine yet there is a strong relationship between them.
- Servers can be added or removed under particular namenode as per the demand which leads us to the conclusion that cluster behave dynamically.
- Namenode conducts has much more intelligent functions in comparison to the datanodes.

Source - http://www.ibm.com/developerworks/library/wa-introhdf

# Data nodes

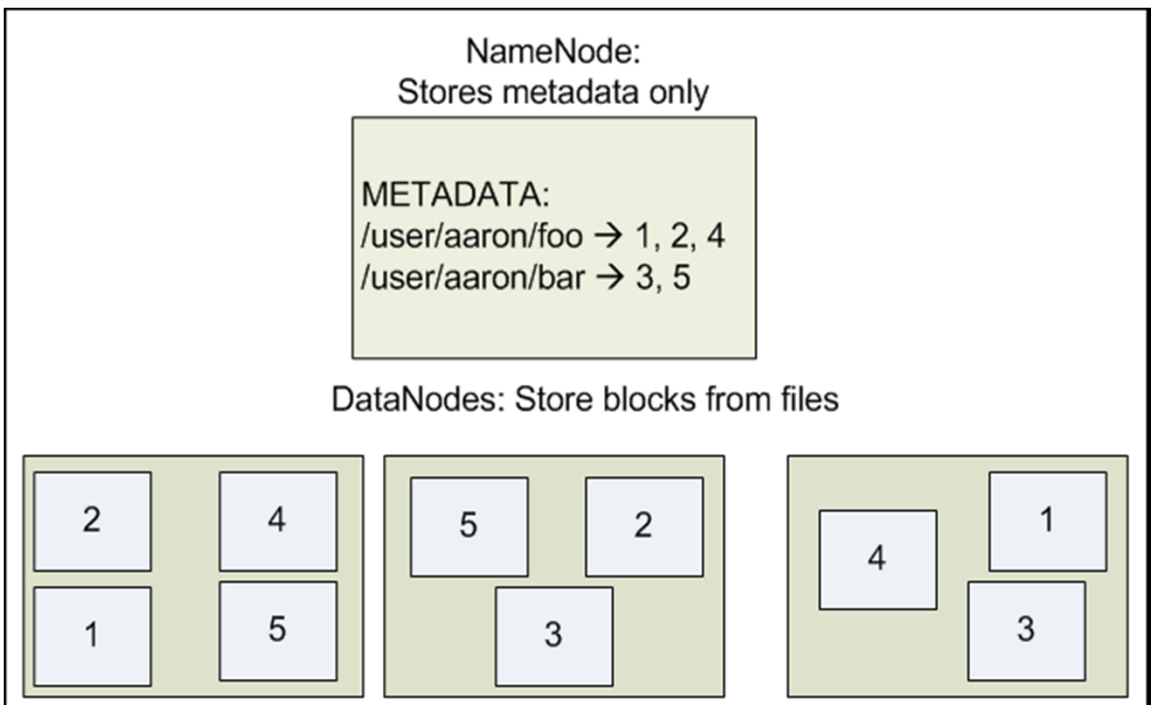


- Name nodes and data nodes are software components designed to run in a decoupled manner on commodity machines across heterogeneous operating systems.
- HDFS is built using the Java programming language; therefore, any machine that supports the Java programming language can run HDFS.
- A typical installation cluster has a dedicated machine that runs a name node and possibly one data node.
- Each of the other machines in the cluster runs one data node.
- Data nodes continuously loop, asking the name node for instructions.
- A name node can't connect directly to a data node; it simply returns values from functions invoked by a data node.
- Each data node maintains an open server socket so that client code or other data nodes can read or write data.
- The host or port for this server socket is known by the name node, which provides the information to interested clients or other data nodes.

Source - http://www.ibm.com/developerworks/library/wa-introhdf

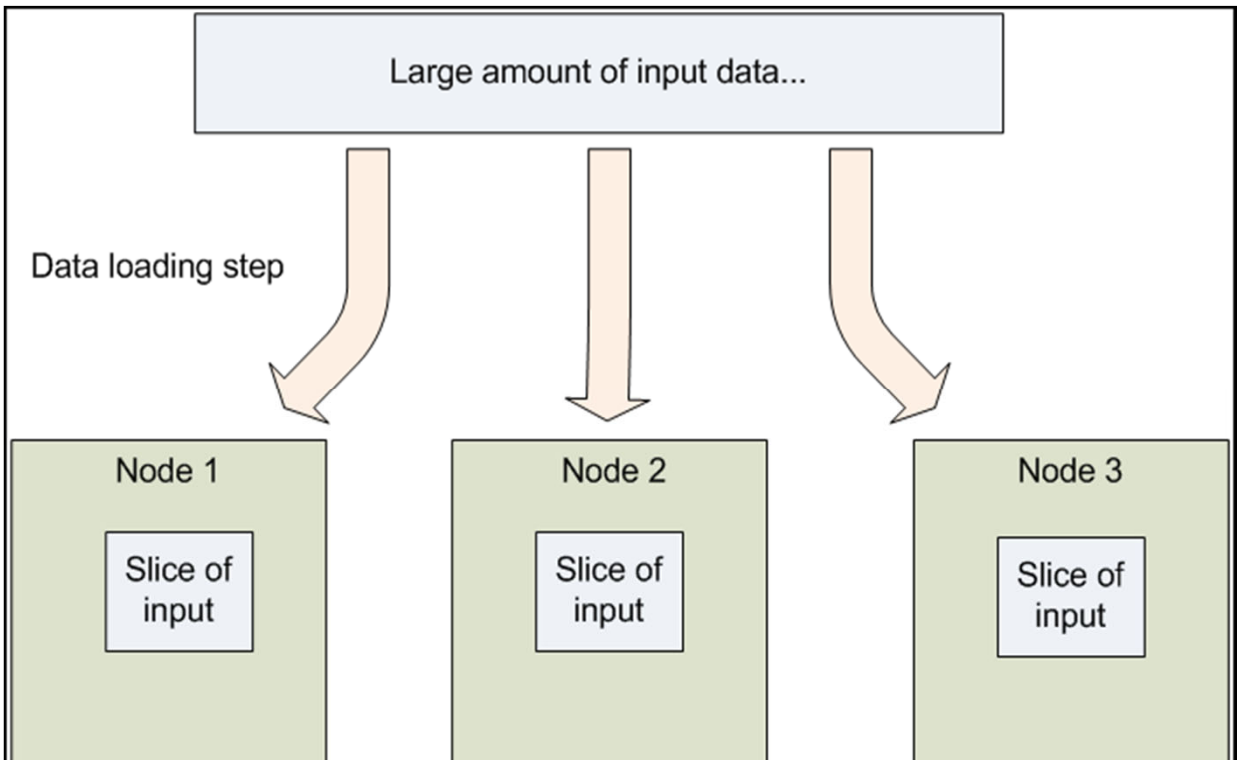
# DataNodes with Blocks of Multiple Files with a Replica of 2





# The data is distributed across nodes at the time of loading





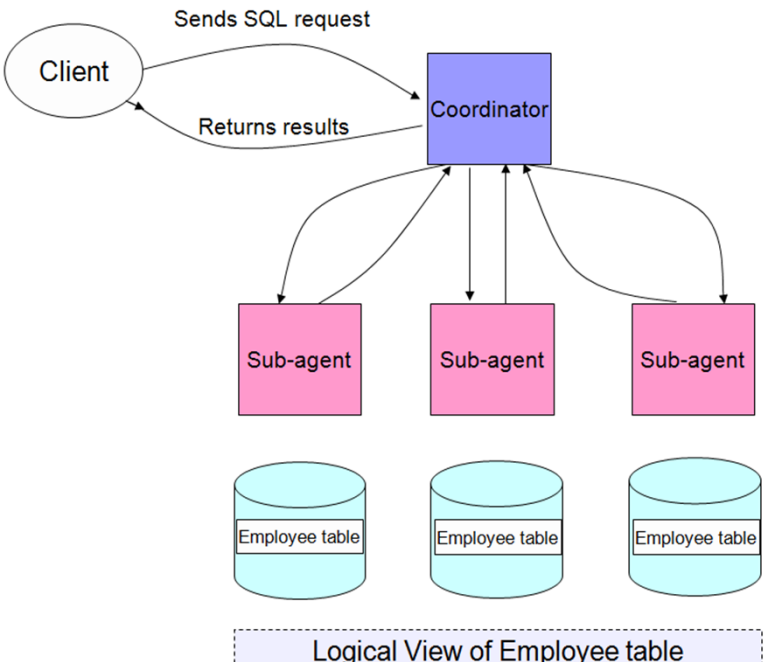
# MapReduce



- Designed to processes huge datasets for certain kinds of distributable problems using a large number of nodes
- Map
  - Master node partitions the input into smaller sub-problems
  - Distributes the sub-problems to the worker nodes
  - Worker nodes may do the same process
- Reduce
  - Master node then takes the answers to all the sub-problems
  - Combines them in some way to get the output
- Allows for distributed processing of the map and reduce operations

# An SQL example of MapReduce






1. Client sends SQL request to the coordinator node
2. Coordinator node sends request of appropriate sub-agent nodes
3. Sub-agent nodes process request against a subset of the data
4. Sub-agent nodes send results back to coordinator node
5. Coordinator node does additional processing
6. Coordinator nodes sends results back to the client

# The Map function



- A Map function
  - Takes a series of key / value pairs and processes each
  - Generates zero or more output key / value pairs
- Map functions can be run on all DataNodes in parallel
  - Requires that there are no dependencies between the data
- Output of Map is stored on local disk
- Example:
  - You want to count the occurrences of words in documents
  - The Map function reads in a line of text
  - For each word encountered in the text
    - Emits an output pair that contains the word (as the key) and a value of 1

# Sort phase



- Invoked automatically by the MapReduce framework
  - Output of key / value pairs from the Map function is sorted on the key value
- After the sort, records are sent to the Reduce function nodes
  - Records with the same key value from all Map function nodes are sent to the same Reduce function node
  - Records then become input into the Reduce function

# The Reduce function



- **REDUCE** is a family of higher-order functions that iterates an arbitrary function over a data structure in some order and builds up a return value
- Typically, a Reduce deals with two things:
  - A combining function and a list of elements of some data structure
- The fold then proceeds to combine elements of the data structure using the function in some systematic way
- Output of Reduce is stored in HDFS
- Reduce functions can run in parallel

- Example:
  - The key / value pair from the Map function is sorted (merged) on the key
  - Reads all of the key / value pairs for a particular word and totals the occurrences
  - Outputs another key / value pair that contains the word (as the key) and the total of the word occurrences

- In Hadoop a Reduce function always takes a Map function as an input

# Combiner and Partition functions



- Combiner function
  - Optional
  - Lessens the data transferred between Map and Reduce tasks
  - For each Map function
    - Takes the output and combines it into a lesser number of records
    - Becomes the input to a Reduce function

- Partition function
  - Can be used to override the default partition function
  - Is given a key value and the number of reducers
  - Returns an index of the desired reducer

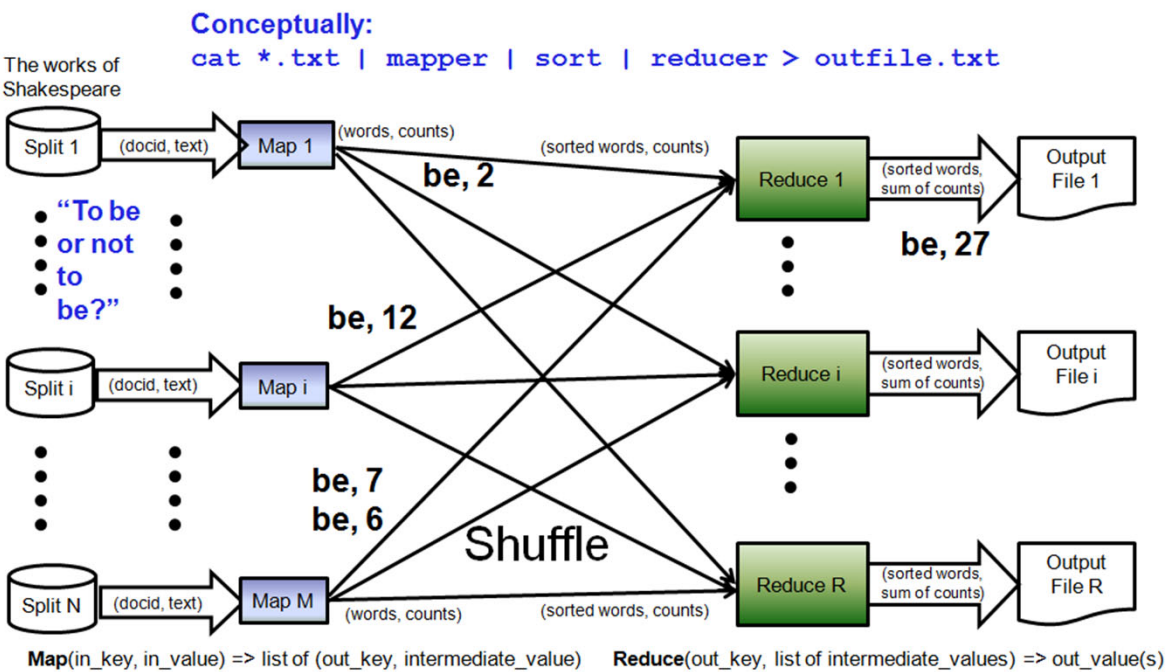
# Streaming and pipes



- Normally MapReduce programs are written in Java
- Hadoop streaming
  - Hadoop provides an API to allow the map and reduce function to be coded in other languages
    - Language must be able to read standard input
    - Language must be able to write standard output
  - UNIX standard streams is the interface between Hadoop and your program
- Hadoop pipes
  - C++ interface to Hadoop MapReduce
  - Uses sockets as the means for communications between the TaskTracker and the C++ map or reduce functions

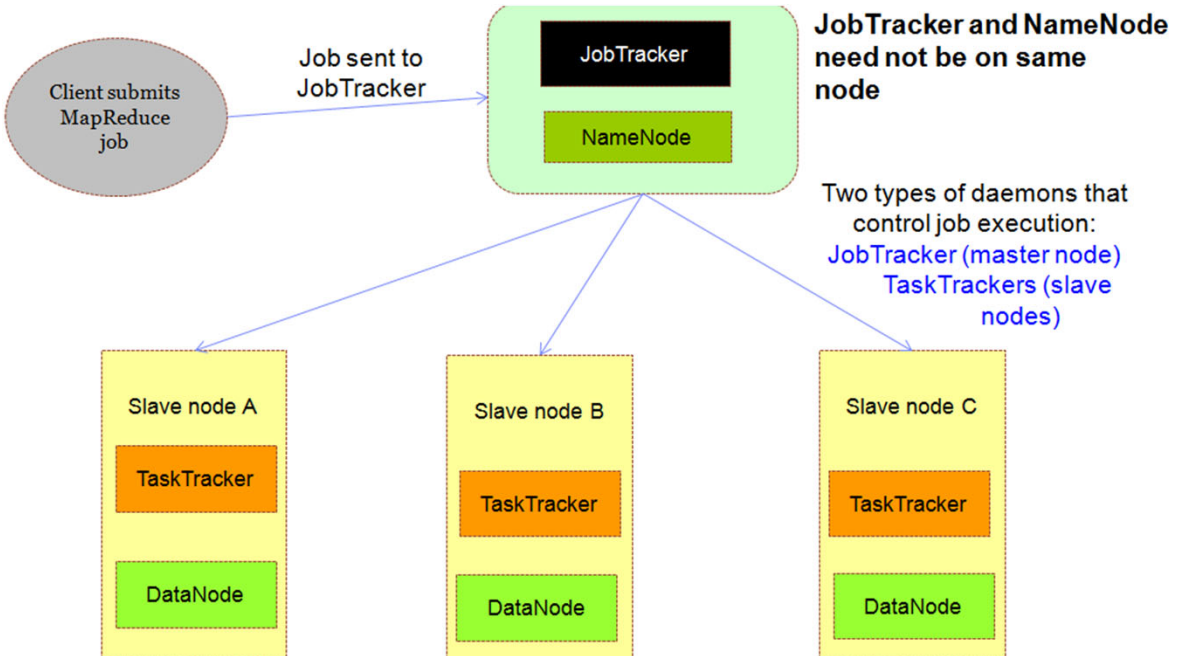
# MapReduce example: wordcount





# MapReduce co-locating with HDFS






**TaskTrackers (compute nodes) and DataNodes co-locate = high aggregate bandwidth across cluster**

# MapReduce Processing



- User runs a program on client computer
- Program submits a job to HDFS. Job contains:
  - Input data
  - MapReduce program
  - Configuration information
- Job sent to JobTracker
- JobTracker communicates with NameNode and assigns parts of a job to TaskTrackers (TaskTracker is run on each DataNode)
  - Task is a single MAP or REDUCE operation over piece of data
  - Hadoop divides the input to MAP / REDUCE jobs into equal splits
- The JobTracker knows (from NameNode) which nodes contain the data, and which other machines are nearby.
- TaskTracker does the processing and sends heartbeats to TaskTracker
  - TaskTracker sends heartbeats to the JobTracker.

# MapReduce Processing (cont)



- Any tasks that did not report within a prescribe time period (default is 10 min) assumed to be failed
  - Its JVM will be killed by TaskTracker and reported to the JobTracker
- The JobTracker will reschedule any failed tasks
  - Will attempt to use a different TaskTracker
- If same task fails 4 times, job fails
  - Fail value is configurable
- Any TaskTracker, reporting a high number of failed jobs on particular node or more than 4 failed tasks from the same job, will be blacklisted
  - No longer assigned tasks
  - Still communicates with the JobTracker
  - Faults expire over time
- JobTracker maintains and manages the status of each job
  - Results from failed tasks will be ignored

# Speculative execution



- Since an entire job cannot complete until all tasks complete
  - Job execution is time sensitive to the slowest running tasks
- Hadoop does not try to determine what is causing a task to run slowly
  - It just tried to detect that a task is running slower than expected
- If a slow running task is detected
  - Another equivalent task is launched
- When one of the two tasks complete
  - The other duplicate task is automatically killed
- Speculative execution is turned on by default

# MapReduce – a tale of two APIs



- A new Java MapReduce API was introduced in Hadoop 0.20.0
  - Referred to as Context Objects
  - Designed to make the API easier to evolve in the future
  - Type-incompatible with the older API
- New API uses abstract classes over interfaces
- New API makes use of context objects
  - New Context essentially encompasses the roles from the old API of
    - JobConf
    - OutputCollector
    - Reporter
- Naming of output files are slightly different between the two APIs
- **Note:** In this class, the new API will be discussed and used

# MapReduce Anatomy (1 of 4)



- There are four basic components of your MapReduce Java class
- There is the map component
  - This class extends the *Mapper* class
  - The *Mapper* class has four formal parameters that specify
    - The input key
    - The input value
    - The output key
    - The output value
  - Within the *Mapper* class, the *map()* method is referenced
    - It is passed the input key and value
    - It provides an instance of *Context* to write the output

# Basic map code



```
public class MyMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {
  . . .
  @Override
  public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    . . .
    context.write(new Text(…), new IntWritable(…));
    }
  }
}
```

# MapReduce Anatomy (2 of 4)



- There is the reduce component
  - This class extends the *Reducer* class
  - The *Reducer* class has four formal parameters that specify
    - The input key
    - The input value
    - The output key
    - The output value
  - Within the *Reducer* class, the *reduce()* method is referenced
    - It is passed the input key and value
      - Normally you want the input value to be iterable
    - It provides an instance of *Context* to write the output

# Basic reduce code



```
public class MyReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {
  @Override
  public void reduce(Text key,
                     Iterable<IntWritable> values,
            Context context)
    throws IOException, InterruptedException {
    . . .
    for (IntWritable value : values) {
    . . .
    }
    context.write(key, new IntWritable(…));
  }
}
```

# MapReduce Anatomy (3 of 4)



- There can be a combiner component
  - This class extends the *Reducer* class
    - It will do the same processing as the reducer
      - But takes place on the mapper node
  - Has the same four formal parameters that as the *Reducer* class
    - The input key
    - The input value
    - The output key
    - The output value
  - Within the *Reducer* class, the *reduce()* method is referenced
    - It is passed the input key and value
      - Normally you want the input value to be iterable
    - It provides an instance of *Context* to write the output

# MapReduce Anatomy (4 of 4)



- You tie the mapper, reducer, and combiner classes together in the *main()* method
  - This is what runs the MapReduce code
- Job control is performed through the *Job* class
- Parameters can be passed to your MapReduce application
  - For example
    - Locate input directory or file(s)
    - Located output directory

# What is HBase?



- HBase is the Hadoop database modeled after Google's Bigtable
  - "A Bigtable is a sparse, distributed, persistent multidimensional sorted map"
  - http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/archive/bigtable-osdi06.pdf
- Use HBase for random, real-time read/write access to your BigData
  - Runs on clusters of commodity hardware
- Open source Apache Top Level Project
  - Embraced and supported by IBM
  - Facebook, Twitter, and other leading websites uses HBase for their BigData
  - http://wiki.apache.org/hadoop/Hbase/PoweredBy
- HBase and its native API is written in Java, but you do not have to use Java to access its API.

# NoSQL technology



- HBase is a NoSQL data store
  - NoSQL = “Not only SQL”
  - Not intended to replace RDBMS
  - Suited for specific business needs
- Traditional RDBMS is not very scalable when data reaches the terabytes and petabytes realm
  - Sharding occurs when data volume goes up too high.
  - Sharding becomes a major challenge with RDBMS
    - Keeping referential integrity
    - Joins
    - Rebalancing
- A cost-effective way to use commodity hardware
- Flexible data models are needed to support BigData applications
  - We won’t always know the schema up front.
  - Records can be sparse (social media data is variable)

# CAP Theorem



- **C**onsistency
  - All clients see the same data at the same time
- **A**vailability
  - A guarantee that every request will see a response (success or fail)
- **P**artition Tolerance
  - The system will continue to operate even one parts of the system fails
- A distributed system can only guarantee two of the three CAP properties
- HBase is eventually consistent and implements consistency and partition tolerance.
  - For example, a Region Server failure is recoverable, but the data will be unavailable for a period of time

- http://en.wikipedia.org/wiki/CAP_theorem

# ACID Properties



- **A**tomicity: an operation is atomic if it either completes entirely or not at all
  - Only provides row level atomicity
- **C**onsistency & **I**solation
  - All actions cause the table to transition from one valid state to another
  - Scans are **not** consistent
  - A scan will always reflect the view of the data *at least as new* as the beginning of the scan.
  - Row-level consistency only. All rows returned will be a complete row that existed at some point in the table's history
- **D**urability: Any successful updates will not be lost
  - Any operation that returns a success will be made durable
- http://hbase.apache.org/acid-semantics.html

# Why HBase?



- Apache Hadoop includes HBase
  - Allows HBase to leverage the Map/Reduce model
- HBase can replace costly implementation of RDBMS for Big Data applications
  - HBase is not a one size fit all approach. It is designed for big data applications with specific data access patterns in mind. Not meant to replace RDBMS entirely!
- HBase enables horizontal scalability for very large data sets
  - Easily scalable by adding additional commodity hardware without interruption.
- HBase supports flexible data models of sparse records
  - We do not need to know the data types in advance. Keeps our schema very flexible.
- HBase supports random read/write access to the Big Data
  - Very quick and efficient, random reads and writes
- Sharding is automatic without consequences from the RDBMS approach
  - Built-in feature to HBase that can be customized

# Important things to keep in mind



- HBase is not design to replace RDBMS.
  - Does not support SQL
  - Not for transactional processing
  - Does not support table joins
- Not to be used as a general purpose database for Hadoop.
  - Use for specific cases where your data exhibits behavior supported by HBase (sparse data set, variable schemas, key-based access, partial dataset scans, etc.)
- Everything in HBase is stored as an array of bytes with the exception of the timestamp value, which is stored as a long integer.

# HBase vs. RDBMS



| | HBase | RDBMS |
|---|---|---|
| Hardware architecture | Similar to Hadoop. Clustered commodity hardware. Very affordable. | Typically large scalable multiprocessor systems. Very expensive. |
| Fault Tolerance | Built into the architecture. Lots of nodes means each is relatively insignificant. No need to worry about individual node downtime. | Requires configuration of the HW and the RDBMS with the appropriate high availability options. |
| Typical Database Size | Terabytes to Petabytes - hundred of millions to billions of rows. | Gigabytes to Terabytes – hundred of thousands to millions of rows. |
| Data Layout | A sparse, distributed, persistent, multidimensional sorted map. | Rows or column oriented. |
| Data Types | Bytes only. | Rich data type support. |
| Transactions | ACID support on a single row only | Full ACID compliance across rows and tables |
| Query Language | API primitive commands only, unless combined with Hive or other technology | SQL |
| Indexes | Row-Key only unless combined with other technologies such as Hive or IBM's BigSQL | Yes |
| Throughput | Millions of queries per second | Thousands of queries per second |

# For example…



Given this RDBMS:

| ID (Primary key) | Last name | First name | Password | Timestamp |
|---|---|---|---|---|
| 1234 | Smith | John | Hello, world! | 20130710 |
| 5678 | Cooper | Joyce | wysiwyg | 20120825 |
| 5678 | Cooper | Joyce | wisiwig | 20130916 |

Logical view in HBASE

| Row-Key | Value (CF, Qualifier, Version) |
|---|---|
| 1234 | info {'lastName': 'Smith', 'firstName': 'John'}<br>pwd {'password': 'Hello, world!'} |
| 5678 | info {'lastName': 'Cooper', 'firstName': 'Joyce'}<br>pwd {'password: 'wysiwyg'@ts 20130916,<br>'password': 'wisiwig'@ts 20120825} |

# Physical view in HBase



- "*info*" column family

| Row-Key | Column-Family:Column-Key | Timestamp | Cell Value |
|---|---|---|---|
| 1234 | info:fn | 1234564897 | John |
| 1234 | info:ln | 1234564897 | Smith |
| 5678 | info:fn | 1234567895 | Joyce |
| 5678 | info:ln | 1234567895 | Cooper |

- "*pwd*" column family

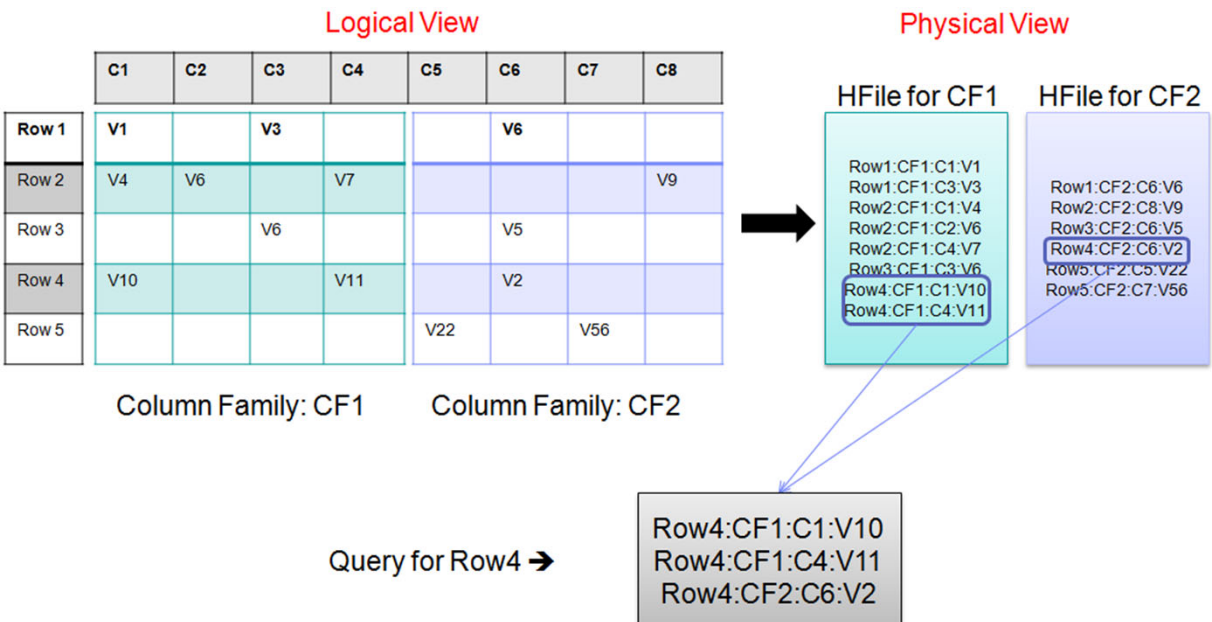
| Row-Key | Column-Family:Column-Key | Timestamp | Cell Value |
|---|---|---|---|
| 1234 | pwd:p | 1234564897 | Hello, world! |
| 5678 | pwd:p | 1234567895 | wysiwyg |
| 5678 | pwd:p | 1234567895 | wisiwig |

KEY (Row-Key, CF, Qualifier, Timestamp)

VALUE

# Logical to physical view





# HBase Components



- Region
  - The rows of a table are stored within Region
  - A table's data is automatically sharded across multiple regions when the data gets too large.
  - Each region stores a single Column Family.
- Region Server
  - Contains one or more Regions
  - Hosts the tables, perform read/writes, buffers
  - Client talks directly to the Region Server for their data.
- Master
  - Coordinating the Region Servers
  - Detects status and load rebalancing of the Region Servers
  - Assigns Regions to Region Servers
  - Multiple Masters are allowed, but only serves as backups
  - Not part of the read/write path
  - Highly available with ZooKeeper

# HBase Components Definitions and Roles



- ZooKeeper
  - Critical component for HBase
  - Ensures one Master is running
  - Registers Region and Region server
  - Integral part of the fault tolerance on HBase
- HDFS
  - The Hadoop filesystem
- API
  - The JAVA client API

# Characteristics of HBase tables



- HBase tables consists of rows.
  - Each row must have a unique row key
- HBase rows are formed by one or more columns
- Columns can have multiple timestamps to indicate the version.
  - The most recent version of the data is shown first
- Columns are grouped into Column Families which must be defined up front
- Column-Families contains columns
  - From our example earlier: *info:l*
  - *info* is the column family
  - *l* is the column qualifier or column key of the last name
  - Column → Column Family + Qualifier
- Columns can be added on the fly
- Columns can be NULL
  - NULL columns are not stored and free of costs
  - Making HBase sparse.
- Cell → (Row Key, CF, CQ, Version)

# HBase is a sorted multidimensional map



- You should think of HBase as a map.
- Each row (key) maps to some data (value).
- The value is contained in some column that belongs to a column family.

# Row key design considerations



- Having an efficient HBase system is highly dependent on how your choose your row key, or primary key (for those still in the RDBMS mindset)
- Depending on how your data is accessed, you will need to design your row key accordingly
- Some examples:
  - Lots of writes to your table: random row keys → good for performance
  - Lots of reads to your table: sequential row keys → good for performance
  - Time series data like log files: sequential row keys → good for scans of specific time
- This is not a trivial task and can be considered an advance topic, but it's good to keep this on the back of your mind as you learn about HBase
- Choose keys that can allow your data to be distributed as evenly as you can for your usage

# Column family design considerations



- Another important, non-trivial, task to keep in mind is the column family design.
- With traditional RDBMS, if you want to get all the data pertaining to a user, for example, from the tables, user, user_basic_info, user_miscellanous_info, etc., you will need to join them all together using foreign keys.
- With HBase, you just need to specify the column family of user and it would be able to immediately get you want you need.
- Each region contains a single column family.
- You specify the column families for columns that have similarities. If you store data across different column families that are related, you'll end up having to query multiple regions just go get your data, defeating the purpose of HBase.
- Column families can have their own compression methods, memory and resource allocation, etc.
- Column families are defined up-front at table creation time.
- Typically want no more than 2-3 column families per table.

# Cluster Configuration



- HDFS settings can be found in a set of XML files in the configuration directory Hadoop; conf/ under the main installation directory Hadoop (where you unzipped Hadoop).
- The conf/Hadoop-defaults.xml contains the default values of all parameters of Hadoop.
- Configuration options are a set of key-value pairs in the format:

```
<Property>
    <Name></name>
    <Value>property-value< /value>
 < /Property>
```

- Add the line <Last>true< /end> within the body of property, to avoid the properties to be overridden by user applications. This is useful for most configuration options.

# HDFS Configurations settings



| Key | Value | Example |
|---|---|---|
| Dfs.data.dir. | *Path* | /Home/*user/data*/hdfs |
| Dfs.name.dir | *Path* | /Home/username /hdfs/name |
| Fs.default.name | *Protocol*:// servername*:port* | HDFS: //mike.alpha.org:9000 |

# Hadoop site.xml for a Single-node Configuration



- <Configuration>
  - <Properties>
  - <Name>fs.default.name</name>
  - <Value>hdfs://your.server.name.com:9000 < /value>
  - < /Property>
  - <Properties>
  - <Name>dfs.data.dir< /name>
    - <Value> /home/ username /hdfs/data< /value>
  - < /Property>
  - <Properties>
  - <Name>dfs.name.dir</name>
  - <Value> /home/ username /hdfs/name< /value>
  - < /Property>
- < /Configuration>

# HDFS Start



- Formatting the file system you have just set up:
- User@namenode:Hadoop$ bin/Hadoop namenode –format
- This process is performed only once. When it is completed, we are free to start the distributed file system:
- User@namenode:Hadoop$ bin/start-dfs.sh
- On the master machine, name node server will be prompted to start by this command.
- It also begins the DataNode instances in each of the slaves
- In a single machine "cluster", this is the same computer as the NameNode instance.
- this script will be converted from ssh to eac on the cluster machine.

# Interact with HDFS



- Interacting with the HDFS, loading and recovery of data, manipulate files are done using commands. Communication with the cluster are performed by a monolithic script called bin /Hadoop .Hadoop system can be loaded with the help of Java virtual machine and then we will execute a user command
- Commands are specified in the following way:
  - User@machine:Hadoop$ bin/Hadoop moduleName -cmd arguments...
  - The moduleName tells the program what subset of Hadoop functionality to use
  - -Cmd is a specific command to run the module. Their arguments are the name of the command
- Two of these modules are relevant to the HDFS: dfs and dfsadmin.

**Insertion of the Data in the Cluster**

- Create the home directory if it does not already exist
  - mike@somenode:Hadoop$ bin/Hadoop dfs -mkdir /user)If there is no /user directory in the first place, it will automatically be created later if necessary)
- Upload a file. To load single file in HDFS, we can use the put command
  - mike@somenode:Hadoop$ bin/Hadoop dfs -putting /home/someone/myFile.txt /user/yourusername/
  - This copy /home/someone/myFile.txt from the local file system in /user/yourusername/myFile.txt HDFS
- Check if the file is in HDFS using command
  - mike@somenode:Hadoop$ bin/Hadoop dfs -ls /user/yourusername
  - mike@somenode:Hadoop$ bin/Hadoop dfs -ls

# Examples for put command



| Command: | Assuming: | Result: |
|---|---|---|
| `Bin/Hadoop dfs -place foo bar` | No file/directory with the name/user/ `$USER/bar` exists in HDFS | Local Load file foo in a file called `/user/ $USER/bar` |
| `Bin/Hadoop dfs -place foo bar` | `/User/ $USER/bar` is a directory | Local Load file foo in a file called `/user/ $USER/bar/foo` |
| `Bin/Hadoop dfs -place foo somedir/somefile` | `/User/ $USER/somedir` does not exist in the HDFS | Local Load file foo in a file called `/user/ $USER/somedir/somefile`, the creation of the directory lack |
| `Bin/Hadoop dfs -place foo bar` | `/User/ $USER/bar` is already a file in HDFS | No change in the HDFS and an error is returned to the user. |

# Retrieve Data from HDFS



- Data visualization with the cat command

- Assume that a file with the name "foo" has been loaded into your directory "home" in HDFS
  - mike@somenode:Hadoop$ bin/Hadoop dfs -cat foo
  - (The contents of foo shown here)
  - …….
  - …….

  urn off the HDFS

- Switch off the HDFS cluster functionality, then this can be done by logging in to the NameNode machine and running:
  - mike@somenode:Hadoop$ bin/stop-dfs.sh
- .

# HDFS Command Reference



| Command | Operation |
|---|---|
| -Ls Path | Displays the contents of a directory specified in the path, which lists the names, permissions, owner, size and modification date of each entry. |
| -LSR *way* | It behaves like -ls, but displays the entries of recursively all subdirectories of the path. |
| -Du *path* | Displays disk usage, in bytes, of all the files that match the path and file names are reported with the prefix HDFS protocol. |
| -Dus *way* | As -du, but prints a summary of the disk usage of all the files and directories in the path. |
| -Mv *src dest* | Moves the file or directory indicated by *src* to dest, in HDFS. |
| -Cp *src dest* | Copy the file or directory identified by *src* to dest, in HDFS. |
| -Rm *way* | Deletes the file or directory gap identified by path. |
| -RMR *way* | Deletes the file or directory identified by path. Recursively deletes all entries secondary (i.e., files or subdirectories of the path). |

# HDFS Command Reference (Cont.)



| | |
|---|---|
| -Cat filename | Displays the contents of the *file to stdout.* |
| -Mkdir *path* | Creates a directory named *path in* HDFS. Creates the directories on the main road, which are missing (e.g., such as `mkdir -p` in Linux). |
| -Touchz *way* | Creates a file in the path that contains the current time as a timestamp. Don't know if a file already exists in the path, unless the file is already size 0. |
| -CHMOD [ -R]*mode,mode, ... the path ...* | Change the file permissions associated with one or more objects that are identified by path.... Performs the changes recursively with `-R, it is a` mode of 3 digit octal mode or {augo`} + / - {rwxX}.` If it does not assume a specified range and do not apply a umask. |

# HDFS Command Reference (Cont.)



| | |
|---|---|
| -Chgrp [ -R]*group ...* | Set the group ownership of files or directories identified by path.... Group recursively sets if you specify -R. |
| -Help *cmd* | Returns information about the use of one of the commands listed above. You have to omit the main character " -" in cmd |
| -Setrep [ -R] [ -w]*rep way* | Sets the objective of replication of files that have been identified in the path to rep (The actual replication factor will move toward the goal at the time) |
| -Chown [ -R] [*owner*] [ : ]] [ group *path ...* | Sets the user owner and/or a group of files or directories identified by path.... Joint owner recursively if -R is specified. |

# HDFS Permissions and Security



- HDFS file permissions based on permission system the POSIX model
- The HDFS permissions system is designed to prevent accidental corruption of data or improper use of informal information within a group of users that share access to a group
- Security of HDFS is based on the POSIX model of users and groups.
- In POSIX model three type of permissions are given to the user-read, write and execute.
- It is associated with three different granularities: the owner of the file, users in the same group as the owner, and the rest of the users on your system.
- As the HDFS does not provide all the POSIX spectrum of activity, some combinations of bits would not make sense.
- For example, files cannot be executed; the +x bit cannot be set on files (only the directories). Even though the +w bits are enabled, the files cannot be written to.
- Security permissions and ownership can be modified using the bin/Hadoop dfs -chmod, -chown and chgrp operations described earlier in this document, which work in a similar fashion to the POSIX/Linux tools of the same name.

# HDFS Additional Tasks- Rebalancing Blocks



- New nodes can be added to a cluster in a simple way
- In the new node, the same Hadoop version and configuration (conf/Hadoop-site.xml) as in the rest of the group must be installed
- Starting the DataNode daemon on the machine causes the contact with the NameNode and it joins the cluster
- However, the new DataNode does not have the data at the first moment; therefore, it is not alleviating space problems in the existing nodes
- New files will be stored in the new DataNode in addition to the already existing, but for optimum usage, storage should be balanced in all nodes.
- This can be achieved with the automatic balancing tool included with Hadoop. Balancer helps in balancing the smart block thus creating uniform distribution of blocks with little threshold. (The default value is 10 %.) Smaller percentages has the nodes more balanced, but may require more time to reach this state. Perfect balance (0 %) is unlikely to be reached.
- New nodes can be added to a cluster in a simple way.
- Starting the DataNode daemon on the machine causes the contact with the NameNode and it joins the cluster.

# Closing The Nodes



•The nodes can be removed from the cluster while it is running, without data loss

•The nodes must be removed with a calendar that allows for HDFS to ensure they are not fully replicated in those who have retired DataNodes.

•In addition to allowing the nodes that are added to the cluster on the fly, the nodes can be removed from the cluster while it is running, without data loss. But, if the nodes are simply closed down "hard", it can cause a loss of data, it is possible to maintain the only copy of one or more blocks of files.

•HDFS feature provides a closure that ensures that this process is performed safely. To use this feature, follow the steps below:

- 
- •Configuring the cluster
- • Determine the hosts to remove
- • Force configuration reload
- • Turn off the nodes.
- • Edit excludes file again

# Check Health File System



- Hadoop provides a command fsck to make sure that the file system is healthy.
- Hadoop provides a **command fsck** to do exactly this. It can be launched to the command line as follows:
  - Bin/Hadoop fsck [track] [ options ]
  - If we run without arguments, it prints current usage information and exit.
  - If we run with the argument, the health state of the entire file system is checked and printed as a report.
  - If we have a trajectory of a particular directory or file, just check the archives of that route.
  - If an option argument is given but there is no path, you will start from the file system root.
- Options may include two different types of options:
- **Options for action** specify what action should be carried out when there are damaged files. This can be -move, which moves corrupt files in / lost+found, or - delete, that eliminates damaged files.
- **Detailed Information** options specify how the tool must be in your report. The option of files will display a list of all the files, make sure that it does. This information can be extended by adding the option -blocks, which prints the list of blocks for each file.

# Load scenarios



Data at rest

Data in motion

Streaming data

Data from a web server


Data from a data warehouse

# Load solution using Flume

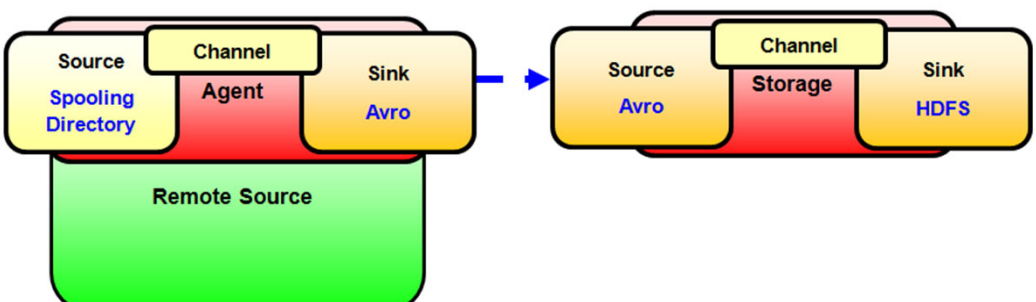


- **Flume –** is a distributed service for collecting data and persisting it in a centralized store
- - Can be use to collect data from sources and transfer to other agents
- - A number of supported sources
- - A number of supported sinks
- - Flume agents are placed in source and target locations

# How Flume works

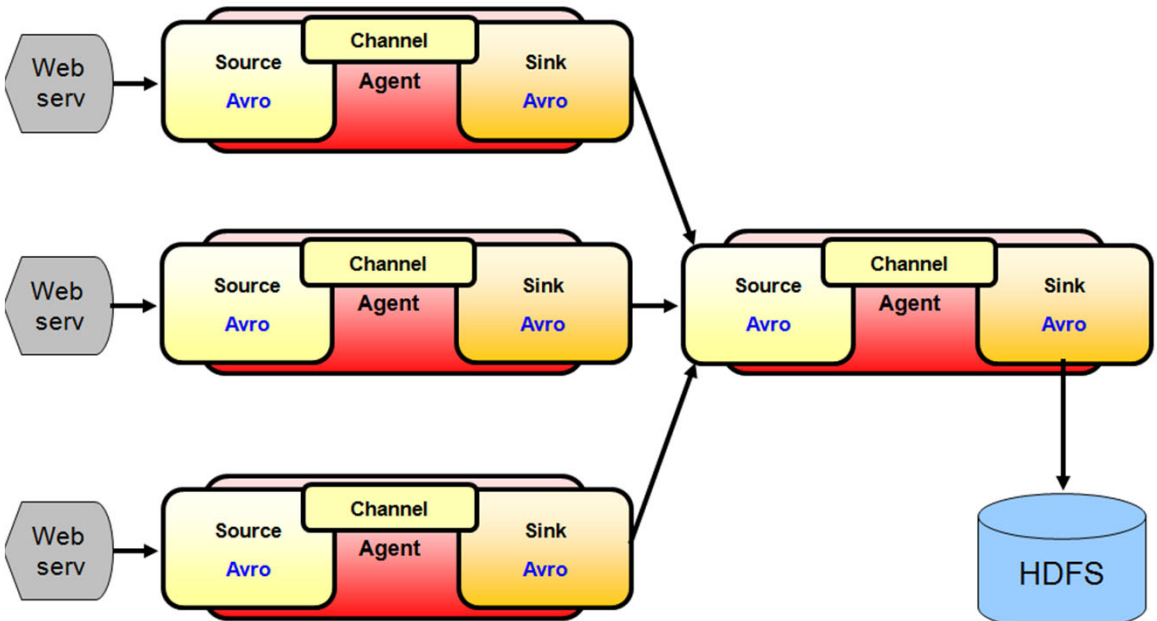


- It is built on the concept of flows
- Flows might have different batching or reliability setup
- Flows are comprised of nodes chained together
- Each node receives data as “source”, stores it in a channel, and sends it via a “sink”



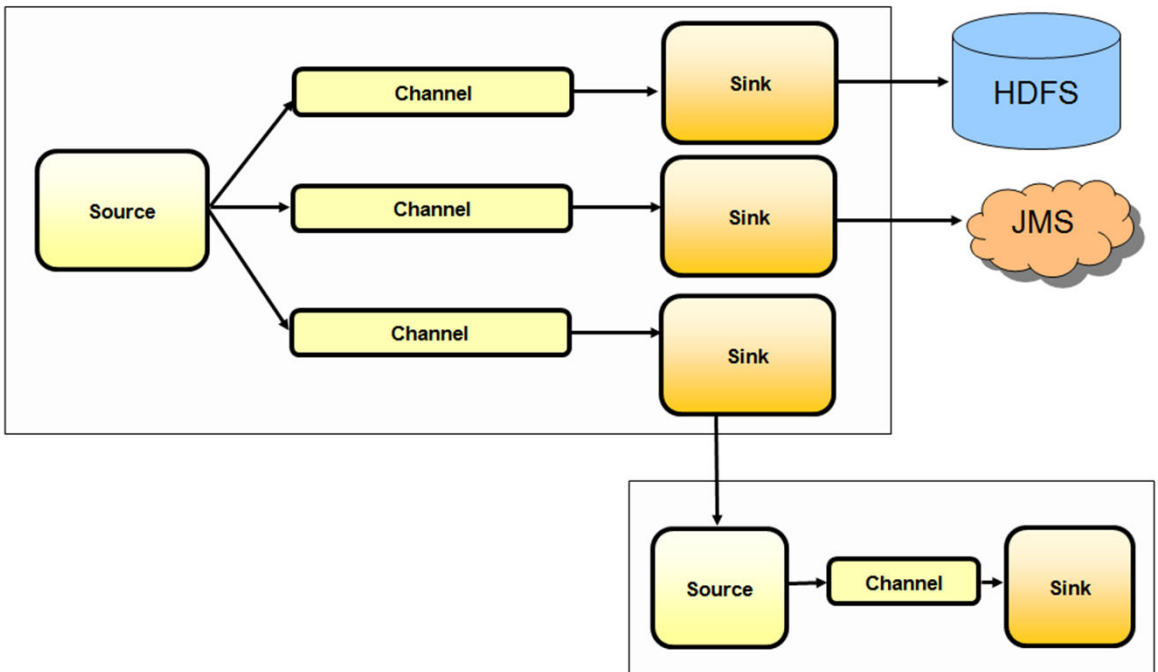
# Consolidation





# Replicating and multiplexing





# Checkpoint (1 of 4)



1. Apache Hadoop is a free opensource framework for
   - A. Data Intensive Application
   - B. Web Search Technology
   - C. Read Intensive Application
   - D. All the above

2. PIG is a high level language that generates
   - A. Map
   - B. Functions
   - C. Data sets
   - D. All the above

# Checkpoint solution (1 of 4)



1. Apache Hadoop is a free opensource framework for
   - A. Data Intensive Application
   - B. Web Search Technology
   - C. Read Intensive Application
   - **D. All the above**

2. PIG is a high level language that generates
   - **A. Map**
   - B. Functions
   - C. Data sets
   - D. All the above

# Checkpoint (2 of 4)



3. Parallel data processing supports the following:

   A. Grid Computing
   B. Workload balancing
   C. Parallel databases
   D. Option A and C

4. Hadoop Distributed File System (HDFS) was created by

   A. Google
   B. Yahoo
   C. FB
   D. None of the above

# Checkpoint solution (2 of 4)



3. Parallel data processing supports the following:

A. Grid Computing
B. Workload balancing
C. Parallel databases
**D. Option A and C**

4. Hadoop Distributed File System (HDFS) was created by

A. Google
**B. Yahoo**
C. FB
D. None of the above

# Checkpoint (3 of 4)



5. Hadoop Distributed File System(HDFS) was created for

   A. Processing small files
   B. Processing large files
   C. Scaling the data
   D. None of the above

6. This feature is not the design principle of Hadoop

   A. Failures
   B. Scalability
   C. Workload balancing
   D. Distribute Data and Utilitize Paralellism

# Checkpoint solution (3 of 4)



5. Hadoop Distributed File System(HDFS) was created for

A. Processing small files

**B. Processing large files**

**C. Scaling the data**

D. None of the above

6. This feature is not the design principle of Hadoop

A. Failures

B. Scalability

**C. Workload balancing**

D. Distribute Data and Utilitize Paralellism

# Checkpoint (4 of 4)



7. touchz command is used to

A. create a file with zero length
B. create a file with zero byte
C. create a read only file
D. Both A and B

8. This is the way to process large data sets by distributing the work across large number of nodes.

A. Maps
B. MapReduce
C. MapDeduct
D. ReduceMap

# Checkpoint solution (4 of 4)



7. touchz command is used to

A. create a file with zero length

B. create a file with zero byte

C. create a read only file

**D. Both A and B**

8. This is the way to process large data sets by distributing the work across large number of nodes.

A. Maps

**B. MapReduce**

C. MapDeduct

D. ReduceMap

# Unit summary



Having completed this unit, you should be able to:

- Define Hadoop concept and Hadoop File System (HDFS)
- Administer HDFS on server system
- Define the concept of Map / Reduce
- Setup an Hadoop Cluster
- Manage Job Execution using HDFS
- Understand JAQL
- Load the data
- Overview the workflow engine