Cloudera Certification Notes

Table of Contents

Hadoop 3

HDFS 4

HDFS Architecture 6

HDFS Data Flow 8

Hadoop Archive Files (HAR) 8

Hadoop Commands 9

MapReduce 10

Classic MapReduce vs. YARN 15

MapReduce Architectures 16

Job Failures 19

Writing MapReduce Jobs 21

Combiners and Partitioners 28

Input/Output Format Classes 32

Old vs. New Java APIs 34

Advanced MapReduce Topics 35

Debugging MapReduce Jobs 35

Logging 37

Unit testing with MRUnit 39

Counters 41

Serialization 42

File Compression 43

Algorithms 45

Sorting 45

Searching 48

Indexing 49

Joins 50

Hadoop Ecosystem 51

Sqoop 51

Pig 53

Pig Latin 54

Hive 60

Flume 64

Impala 65

Oozie 67

Appendix 1: MasterPeace Lab 68

Logging In 68

To add a local file to HDFS 68

Appendix 2: Running MapReduce Jobs 69

Appendix 3: Setting Up A MapReduce Project 70

Appendix 4: Hadoop Streaming and Hadoop Pipes 71

# Hadoop

1. Distributed computing
   1. Motivation for distributed processing
      1. When handling extremely large files, not all the data will fit into memory
      2. Processing large amounts of data may take too long on a traditional system
      3. Handling huge amounts of data may not be cost effective
         1. Memory may not scale
         2. Disk storage systems may no longer be able to keep up at a reasonable cost
   2. Distributed systems
      1. Use multiple machines for single job
      2. Job is divided among the machines
      3. Job is moved to the data
      4. Pieces of the job are run in parallel
   3. Challenges
      1. Programming jobs can be complex
      2. Difficult to keep data and processes in synchronization
      3. Finite bandwidth sharing data and process info across nodes
      4. How does one deal with job failure or hardware failure?
2. The Hadoop framework solves the aforementioned challenges and hides the plumbing so that developers can focus on the analytics.
3. Radical new approach to distributed computing:
   1. Distribute data when the data is stored
   2. Run computation where the data is located
4. Originally based upon work done at Google
5. Open-source project overseen by Apache Software Foundation
6. Terminology
   1. Cluster - a group of computers working together
   2. Node – an individual computer in the cluster
      1. Master nodes control distribution of work and data to the slave nodes
      2. SR
   3. Daemon - a program running on a node
7. Core concepts
   1. Computation is based upon the Map-Reduce programming approach
   2. Nodes talk to each other as little as possible
   3. Data is distributed in advance when it is ingested
   4. Hadoop system scales to “Big Data” - Petabytes
   5. Hadoop system is fault-tolerant
      1. If job fails on one node, it is started again on another node
      2. One or more nodes fail → system not affected
8. Scalability
   1. Adding nodes adds capacity proportionally
   2. Increasing the load results in a graceful decline in performance, not failure of the entire system
9. Fault-tolerance
   1. What happens if a job on a node fails?
      1. The job manager assigns the job to run on a different node
      2. Original job is terminated
   2. What happens if one or more nodes fail due to a hardware/software problem?
      1. System continues to function
      2. Master re-assigns tasks to a different node
      3. Data replication ensures no loss of data
      4. Nodes which recover rejoin the cluster automatically
10. Use Cases
    1. Extremely large amounts of data
    2. Analysis or processing can be done in batch
    3. Analytic algorithm can be performed within the Map-Reduce framework
    4. All or most of the data must be scanned
    5. Data can be written one time and then read / analyzed multiple times
    6. Data typically unstructured - no set schema
11. Architecture of a Hadoop Cluster[[1]](#footnote-1)
    1. Hadoop Cluster: a group of machines working together to store and process data
    2. Two Master nodes
       1. NameNode
          1. Manages HDFS
          2. Knows where the data lives, on which node.
          3. The NameNode daemon has to be running for Hadoop to work.
             1. Must be running at all times.
             2. If the NameNode stops, then the cluster becomes inaccessible
          4. High Availability Mode
             1. Available in CDH4 and later
             2. Two NameNodes - one active and one on Standby
          5. Classic Mode
             1. One NameNode
             2. One “helper” node called the Secondary NameNode
       2. Stand-by NameNode
    3. Any number of Slave nodes
       1. Use HDFS to store data
       2. These nodes store data, thus are called “data nodes.”
       3. Run MapReduce jobs

## HDFS

1. HDFS Concepts
   1. HDFS stands for Hadoop Distributed File System.
   2. HDFS is a filesystem written in Java
   3. It is a distributed filesystem, meaning that the files are stored across multiple computers.
   4. Sits on top of native filesystem
   5. Files are broken up by HDFS into smaller chunks called “blocks.”
   6. Provides redundant storage for massive amounts of data.
      1. This replication enables the cluster to preserve the data and continue its work even if some nodes fail.
      2. With a large number of computers involved, hardware failures become more likely.
   7. Files in HDFS are “write-once.”
      1. Files can be deleted but not updated.
      2. However, appends are supported.
   8. Ways to Access HDFS
      1. Shell: hadoop fs
      2. Java API
      3. EcoSystem Projects: Flume, Sqoop, Hue
   9. To work with data, you have to load it into HDFS:  
      hadoop fs -put <source file> <destination file>
      1. This will copy the file to /user/<username>/<destination file>
      2. E.g., for the user pford, hadoop fs –put foo.txt foo-hdfs.txt copies foo.txt to /user/pford/foo-hdfs.txt
   10. More HDFS examples
       1. To get a directory listing of the user’s home directory in HDFS: hadoop fs –ls
       2. To get a directory listing of HDFS root: hadoop fs –ls /
       3. Display contents of HDFS file: hadoop fs -cat /user/username/bar.txt
       4. Copy a file to local disk from HDFS:  
          hadoop fs -get /user/<username>/<hdfs file> <local file>
       5. Create a directory called “input” in the user’s home directory:  
          hadoop fs -mkdir input
   11. HDFS performs best with a relatively modest number of large files.
       1. Why? …. The NameNode stores file meta-data in memory.
       2. Thus, millions of files is better than billions.
       3. Files are typically 100MB or more in size.
       4. HDFS is designed for very large files and won’t work efficiently with lots of small files.
       5. HDFS is not designed for quick reads of data, nor for large amounts of small files.
          1. It is not a good choice for an online transaction-processing database (OLTP).
          2. It would not be a good choice for an online shopping cart.
       6. Doesn’t support arbitrary file modifications. Only appends are supported.
   12. HDFS is optimized for large, streaming reads of files
   13. How HDFS stores files
       1. Data files are split into blocks and distributed at load time.
       2. Each block is replicated on multiple data nodes (default 3X).
          1. This value is set in the dfs.replication property, found in the hdfs-site.xml configuration file in Hadoop.
          2. The value can be changed in hdfs-site.xml.
       3. The NameNode stores metadata about the files in memory.  
          
   14. Block Size
       1. The minimum amount of data that can be read/written in a file system.
       2. The default size is 64MB – which is very big for a filesystem.
          1. The usual on a disk is 512 bytes.
          2. The HDFS block size so large to reduce the seek overhead to around 1% of the transfer time. (HDFS seek times are very slow.)
       3. A file smaller than the block size does not occupy a full block in HDFS.
       4. If you are trying to copy a file to HDFS but cannot because the data nodes are almost full, but they still have enough space to store your file, what would you do? Decrease the block size of the files you are copying.

### HDFS Architecture

1. Follows master-slave architecture.
   1. The NameNode is the master.
   2. The data nodes are the worker/slave nodes.
2. NameNode maintains the Namespace Image and Edit Log.
   1. It holds them in main memory.
   2. The NameNode is a Java program, nothing more.
   3. The Namespace Image and Edit Log form the complete file system image.
   4. If the NameNode fails, the file system goes down.
      1. Thus, the NameNode hardware should be resilient and should be higher-grade hardware.
      2. The NameNode is the Single Point of Failure in HDFS.
   5. Hardware recommendations for the NameNode:
      1. Storage: Raid
      2. RAM: 32GB and up
      3. Processor Cores: 16 and up.
      4. Networking: multiple ports/10 GB bandwidth to switch.
   6. Requires a lot of memory – 1000 MB per million storage blocks is recommended.
   7. Responsible for updating the Namespace Image and Edit Log to local disk and to a network storage location.
3. Secondary NameNode
   1. Merges the Edit Log and Namespace Image regularly, so that the NameNode does not run out of memory, and creates checkpoints.
      1. These are the NameNode’s memory.
      2. That is its only purpose.
      3. Thus, it frees on memory on the NameNode.
   2. It is a Java program.
   3. It is processing-intensive and requires good hardware, should be the equal of the NameNode.
   4. It does **not** take over as NameNode in case of failure, but it does label the machine that is the next best choice for a NameNode, should the NameNode fail.
4. HDFS Federation
   1. Added in Hadoop 2.3
   2. Uses 2 NameNodes to reduce the load on any one NameNode: NameNode memory becomes a limiting factor once the cluster reaches around 2000 nodes.
5. High Availability
   1. A new feature as of Hadoop 2.3.
   2. Again uses a second NameNode, as a stand-by in case the first NameNode fails.
   3. In case of failure, the reserve NameNode takes over.
   4. The two NameNodes share the namespace and edit logs through shared storage.

### HDFS Data Flow

1. Writes
   1. HDFS client makes a write request to the NameNode.
   2. The NameNode performs various checks, such as authorizing the client.
   3. IF all is well, the NameNode then sends back a list of Nodes to the client.
   4. The client writes blocks to the three nodes in the pipeline (3 being the default number for data replication).
   5. The data nodes send acknowledgements back through the pipeline to the client.
   6. Steps 3-5 are repeated until the data is all written.
   7. In case of Data Node failure:
      1. The data is written to the two remaining nodes.
      2. The NameNode notices the under-replication and arranges for more replication.
      3. The same happens if multiple data notes fail.
   8. The whole process of selecting nodes for replication happens behind the scenes, and the programmer and client don’t have to worry about.
   9. Node Distance
      1. Calculated is based on bandwidth because bandwidth is so precious.
      2. Blocks on the same data node have distance 0.
      3. Block on a different node on the same rack have distance 2
      4. Blocks on different racks in the same data center have distance 4.
      5. Blocks on racks in different data center have distance 6.
2. Reads
   1. The client receives a list of 3 data nodes from the NameNode.
   2. The client contacts the nearest node.
   3. If the nearest node fails, the client proceeds to the next closest node.

### Hadoop Archive Files (HAR)

1. The problem with small files is that the NameNode has to devote memory to holding their metadata, so NameNode memory becomes a limiting factor.
2. Hadoop Archive Files (HAR) can help deal with small files.
   1. They are used to concatenate a number of files.
   2. To create one:   
      hadoop archive –archiveName <archive name> -p <parent> [-r <replication factor>] <src> <dest>
3. Limitations
   1. Don’t support compression, so they are like a duplicate file.
   2. They are immutable.

### Hadoop Commands

1. dsitcp
   1. Used to parallel-copy data in large amounts from one HDFS to another.
   2. Used in the context of HDFS federations.
   3. Syntax:  
      hadoop distcp hdfs://<source url> hdfs://<destination url>

# MapReduce

1. The test is mostly about MapReduce.
2. MapReduce is a method for distributing a computational task across multiple nodes.
   1. MapReduce breaks complex tasks down into smaller elements which can run in parallel
   2. MapReduce programs can be written in Java, C++, Python, or Ruby.
   3. MapReduce programs are inherently parallel; thus they put large-scale data analysis in the hands of anyone with enough computers at his/her disposal.
3. Each node processes the data stored on that node (when possible)
4. Consists of two major phases
   1. Phases
      1. The “map” - maps keys to values
      2. The “reduce” - reduces all values mapped to the same key
   2. Each phase has key/value pairs as input and output.
   3. The programmer writes a function for each phase: the **map** function and the **reduce** function.
      1. The input for the mapper function is the raw data.
      2. After the mapper completes, the MapReduce framework performs processing on the output (sorting and grouping the output by key) before passing it to the reducer.
      3. To write the map function, extend Hadoop’s Mapper class (an abstract class) and override the map method.
5. Features
   1. Automatic parallelization and distribution of jobs across the cluster
   2. Fault-tolerance
   3. A clean abstraction for programmers
      1. Jobs are usually written in Java
      2. Jobs can also be written in scripting languages using *Hadoop Streaming*
   4. MapReduce abstracts all of the housekeeping away from the developer
6. Terminology
   1. **Job** - a complete execution of Mappers and Reducers over a dataset
   2. **Task** - the execution of a single Mapper or Reducer over a slice of data
   3. **Speculative execution** - when Hadoop schedules another copy of a mapper task in order to speed up a slow running mapper.
   4. **Task Attempt** - particular instance of attempt to execute a task
      1. There will be at least as many task attempts as there are tasks
      2. When a task attempt fails, another is started either by the JobTracker (v1) or by the ApplicationMaster (v2)
      3. Speculative execution can result in more task attempts than completed tasks
7. Stages
   1. Mapper
      1. Mapper takes as input a key/value pair and will output a list of zero or more “intermediate” key/value pairs.
      2. Each map task (typically) operates on a single HDFS block
      3. Map tasks (usually) run on the node where the block is stored
      4. Mapper task intermediate data is stored on local disk, not HDFS.
   2. Shuffle and Sort  
        
        
      1. Sorts and consolidates intermediate data from all the mappers
      2. Every MR job goes through this phase.
      3. Happens after all the map tasks are complete and before the Reduce starts
      4. Intermediate keys are grouped together
      5. Each key and value list is pulled by reducer from the intermediate data zone
         1. All values for intermediate key go to same reducer
         2. The intermediate keys/value lists are passed in sorted order
      6. This phase produces the list of key/list-of-values pairs that is sent to the reducer.
         1. It sorts the keys.
         2. It combines all of the values received from the mappers for each key into one list of values for each key.
         3. Prior to this phase, the results from the mappers were on value for each key, and the keys were not unique.
      7. During the shuffle part of this phase, the mapper writes results to an in-memory buffer.
         1. One the buffer size crosses a threshold, its contents are written to local disk.
         2. The size of the buffer and threshold are configurable with properties.
      8. Combiners run in this phase, before the sort.
         1. They may run multiple times.
         2. Thus, running them multiple times must not affect the result.
         3. When a lot of intermediate data is produced, both disk I/O and network I/O can become bottlenecks.
      9. Partitioners also run in this phase before the reduce phase: the determine which reducers to send intermediate data to.
   3. Reducer
      1. Reducer outputs zero or more final key/value pairs for each input key - written to HDFS.
      2. Pulls shuffled/sorted intermediate data from Map tasks
      3. Operates on this data to reduce to one value per key.
      4. Produces final output, which it writes to HDFS.
      5. In the reduce phase, all items with the same key (from the various mappers) will be processed by the same reducer, guaranteed. Important to know for certain algorithms.
      6. Uses
         1. Summation
         2. Mat functions like max/min, avg.
         3. Collating
         4. Sorting
8. Mappers
   1. Mapper takes as input a key/value pair and will output a list of zero or more key value pairs.  
        
      
   2. The input key is, by default, the byte offset of a line from a file; the value is the line.
   3. The mapper does not have to use the input key.
      1. Use of the key is optional
      2. Often the mapper’s input key is irrelevant.
   4. If the mapper writes anything out, then the output must be in the form of key/value pairs.
   5. In the resulting set of key/value pairs, the keys are **not** unique.
      1. At this point, you have one value for each key.
      2. The shuffle-and-sort phase, produces the sorted list of key/list-of-values pairs that is sent to the reducer.
   6. The number of mappers created by Hadoop equals the number of input splits calculated by Hadoop.[[2]](#footnote-2)
      1. An input split is….
      2. So if the block size for a cluster is 64MB, and you are processing a directory of 100 files, each 100MB in size, then….
         1. Two blocks are needed for each file, and each file produces two input splits.
         2. Thus, for 100 such files, Hadoop will calculate that it requires 200 input splits.
         3. Thus, Hadoop will create 200 mappers for the job.
      3. The number of mappers used is controlled by the MapReduce framework based on the number of input splits—one mapper per input split.
   7. You get a separate JVM for each mapper.
   8. Common Uses for Mappers
      1. Data Translation
         1. Convert input into upper case
         2. Convert text language
      2. Data “Explosion”
         1. Read in line of text but write out each instance of a word
         2. Useful in counting and sorting tasks
      3. Data Filtration
         1. Only output key/value pairs where some condition is true
         2. Useful for filtering type of data
         3. For example, getting temperature where it is greater than threshold.
      4. Changing Keyspaces - key mapped to lookup key
9. Reducers  
     
     
   1. Reducer outputs zero or more final key/value pairs for each intermediate input key - written to HDFS.
   2. Takes an intermediate key, with all of the intermediate values for that key, as input.
      1. These key/list-of-values pairs come to the reducer already sorted, by the shuffle-and-sort.
      2. The last of values is also collated by the shuffle-and-short.
   3. Common uses for Reducers
      1. Summation
      2. Math functions: max/min, avg.
      3. Identity – dump out each value with the key
      4. Collating – group values by key
      5. Sorting
   4. The number of reducers used is controlled by the programmer.
   5. How many reducers are needed?
      1. It is important to consider how many reducers should be specified when configuring the job.
      2. The default is a single reducer.
      3. With a single Reducer, one task receives all the keys in sorted order.
         1. This can be advantageous if the output must be in sorted order.
         2. However, this can cause performance issues if there is a large amount of intermediate data.
            1. In that case, the node running the reducer may not have sufficient disk space
            2. Thus, the job may take a long time to run.
      4. Some jobs may require a specific number of reducers, for example:
         1. Batching output by month [0..11] - specify 12 reducers.
         2. Batching output by day or week [0..6] - specify 7 reducers.
         3. Batching messages by hour of the day [0..23] - specify 24 reducers.
      5. Many jobs can be run with a variable number of reducers.
         1. Each reducer should get a reasonable amount of intermediate data, but not too much.
         2. The developer must decide how many to specify by:
            1. Testing the job first with a relatively small test data set,
            2. Then extrapolating to calculate the amount of intermediate data expected from the ‘real’ input data.
            3. Then using that result to calculate the number of reducers that should be specified.
      6. You should take into account the number of Reduce slots likely to be available on the cluster.
         1. If your job requires one more Reduce slot than there are available, then a second wave of Reducers will have to run.
         2. In this way, while increasing the number of Reducers may cut down the time spent for each Reducer, the second wave may add more overall time to the job.
10. Job data
    1. Mapper Data Locality Preference
       1. When possible, Hadoop will attempt to run map tasks on the node where the block of data is stored locally.
       2. If this is not possible, the map task will transfer the data across the network as it processes that data.
    2. Where does the job data live?
       1. Mapper task intermediate data is stored on the local disk (not HDFS)
       2. What about reducers?
          1. There is no concept of data locality for reducers.
          2. Reducers pull the mapper intermediate data across the network.
          3. Reducers write their output to HDFS.
          4. The reduce method won’t start until all intermediate data has been transferred.
          5. However, Hadoop will start to allow reducers to pull intermediate data as soon as the mappers finish their work.
11. Speculative Execution
    1. It is possible for one map task to run more slowly than others:
       1. Faulty hardware
       2. Slow machine
    2. This would create a bottleneck since reduce method won’t start until all mappers have completed.
    3. Hadoop uses ***speculative execution*** to mitigate this condition.
       1. If a mapper appears to be running much more slowly than others, a new instance of the mapper will be started on another machine, operating on the same data (new task attempt for same task)
       2. The results of the first mapper to finish will be used.
       3. Hadoop will kill off the mapper that is still running
    4. Launched only after all of the tasks are launched.
    5. Enabled for mappers and reducers with properties.
       1. mapred.amp.tasks.speculative.execution.
       2. mapred.reduce.tasks.speculative.execution
       3. Default is true.
       4. Setting them to true is recommended.
    6. This is a performance-tuning feature.
12. Further performance tuning features
    1. Task JVM reuse
    2. Skipping Bad Records.
       1. Skipping mode is turned on after 3 failed attempts.
          1. If a job fails because of a bad record, that record is communicated back to the JobTracker after 3 attempts.
          2. The reason is does not happen after the each failure is to reduce network traffic.
       2. The default max number of failed task attempts is 4, so to use this feature effectively, you must increate the map max attempts and reduce max attempts properties.

.

## Classic MapReduce vs. YARN

1. YARN is a step closer to realizing the vision in the original Google MapReduce papers.
2. For both, the Hadoop run mode (local, pseudo-distributed, or distributed) is set in ${HADOOP\_HOME}/conf/mapred-site.xml.
   1. For Classic MapReduce, the property that sets the mode is mapred.job.tracker.
      1. The default value is local, which runs on a single JVM.
      2. For pseudo-distributed or distributed modes, set host:port.
   2. In the new MapReduce, the mode is set in mapreduce.framework,name, and the value can be local, classic, or yarn.
3. MapReduce V1
   1. Uses a JobTracker / TaskTracker architecture
   2. One JobTracker per cluster - limits cluster size to about 4000 nodes
   3. Slots on slave nodes designated for Map or Reduce tasks.
   4. Employs a JobClient, which runs on the client, and which interacts with Hadoop.
      1. Must be in the machine that accesses or interacts with Hadoop.
      2. Not present in the new MapReduce.
   5. Has scalability problems when the cluster size grows to over 4000 nodes.
4. MapReduce V2
   1. Built on top of YARN – Yet Another Resource Negotiator.
   2. Uses ResourceManager / NodeManager architecture – scalability
      1. Thus, it splits the old JobTracker responsibilities in two.
      2. The ResourceManager handles job scheduling.
      3. The Application Master handles task monitoring.
   3. Node resources can be used for any type of task
   4. Improves cluster utilization
   5. Support for non-MapReduce jobs.
   6. Programs written in Classic MapReduce **do** run on YARN.
   7. Advantages
      1. Increased scalability dramatically.
      2. More than one distributed framework can co-exist on the same cluster.
      3. Better memory utilization with the concept of containers
         1. Similar to slots.
         2. Containers are more flexible.
5. Job Scheduling
   1. FIFO
      1. Not available in YARN.
   2. Capacity Scheduler
      1. SR
   3. Fair Scheduler

### MapReduce Architectures

1. MapReduce v1 Architecture  
   
   1. One JobTacker per cluster that manages MR jobs and distributes tasks to TaskTrackers
      1. The JobTracker and TaskTrackers are just Java programs.
      2. Communicates overall combined results to the client.
   2. TaskTracker - one per slave node that starts and monitors Map or Reduce tasks
      1. Each node has multiple slots for Mappers and Reducers.
      2. Sends regular reports to the JobTracker, which then updates the JobTracker, which updates the client.
      3. TaskTrackers have a fixed number of slots to run mappers and reducers.
      4. The number of tasks that run on a node is decided by its RAM and the number of CPU cores on the node.
2. Running a Job in a MapReduce v1 Cluster  
     
     
   1. JobClient
      1. Connects to JobTracker and gets a job ID from the JobTracker.
      2. Checks whether output has been created or not.
      3. Calculates the splits, copies resources to HDFS with high replication, and submits the job.
      4. Create an instance of JobSubmitter to submit the job, putting in the queue of JobTracker.
   2. Job initialization
      1. Creates one map task per split.
      2. Creates a Job object which encapsulates the tasks and does bookkeeping.
      3. The number of reducers is decided by the mapred.reduce.tasks property or by Job.setNumReduceTasks(), which is an instance method of Job.
      4. JobTracker runs setup and cleanup jobs on TaskTrackers.
   3. Task Assignment
      1. TaskTracker runs a simple loop that periodically sends heartbeats to the JobTracker.
      2. JobTracker receives status of TaskTracker.
   4. Task execution
      1. TaskTracker retrieves the JAR from HDFS.
      2. TaskTracker runs a new JVM for each task.
      3. TaskTracker updates the JobTracker about status regularly.
      4. TaskTracker runs cleanup after the reduce tasks run: cleans up intermediate data.
   5. Calculating progress
      1. Progress during mapping is calculated by input size and the amount processed.
      2. Reduce phase is trickier.
3. MapReduce v2 architecture  
     
     
   1. **Client**
      1. Submits the job
      2. Interacts with HDFS.
   2. **Resource Manager**
      1. 1 per cluster
      2. Starts ApplicatonMasters and allocates resources on slave nodes
      3. Divided into two parts
         1. Scheduler
         2. Application manager
   3. **Application Master**
      1. 1 per job
      2. Requests resources
      3. Manages MR tasks
      4. Monitors progress of tasks.
      5. Launches application containers.
      6. Sends reports to the client.
      7. Carries out execution of the job.
      8. Terminates after completion of the job.
   4. **Node Manager** - 1 per slave node, manages resources on slave
   5. **JobHistory** - 1 per cluster, archives jobs’ metrics and metadata
4. Running a Job in a MapReduce v2 Cluster  
     
     
   1. Job Submission
      1. MR program contacts the job client.
      2. Client requests job ID.
      3. Checks whether the output has already been created.
      4. Copies resources to HDFS with high replication.
      5. Submits the job to resource manager.
      6. SR
   2. Job Initialization
      1. Resource Manager contacts a Node Manager to allocate a container and launch Application Master.
      2. Application Master one map per split.
      3. Application Master decides how to run MR jobs.
         1. Uber tasks are jobs that the Application Master decides to run on the same JVM on a single node.
            1. For small jobs.
            2. This happens because the overhead of allocating other resources for the jobs outweighs the overhead of running them on the same JVM.
         2. SR
   3. Task Assignment
      1. Scheduler considers data locality.
      2. If job is not uber, Application Master contacts Resource Manager
   4. Task Execution
      1. YARN child is launched.
         1. Analogous to the TaskTracker.
         2. It is a Java program.
         3. Runs on a separate JVM to isolate it from user code.
         4. While in Classic MR, a task cold run on the same JVM as the TaskTracker, in YARN, tasks cannot run on the same JVM as the YARN child.
      2. SR
   5. Progress Updates
      1. YARN sends progress to Application Master every three seconds.
      2. Application Master updates the client.

### Job Failures

1. Job Failures in Classic MapReduce
   1. Three types of failures
      1. Task failure (code error)
      2. TaskTracker
      3. JobTracker
   2. Task failures
      1. If a task stalls or takes too long to finish, the TaskTracker would mark the job failed after a period of time.
         1. The mapred.task.timeout property sets the time to wait for a task to finish.
         2. That property can be set to 0, which would prevent the TaskTracker from ever failing a long-running job. (Not recommended)
      2. If a task fails, the TaskTracker attempts to re-run it.
         1. The mapred.map.max.attempts property sets the number of attempts on a map task.
         2. The mapred.reduce.max.attempts property sets the number of attempts of a reduce task.
         3. Both properties default to 4.
   3. TaskTrackers can fail too.
      1. If the JobTracker stops receiving heartbeats from a TaskTracker, the JobTracker concludes that the TaskTracker is dead.
      2. In that case, the JobTracker reschedules the TaskTracker’s tasks on a different TaskTracker.
      3. The JobTracker reschedules both…
         1. The uncompleted tasks.
         2. And the tasks that did complete but whose job did not.
      4. The JobTracker also removes the TaskTracker from its pool of available TaskTrackers.
      5. If the number of tasks that fail on a TaskTracker crosses a threshold (mapred.max.tracker.failures), the TaskTracker is blacklisted.
         1. It is not available until restart.
         2. Or after a certain period of time.
   4. JobTracker failures
      1. JobTracker is the SPOF, so not much can be done if it fails.
      2. The most serious failure in MR
      3. Thus, JobTracker should run on better hardware.
      4. In that case, all jobs in progress should be resubmitted.
2. Job failures in YARN
   1. Task failure – handled the same as in Classic MR.
   2. Some property names changed
      1. mapred.map.maxattempts replaced ….map.attempts.
      2. Same for the reduce attempts.
      3. Failure threshold has been changed to:
         1. mapred.map.failires.maxpercent
         2. map.reduce.failures.maxpercent
   3. Application Master failures
      1. Tasks do not have to be re-run.
      2. They can be recovered if yarn.app.mapreduce.am.job.recovery.enable is true.
      3. Handled like task failures.
      4. Resource Manager stops getting heartbeats – starts AM on a new container. Execution of job is continued.
   4. Node Manager failures
      1. Resource Manager stops getting heartbeats after a period of time.
      2. All remaining tasks are re-spawned on a new Node Manager.
      3. Blacklisting is the same as in Classic.
         1. Blacklisting identifies poorly performing nodes.
         2. SR
   5. Resource Manager failures
      1. The most serious failure.
      2. Without RM, the jobs
      3. YARN has a backup: it can start a new ResourceManager that can recoever the last saved state, so all of the jobs don’t have to be re-run.

## Writing MapReduce Jobs

1. Components of a MapReduce job
   1. Write a **mapper**.
   2. Write a **reducer**.
   3. Write a **driver** program.
      1. Contains the main method which invokes the mapper(s) and reducer(s).
      2. Specifies job configuration and the type of components to use in the job.
   4. Other possible tasks to write[[3]](#footnote-3)
      1. Combiner
      2. Partitioner
2. The Mapper must extend the abstract Mapper class

|  |
| --- |
| // Example of a concrete Mapper import org.apache.hadoop.io.IntWritable; |
| import org.apache.hadoop.io.LongWritable; |
| import org.apache.hadoop.io.Text; |
| import org.apache.hadoop.mapreduce.Mapper; |
|  |
| public class MaxTemperatureMapper |
| extends Mapper<LongWritable, Text, Text, IntWritable> { |
|  |
| @Override |
| public void map(LongWritable key, Text value, Context context) |
| throws IOException, InterruptedException {  .... |
| } |
| } |

* 1. The Mapper class is a generic type with 4 parameters: KEYIN, VALUEIN, KEYOUT, VALUEOUT.
     1. These formal parameters can only be Hadop-specific data types.
     2. SR
  2. Override the map method to do the mapping.
     1. The map method takes the parameters of KEYIN and VALUEIN type, as well as a Context object.
     2. The result is written to the Context object with context.write().
        1. The first parameter is the KEYOUT.
        2. The second parameter is the VALUEOUT.
     3. The parameter types must match the parameter types of declared in the concrete class’ parameters—the KEYIN and VALUEIN types.
     4. Throws IOException and InterruptedException.
     5. Return value is void. The result is written to the Context object.

1. The Reducer must extend the abstract Reducer class.

//Example of a concrete Reducer  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Reducer;

|  |
| --- |
| public class MaxTemperatureReducer extends Reducer<Text, IntWritable, Text, IntWritable> {  @Override |
| public void reduce(Text key, Iterable<IntWritable> values, Context context)  throws IOException, InterruptedException {  ...... |
| } |

}

* 1. It is also a generic, taking the KEYIN, VALUEIN, KEYOUT, VALUEOUT parameters.
     1. The KEYIN and VALUEIN types must match the KEYOUT and VALUEOUT types of the concrete Mapper class.[[4]](#footnote-4)
     2. SR
  2. Override the reduce method to perform your reduction.
     1. The method takes a KEYOUT, VALUEOUT, and Context object as input.
     2. The parameter types must be the same as those declared in the class’ parameters—the KEYOUT and VALUEOUT.
     3. Any result is written to the Context object.
        1. Call context.write()
        2. The first parameter is the key.
        3. The second parameter is the reduced value, converted to the expected VALUEOUT type.
        4. Example  
             
           public void reduce(Text key, Iterable<IntWritable> values, Context context){  
            int sum = 0;  
            for (IntWritable value: values){  
            sum += value.get();  
            }  
            context.write(key, new IntWritable(sum));  
           }
     4. The VALUEIN for the reduce method is always an Iterable because the value for each key sent from the shuffle-and-sort sends a list of values.
     5. Every reduce ends up using the same loop in its solution—iterating over the VALUEIN Iterable and doing something with each value to reduce the list of values associated with the key to one value.
     6. Throws IOException and InterruptedException.
     7. Return value is void. The result is written to the Context object.

1. The third essential piece of a MapReduce job is a driver program that runs the job.

// Example of a driver program to run the MR job

public class MaxTemperature {

public static void main(String[] args) throws Exception {  
 ....  
 Job job = new Job();  
 job.setJarByClass(MaxTemperature.class);  
 job.setJobName("Max temperature");  
  
 # Setting input and output paths  
 FileInputFormat.addInputPath(job, new Path(args[0]));  
 FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
 # Setting Mapper and Reducer classes  
 job.setMapperClass(MaxTemperatureMapper.class);  
 job.setReducerClass(MaxTemperatureReducer.class);  
 job.setOutputKeyClass(Text.class);  
 job.setOutputValueClass(IntWritable.class);  
 System.exit(job.waitForCompletion(true) ? 0 : 1);  
 }  
}

* 1. The driver should create a Job object to control the job.
  2. The input and output paths for the job are command line arguments that are added to an InputFormat, like FileInputFormat above.
  3. Set the mapper class and reducer class in the Job object.
  4. The job is triggered by a call to job.waitForCompletion().

1. Defaults
   1. Suppose you ran a job with minimal configuration—that is, without specifying did not specify a mapper, reducer, etc.
      1. The map method of the Mapper base class would be used as the default.
      2. And the reduce method of the Reduce base class would be used as the default.
      3. And no combiner would be used.
      4. The end result would be a set of key/value pairs in which the values are the original records and the keys are the byte offsets of the records.
         1. Essentially this would be an identity mapper and reducer.
         2. The default mapper and reducer simply spit out what comes in.
   2. The map method of Mapper simply writes the KEYIN and VALUEIN to its Context argument.
   3. The reduce method of Reducer does the same.
2. Package the job in a JAR.
   1. Hadoop will distribute that JAR around the cluster.
   2. When you run the job, you will use the hadoop command line task, specifying the name of the JAR containing the job, followed by the input file as the next parameter and the output file as the last parameter.
3. Hadoop Wrapper Classes
   1. Keys and Values in Hadoop are wrappers objects around primitives
      1. Keys and Values are NOT primitives.
      2. This is done to support serialization for HDFS
         1. These are needed for efficiency.
         2. They are compact, fast.
      3. Examples:
         1. IntWritable -> int
         2. LongWritable -> long
         3. FloatWritable -> float
         4. DoubleWritable -> doubles
         5. Text -> String
   2. Values are objects which implement the ***Writable*** Interface.
   3. Keys are objects which implement ***WritableComparable*** Interface.
      1. A *WritableComparable* is a Writable which is also Comparable
      2. Two *WritableComparable* objects can be compared to determine their sort order.
      3. Keys must be *WritableComparable* so that they can be passed to the Reducer in sorted order.
      4. Hadoop box classes implement both *Writable* and *Comparable*.
         1. Ex: *IntWritable* is actually a *WritableComparable*.
         2. SR
      5. The data type for map output must implement WritableComparable.
4. Other important classes in the MapReduce API
   1. ***InverseMapper*** - swaps keys and values
   2. ***RegexMapper*** - extracts text based on a regular expression
   3. ***IntSumReducer*** - adds up all Integer values for a key
   4. ***LongSumReducer*** - adds up all Long values for a key
   5. ***IdentityReducer*** – merely spits out the input it was given without reducing.
      1. You would use it when you don’t want to reduce, but you still want the shuffle-and-sort to happen.
      2. Shuffle-and-sort doesn’t happen without a reducer.
5. ToolRunner
   1. The job configuration can be set in the job driver; however, doing so reduces portability.
   2. By implementing the Tool interface and extending the Configured class, we can leverage Hadoop’s *Configuration* object via the *GenericOptionsParser*.
      1. Hence, we can specify configuration at run-time via command line, using Java options.
      2. hadoop jar uber.jar mypackage.MyDriver **-D mapred.reduce.tasks=1** input output
   3. How to use command-line parameters with the –D option[[5]](#footnote-5)
      1. Extend Configured and implement Tool interface
      2. Anything specified in the option -D at command line becomes part of the Configuration:  
         hadoop jar myjob.jar package.MyDriver -Dcolor=yellow inputDir outputDir
      3. Set the values onto the configuration object in main of the driver.

Configuration conf = new Configuration();  
conf.setInt(“favcolor”, “blue”); // blue is default color  
Job job = new Job(conf);

* + 1. Get the parameters within the mapper or the reducer.

Configuration conf = context.getConfiguration();

String myColor = conf.get(“favcolor”, “blue”);

* 1. ToolRunner command-line options[[6]](#footnote-6)
     1. -D flag will override any default or site properties in the configuration but will not override those set in the driver code.
     2. -D options must appear before any additional program arguments:  
        hadoop jar myjob.jar MyDriver -Dmapred.reduce.tasks=10 indir outdir

1. Mapper/Reducer initialization[[7]](#footnote-7)
   1. If you want to perform some initialization prior to Map or Reduce, you can use the setup method.
   2. Some tasks that you may want to perform in setup:
      1. Initialize data structures
      2. Read from external file
      3. Grab parameters from the Configuration
   3. The setup method is run prior to the map or reduce method being called for the first time:  
        
      public void setup(Context context)  
      1. Anything relating to setup that needs to be done once per mapper/reducer can be done here.
      2. SR
2. Mapper/Reducer teardown.
   1. Use the cleanup method to perform some action after your Mapper or Reducer has processed all the records.
   2. The cleanup method is called before the Mapper or Reducer terminates:
   3. Signature:  
        
      public void cleanup(Context context) throws IOException, InterruptedException
3. Side data distribution
   1. Often you will want to use some side data along with the processing going on in your Mapper or Reducer. For Instance:
      1. Lookup Tables
      2. Dictionaries
      3. Configuration values
   2. Options for accessing side data
      1. Use the Configuration
         1. In the setup method, use setter/getter methods on Configuration to access configuration properties.
         2. This works well for small amounts of data, but won’t scale because:
            1. Job configuration is read by JobTracker, TaskTracker, and child JVM.
            2. Every time configuration is read, all entries read into memory
      2. Use the Distributed Cache
         1. Distributed Cache is the better method long-term.
         2. Very important.
   3. Distributed Cache
      1. A service for copying files and archives to task nodes.
      2. Distributed Cache is the only way to do a join between separate sets of data in MapReduce.
      3. Files are copied in time for tasks to use them when they run.
      4. Files are normally copied to a node once per job.
      5. Files are automatically deleted from slave nodes when the job finishes.
   4. Files in Distributed Cache are read-only.
   5. Use ToolRunner to add files to Distributed Cache from command line.
      1. Don’t need to copy files to HDFS first.
      2. Use the -files option to add files:  
         hadoop jar myjob.jar MyDriver -files file1, file2, …, file\_n
      3. To distribute a single file you can use the –file option.
   6. The -archives flag adds archived files, and automatically unarchives them on the destination machines.
   7. The -libjars flag adds jar files to the classpath.
   8. Files added to the Distributed Cache are made available in your task’s local working directory.
      1. These files would be accessed from your mapper or reducer the same way that you would read any ordinary local file:  
         File f = new File(“file\_loaded\_into\_cache”);[[8]](#footnote-8)
      2. Connect to the cache in the setup method of the mapper or reduce class.
      3. No changes are needed to the driver class, as long as you were using ToolRunner.
4. Designing Solutions
   1. It helps to begin by picturing the output that you want and reverse-engineering it.
   2. It also is important to get the MR data flow in mind, in particular the following parts:
      1. Mapper
      2. Shuffle-and-sort
      3. Partitioner
      4. Reducer
   3. Newbies often stuck most of their logic in the Mapper, thus do not take advantage of parallel processing.
   4. Also if you only use one reducer, you lose a lot of the benefits of the parallel processing beforehand.
      1. The number of reducers used is controlled by the programmer.
      2. The number of mappers is controlled intelligently by the MapReduce framework.
   5. One objective is to take advantage of Hadoop’s parallel processing.

## Combiners and Partitioners

1. Combiners
   1. Use them to help reduce large amounts of intermediate data.
   2. Using them whenever possible is recommended.[[9]](#footnote-9)
   3. Mappers often produce large amounts of intermediate data.
      1. That data must be passed to the reducers.
      2. This can result in a lot of network traffic.
   4. One solution is to specify a Combiner
      1. This acts like a ‘mini-Reducer’
      2. The combiner runs locally on a single Mapper’s output.
      3. Output from the Combiner is sent to the reducers.
      4. Think of a Combiner as a Reducer that runs locally.
   5. Combiner and Reducer code are often identical.
      1. For example, in Tom White’s example of a job with Combiner, in Ch. 2, the combiner and reducer are the exact same class—using the same class in job.setCombinerClass() and job.setReducerClass().
      2. SR
   6. Using Combiners is only possible when the operation performed is both commutative and associative.

Associative and Communicative operations

* **Associative** - order of operations doesn’t matter as long as the sequence of operators is not changed.
  + (A + B) + C = A + (B + C)
  + addition and multiplication
  + min / max of a set
  + string concatenation
* **Commutative** - order of operands does not change the result
  + A + B = B + A
  + addition and multiplication
  + scalar multiplication of vectors
  + certain binary truth functions
  + intersection / union of sets
    1. In other words, the operation performed by the combiner should not depend on the order of values processed by the operation.
    2. This is required because combiners can run multiple times on map output, and may run in different order each time.
  1. The input and output data types for the Combiner/Reducer must be identical.
     1. Also, as with Reducers, the input for Combiners is a list of key/list-of-values pairs.
     2. And, as with Reducers, the output is a list of key/value pairs.
  2. Like Reducers, Combiners must extend Reducer.
     1. Input: intermediate key and a list of values
     2. Output: zero or more key/value pairs
     3. The actual method called is the reduce method in the class:  
        reduce(intermediate\_key, [v1,v2, ...vn]) -> (result\_key, result\_value)
  3. Specify the Combiner class to be used in the Job’s Driver routine

job.setMapperClass(WordMapper.class);  
job.setReducerClass(SumReducer.class);  
**job.setCombinerClass(SumReducer.class);**

* 1. Again, input and output data types for Combiner and Reducer must be identical
  2. The combiner may run more than once on the output from any given Mapper. Therefore, do not put code in the Combiner that could influence your results in case the Combiner runs more than once.

1. Partitioners
   1. The job of the partitioner is to determine to which reducer each intermediate key and its associated value(s) will be sent.  
        
      
   2. Unless you specify a specific partitioner in the job configuration, then the default partitioner will be HashPartitioner.
      1. Guarantees that all pairs with the same key will go to the same Reducer
      2. Uses the hashcode() method of the key, and modulo with the number of reducers, in order to determine which reducer to send a given key/value pair to.
      3. For most randomly-distributed data, this should result in all partitions being of roughly equal size.
   3. To specify a partitioner:  
      job.setPartitionerClass(Class); // instance method
   4. The partitioners feed data into the shuffle and sort.
   5. The reducers pull data from the intermediate results that came from the partitioners.
   6. **The type of partitioner and the configuration of the job determine how many reducers will be present in the job.** [[10]](#footnote-10)
      1. You can set the number explicitly with the mapred.reduce.tasks property.
      2. SR
   7. Getting sorted output
      1. **If a job needs to output a file containing all the keys in sorted order, then use a single reducer.**
      2. An alternative is to use the ***TotalOrderPartitioner***.
         1. This partitioner uses an externally generated file which contains information about intermediate key distribution.
         2. It partitions data such that all keys which go to the first reducer are smaller than any which go to the second, and so on…
         3. Therefore, the final output concatenated has the output files in a totally ordered list.
   8. Custom Partitioners
      1. To create your own partitioner custom class, implement the ***Partitioner*** interface

public interface Partitioner<K, V> extends JobConfigurable {  
 int getPartition(K key, V value, int numPartitions);  
}

* + - 1. You want a Partitioner which evenly distributes your map output.
      2. The getPartition method must return a number between 0 (inclusive) and the number of Reducers (exclusive).
      3. Make sure that the function you use to calculate the partition number does not return a negative number (wrap with Math.abs()).
    1. To use the customer Partitioner, configure your partitioner in the driver:  
         
       job.setPartitionerClass(MyCustomPartitioner.class);
    2. If you need to use configuration variables in the Partitioner, then the Partitioner should implement ***Configurable***.

class CustomPartitioner extends Partitioner<K,V> implements Configurable {

private Configuration configuration;

/ / Define your own variables here

@Override

public void setConf(Configuration configuration) {

this.configuration = configuration;

// set up your variables here

}

@Override

public Configuration getConf() {

return configuration;

}

public int getPartition(K key, V value, int numReduceTasks) {

// use your variables here

}

….

}

* + - 1. Any Hadoop object that implements *Configurable* allows its setConf() method to be called one time during instantiation.
      2. Thus you can set up variables in the setConf() method which your getPartition() method will then be able to access.
      3. Takes a key/value pair as input and returns an integer result.
    1. Use cases
       1. The keys for your data are not evenly distributed but skewed in some way.
       2. You have a key that is a custom WritableCompable which contains a pair of values (A, B).
          1. You may need all keys with same value for A to go to same Reducer.
          2. Default partitioner won’t work in this case.
       3. You are performing a secondary sort.
    2. Native Hadoop Partitioners
       1. ***BinaryPartitioner***
          1. Used for when you have keys that are ***BinaryComparable***.
          2. The partition based upon the configurable part of the bytes array returned by BinaryComparable.getBytes().
       2. ***HashPartitioner***
          1. The default Partitioner that uses the key’s hashcode() method
          2. SR
       3. ***KeyFieldBasedPartitioner***
          1. Allows for using only parts of the key for comparison
          2. Programmer would configure the separator that separates the parts of the key.
       4. ***TotalOrderPartitioner***
          1. Used in conjunction with *InputSampler* or some other sampled index.
          2. *TotalOrderPartitioner* uses an external sample index to set the partition.

## Input/Output Format Classes

1. Input formats
   1. In MapReduce, data is read from a file and parsed into key/value pairs before being handed to the mapper. The input format class used for the job controls this parsing.
   2. Specified in the driver code with Job.setInputFormatClass(), an instance method.
   3. Defines the location of the input data - typically a file or a directory
   4. Determines how to split the input data into ***input splits***
      1. Each mapper deals with a single input split
      2. So the number of input splits determines the number of mappers, as said above.
   5. Creates a ***RecordReader*** object, which parses the input data into key/value pairs to pass to the mapper
   6. InputFormat is an interface.
   7. The default input format is ***TextInputFormat***.  
        
        
      1. Creates a LongWritable key, which is the byte offset from the start of the input file.
      2. Creates a Text value, which is a line from the file, excluding the line terminator
         1. Treats each \n-terminated line of a file as a value.
         2. SR
      3. Creates a group of ***LineRecordReader*** objects.
   8. Other standard *InputFormat* classes
      1. ***FileInputFormat***: abstract base class used for all file-based InputFormats
      2. ***KeyValueTextInputFormat***: maps lines in which a key and a value already exist and are separated by the configured separator.
         1. Tab is default separator.
         2. \n is taken as the end of a line.
         3. If no separator is present, the line becomes the key, and the value is empty.
      3. ***SequenceFileInputFormat***: binary file of (key,value) pairs plus meta-data
      4. ***SequenceFileAsTextInputFormat***: binary file of (key,value) but maps (key.toString(), value.toString())
      5. ***NLineInputFormat***: divides the input into splits, and each mapper would get a fixed number of lines.
2. Output formats
   1. When the reducer emits key/value pairs, the output format controls how the output is sent to the output file.
   2. The default is TextOutputFormat.
      1. Keys and values are converted to Strings and printed.
      2. By default, keys and values are separated by the tab character.
      3. However, the character that separates keys and values can be specified with the property mapreduce.output.textoutputformat.separator.[[11]](#footnote-11)
   3. The output format can be set in the driver with Job.setOutputFormatClass(), an instance method.
3. Sequence Files
   1. SequenceFiles are flat files containing binary-encoded key-value pairs
      1. Work naturally with Hadoop data types
      2. Include metadata which identifies the data types of the key/value
      3. Cannot be processed as Text objects.
   2. Three file types in one
      1. Uncompressed
      2. Record-compressed
      3. Block-compressed
   3. Compressible and splittable.
   4. Used in intermediate map results.
   5. Use cases
      1. Use where the output from one MapReduce job is the input for another MapReduce job.
      2. Good for sorting too.
   6. Format classes for Sequence Files
      1. SequenceFileInputFormat
      2. SequenceFileOutputFormat
   7. How to access sequence files directly

Configuration config = new Configuration();

SequenceFile.Reader reader =

new SequenceFile.Reader(FileSystem.get(config), path, config);

Text key = (Text) reader.getKeyClass().newInstance();

IntWritable value = (IntWritable) reader.getValueClass().newInstance();

while (reader.next(key,value)) {

 // do something here

}

reader.close();

* 1. SequenceFiles are useful but have some potential problems
     1. They are only typically accessible via the Java API; however, some work has been done to allow access from other languages
     2. If the definition of the key or value object changes, then the file becomes unreadable.

## Old vs. New Java APIs

1. The new API, sometimes referred to as “Context Objects,” was designed to make the API easier to evolve in the future.[[12]](#footnote-12)
2. The new API is type-incompatible with the old, however, so applications need to be rewritten to take advantage of it.
3. Differences
   1. The new API favors abstract classes over interfaces, since these are easier to evolve.
      1. This means that you can add a method (with a default implementation) to an abstract class without breaking old implementations of the class.
      2. For example, the Mapper and Reducer interfaces in the old API are abstract classes in the new API.
   2. The new API is in the org.apache.hadoop.mapreduce package (and subpackages). The old API can still be found in org.apache.hadoop.mapred.
   3. The new API makes extensive use of context objects that allow the user code to communicate with the MapReduce system.
      1. The new Context, for example, essentially unifies the role of the JobConf, the OutputCollector, and the Reporter from the old API.
      2. SR
   4. In both APIs, key-value record pairs are pushed to the mapper and reducer, but in addition, the new API allows both mappers and reducers to control the execution flow by overriding the run() method.
      1. For example, records can be processed in batches, or the execution can be terminated before all the records have been processed.
      2. In the old API this is possible for mappers by writing a MapRunnable, but no equivalent exists for reducers.
   5. Job control is performed through the Job class in the new API, rather than the old JobClient, which no longer exists in the new API.
   6. Configuration has been unified.
      1. The old API has a special JobConf object for job configuration, which is an extension of Hadoop’s vanilla Configuration object (used for configuring daemons).
      2. In the new API, job configuration is done through a Configuration, possibly via some of the helper methods on Job.
   7. Output files are named slightly differently: in the old API both map and reduce outputs are named part-nnnnn, whereas in the new API map outputs are named part-m-nnnnn, and reduce outputs are named part-r-nnnnn (where nnnnn is an integer designating the part number, starting from zero).
   8. User-overridable methods in the new API are declared to throw java.lang.InterruptedException.
      1. This means that you can write your code to be responsive to interrupts so that the framework can cancel long-running operations gracefully.
      2. SR
   9. In the new API, the reduce() method is passed values as a java.lang.Iterable, rather than as a java.lang.Iterator (as the old API does). This change makes it easier to iterate over the values using Java’s for-each loop construct:  
        
      for (VALUEIN value : values) { ... }

# Advanced MapReduce Topics

## Debugging MapReduce Jobs

1. Challenges of debugging in a distributed environment
   1. Troubleshooting and Debugging Map Reduce Jobs is different from standard programs
   2. Due to the distributed nature of MapReduce tasks, every instance of a mapper runs as a separate task, often on a different machine
   3. In such a scenario, it is difficult to attach a debugger to the process
   4. Different nodes mean that log files for tasks are spread around the cluster
   5. Challenges of “Big Data”
      1. Large amounts of data make it difficult to catch edge cases
      2. Unexpected inputs are likely to appear throughout large amounts of unstructured data
   6. Approach to MapReduce Debugging
      1. Start small and build incrementally
      2. Ensure that input data is in the expected format
      3. Use Hadoop’s various run modes to scale your development
      4. Use MRUnit and JUnit unit test frameworks and write unit tests
      5. Expect problems and catch exceptions
         1. A thrown exception will cause a job to fail
         2. Catch exception and then use logging to send a message to the log file
   7. Debugging Tools
      1. Hadoop Run Modes
      2. Logging
      3. Unit Testing
      4. Job Counters
   8. Hadoop Run Modes
      1. Standalone or Local Mode
         1. Commonly used for development. In this mode, everything runs in a single JVM.
         2. Since everything is in a self-contained JVM, then you can attach a debugger from your IDE.
         3. Hadoop can run MapReduce in a single, local process
            1. Does not require any Hadoop daemons to be running
            2. Uses the local file system instead of HDFS
            3. Is a very useful way of quickly testing changes in code
         4. Limitations
            1. Distributed Cache is not available in this mode
            2. Any job can only specify a single Reducer
            3. Certain mistakes may not be caught

For example, attempting to share data between Mappers will work because the code is running in a single JVM

* + - 1. How to run in Local Mode
         1. Option 1: add the following line to your driver code:

Configuration conf = new Configuration();

conf.set(“mapred.job.tracker”, “local”);

conf.set(“fs.default.name”, “file:///”);

* + - * 1. Option 2: set the fs and jt options on the command line if your driver uses ToolRunner:

fs is equivalent to -D fs.default.name

jt is equivalent to -D maprep.job.tracker

Example:  
$ hadoop jar myjar.jar MyDriver -fs=file:/// -jt=local indir outdir

* + 1. Pseudo Distributed Mode
       1. Useful for development and debugging, but simulates a real Hadoop cluster without some of the limitations of Standalone Mode.
       2. However, all of the Hadoop Daemons run on one node.
    2. Fully Distributed Mode
       1. Used for production and large-scale testing of development.
       2. Uses a real cluster of machines with multiple nodes.



Figure : Differences between Hadoop Run Modes

## Logging

1. Writing to Standard Output or Standard Error
   1. Many Java programmers like to write debug messages to standard output[[13]](#footnote-13)
   2. stdout and stderr will show results if running in LocalJobRunner Mode
   3. If running on a cluster, that output will not appear on your console
      1. However, stdout and stderr will be visible via Hadoop’s Web UI
      2. SR
2. Using log4j
   1. Hadoop uses the log4j library to generate all of its log files
   2. Your Mappers and Reducers can also use log4j, initialized “automagically” by Hadoop
   3. Add log4j.jar-<version> file from the distribution or place the dependency in your Maven pom.xml file

import org.apache.log4j.Level;

import org.apache.log4j.Logger;

class GooMapper implements Mapper {

private static final Logger LOGGER = Logger.getLogger

(GooMapper.class.getName());

…

}

* 1. Simply send strings to loggers tagged with severity levels

LOGGER.trace(“debug message”);

LOGGER.debug(“debug message”);

LOGGER.info(“Smurf Hammer for the win”);

LOGGER.warn(“Invalid path for xyz”);

LOGGER.error(“Application has generated invalid value”);

* 1. Beware of expensive operations like string concatenation in log messages.
  2. Node-wide configuration for log4j is stored in /etc/hadoop/conf/log4j.properties
  3. You can override settings for your application either in your own log4j.properties or by setting the log level programmatically (e.g., LOGGER.setLevel(Level.WARN)).
  4. Setting the log levels for a job
     1. You can tell Hadoop to set logging levels for a job using configuration properties:
        1. -mapred.map.child.log.level
        2. -mapred.reduce.child.log.level
     2. Examples
        1. To set the logging level to DEBUG for the Mapper:  
           hadoop jar myjob.jar MyDriver -Dmapred.map.child.log.level=DEBUG
        2. To set the logging level to WARN for the Reducer:  
           hadoop jar myjob.jar MyDriver -Dmapred.reduce.child.log.level=WARN

1. Where to find Hadoop log files
   1. Log files are stored on the machine where the task attempt ran.
   2. The location is configurable.
   3. Default location: /var/log/hadoop-0.20-mapreduce/userlogs/${task.id}/syslog
   4. You will often not have ssh access to a node to view its log files
      1. Much easier to use the JobTracker Web UI which can automatically retrieve and display the log files for you.
      2. SR

## Unit testing with MRUnit

1. MRUnit is Apache Hadoop’s testing framework.
2. Built upon Junit and Mockito
   1. JUnit - provides familiar unit testing framework
   2. Mockito - used to mock up MapReduce framework

<dependency>

         <groupId>org.apache.mrunit</groupId>

         <artifactId>mrunit</artifactId>

         <version>0.9.0-incubating</version>

         <classifier>hadoop2</classifier>

 </dependency>

<dependency>

     <groupId>junit</groupId>

     <artifactId>junit</artifactId>

     <version>4.4</version>

     <scope>test</scope>

</dependency>

<dependency>

        <groupId>org.mockito</groupId>

        <artifactId>mockito-all</artifactId>

        <version>1.9.5</version>

        <scope>test</scope>

</dependency>

Figure : Maven Depndencies for MRUnit

1. MRUnit can be used to test the Mapper, the Reducer, or the full MapReduce Flow
   1. MRUnit builds on top of JUnit
   2. MRUnit provides a mock InputSplit and other classes
   3. @Begin
      1. Set up your MapDriver, ReduceDriver, or MapReduceDriver
      2. Set up your mapper and your reducer
   4. @Test
      1. Set up input expectations and output expectations
      2. Run your Mapper or Reducer or both
2. MRUnit Approaches
   1. MRUnit supports two testing approaches.
      1. Tell the framework both input and output values and let the framework do the assertions. (This is the simpler and easier approach)
      2. The more traditional approach where you do the assertion yourself.
   2. Running under your development environment
      1. Tests can run within your IDE (Eclipse, IntelliJ, Netbeans)
      2. Tests could run at the command line or batch (Hudson, CruiseControl)
      3. You do not need Hadoop installed in order to use MRUnit
3. Use a driver in order to run your tests
   1. MapDriver - for your mapper
   2. ReduceDriver - for your reducer
   3. MapReduceDriver - for your full MapReduce Flow
4. Input and Output Methods
   1. withInput
      1. Specifies input to the Mapper/Reducer
      2. Has a builder method that can be chained
   2. withOutput
      1. Specifies expected output from the Mapper/Reducer
      2. Has a builder method that can be chained
   3. addOutput
      1. Similar to withOutput but returns void
      2. SR
5. Methods to run tests
   1. runTest - runs the test and verifies the output
   2. run - runs test and returns result set, ignores previous withOutput and addOutput calls
6. Drivers
   1. Take a single key/value pair as input
   2. Take multiple key/value pairs as expected output
   3. If you are calling driver.runTest() or driver.run() multiple times, then call driver.resetOutput() between each call. Otherwise, MRUnit will fail.

## Counters

1. Counters allow Mappers or Reducers to pass aggregate values back to the job driver after the job has completed
   1. Their values are also visible from the JobTracker’s Web UI
   2. Values are also reported back on the console after a job completes.
   3. If you have a performance issue, and want to look at the execution times of the map and reduce phases, look first at the output of the job under “Job Counters.”
2. Counters are very basic - just have a name and a value
3. Counters are collected into Groups - within group, each counter has a name
   1. Example: a group of Counters called RecordType
      1. Names: TypeAlpha, TypeBeta, TypeKappa
      2. Appropriate counter can be incremented as each type of record is read in the mapper
   2. Again, these are listed in the console output of a job.
4. Counters can be set and incremented via the getCounter method:  
     
   context.getCounter(“RecordType”, “Alpha”).increment(1);
5. To retrieve counters in the driver code, use code like this:

long typeAlphaRecs = job.getCounters().findCounter(“RecordType”, “Alpha”).getValue();  
long typeBetaRecs = job.getCounters() .findCounter(“RecordType”, “Beta”).getValue();

1. Do not rely on a counter’s value from the Web UI while a job is running
   1. Due to possible speculative execution, a counter’s value could appear larger than the final value
   2. Modifications to counters from subsequently killed/failed tasks will be removed from the final count.
2. Divided into Task Counters and Job Counters
   1. Task counters can be user-defined or built-in.
   2. User-defined counters.
      1. Using them is a good programming practice.
      2. Employed though the Context object in the map/reduce methods.
      3. Again, these can be used for debugging.
         1. For example, in your map method, you can have:  
              
            if (Character.isAlphabetic(word.toString().charAt(0))){  
             context.write(word, 1);  
            } else {  
             context.getCounter(“bad words counter”, “Bad records”).increment(1);  
            }
         2. If you get expected input, you write it to the Context.
         3. But if you get unexpected input, you increment the user-defined “Bad words counter.”

## Serialization

1. Hadoop’s next generation of data serialization is Avro.
   1. Apache Avro is a serialization format which is becoming a popular alternative to SequenceFiles
   2. Project was created by Doug Cutting, the creator of Hadoop
   3. Self-describing file format: schema for data included in the file itself
   4. Uses JSON for describing data types and protocols.
   5. Serializes data in a compact binary format.
   6. Portable across multiple languages: C, C++, Java, Python, and Ruby.
   7. Compatible with Hadoop via the AvroMapper and AvroReducer classes.

## File Compression

1. Important to know in Hadoop because Hadoop is distributed and compressed files cannot be split in all places, and not all compressed files are splittable.
2. Hadoop works with a variety of file compression formats.
3. If a compressed file is included as one of the input files, Hadoop will automatically decompress it and pass contents to the Mapper.
4. Hadoop will support some common formats
   1. gzip
   2. bzip2
   3. LZO
   4. Snappy
5. Considerations when using compression
   1. Compression within Mapper means less data being pulled across the network -> performance gains
   2. Compressing output will certainly lead to less disk space being consumed.
   3. Not every compression format is able to be split (eg. gzip)
      1. If file cannot be split, then everything in that file will be sent to a single mapper task.
      2. If input file size is sufficiently large, then using a non-splitable format may cause performance loss.
   4. Some compression algorithms take longer to compress/decompress and this may cause jobs to become CPU bound.
6. Compression format characteristics

|  |  |  |  |
| --- | --- | --- | --- |
| **Format** | **Algorithm** | **Extension** | **Can Split?** |
| gzip | DEFLATE | .gz | NO |
| bzip2 | bzip2 | .bz2 | YES |
| LZO | LZO | .lzo | YES, if indexed |
| Snappy | Snappy | .snappy | NO |

1. LZO Compression
   1. LZO is capable of being divided into Hadoop splits
   2. Because of licensing restrictions, LZO cannot be shipped with Hadoop
      1. But… it is easy to add
      2. See https://github.com/cloudera/hadoop-lzo
   3. To make an LZO file splittable, you must first index the file
   4. The index file contains information about how to break the LZO file into splits that can then be decompressed
   5. Access the splittable LZO file as follows:
      1. Use LzoTextInputFormat class
      2. For streaming jobs, specify the input format on the command line property com.hadoop.mapred.DeprecatedLzoTextInputFormat
2. The Snappy Codec
   1. Snappy is a relatively new and fast compression codec developed by Google.
   2. Snappy trades file compression size and compatibility of other formats for high speed. **Very Fast**.
   3. Snappy does not compress a SequenceFile and produce a file with a .snappy extension
      1. Instead, it is a codec that can be used to compress data within a file
      2. That data can be decompressed automatically by Hadoop or other programs at the time the file is read
      3. Snappy works well with SequenceFiles and Avro Files
   4. Snappy is now preferred over LZO
   5. Using Snappy compression
      1. Specify output compression in the Job object
      2. Specify either block or record compression (Block is recommended for Snappy)
         1. Set the compression codec to the Snappy codec in the Job object

import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputForma;

import org.apache.hadoop.io.SequenceFile.CompressionType;

import org.apache.hadoop.io.compress.SnappyCodec;

job.setOutputFormatClass(SequenceFileOutputFormat.class);

FileOutputFormat.setCompressOutput(job, true);

FileOutputFormat.setOutputCompressorClass(job, SnappyCodec.class);

SequenceFileOutputFormat.setOutputCompressionType(job, CompressionType, BLOCK);

Figure : Example of using Snappy compression

## Algorithms

### Sorting

1. The ability to sort is at the heart of MapReduce.
2. Using one reducer (easy)
   1. Remember that keys are passed to the Reducer from the Shuffle-Sort in sorted order.
   2. If we are only using one Reducer, then our output file will automatically be sorted.
   3. In such a case, the file to be sorted will have a mapper and reducer with identity functions
      1. Mapper - (k,v) -> (v, \_)
      2. Reducer - (k, \_) -> (k, ‘’)
   4. The problem with this approach is that it does not take advantage of Hadoop’s parallel processing, so it may be inefficient on large jobs.
3. Using multiple reducers is much harder.
   1. Goal: Sort a large set of data by a sort key but do it in parallel with multiple reducers.
   2. This pattern produces a job composed of two major phases: analysis phase and sorting phase.
   3. Analyze Phase - sample data to build partition ranges for the data to build a set of partitions divided by the ranges of values that will build equal size subsets of data.
      1. The analyze phase is a random sampling of data and then the partition set is based upon that random sampling.
      2. Ahead of time, figure out how large of a sample you may need
      3. The mapper does simple random sampling outputting sort key as its output value - data will show up sorted at the reducer
      4. Only one reducer is used, and final output is totally sorted list
      5. The final list gets sliced into data range boundaries which will become partitions
   4. Sorting Phase - custom partitioner is used to partition data based upon the sort key
      1. Also called the Sorted List phase
      2. The order phase is a relatively straightforward application of MapReduce that uses a custom partitioner
      3. The mapper extracts the sort key but writes entire record as the value
      4. A custom partitioner (TotalOrderPartitioner) takes the data ranges from the partition file produced in the previous step and decides which reducer to send the data to.
      5. The reduce function simply takes the values that have come in and outputs them. (Remember, they are coming to each Reducer already sorted by the shuffle/sort.)
      6. The number of reducers needs to be equal to the number of partitions for the TotalOrderPartitioner to work properly.
4. Secondary Sort
   1. Again, the list of values sent to the reducer with a particular key is not sorted.
      1. Moreover, the order of values mapped to that key will change from one run to the next.[[14]](#footnote-14)
      2. Thus, most MapReduce programs are written so as not to depend on the order values passed to the reduce function with a particular key.[[15]](#footnote-15)
      3. However, it is possible to impose order on the values by sorting and grouping the keys in a particular way.
   2. Motivation: Sometimes, you want to sort by components of the key and the value of the record. For instance, you have a person’s last name as the key but you want to sort by Last Name, and then by First Name.[[16]](#footnote-16)  
      
   3. “In reducer” versus “value to key”[[17]](#footnote-17)
      1. Hadoop automatically sorts data emitted by the mappers prior to sending this to the reducers - during shuffle-sort phase.
      2. We can take advantage of this sort phase for secondary sort.
      3. We have two options here:
         1. Have reducer buffer all values for given key and do “in-reducer” sort
            1. Might be the faster processing
            2. But since reducer gets all values for given key, could run out of memory
         2. Create a composite key to natural key: “value to key” approach
            1. offloads sorting to the MapReduce framework
            2. Hadoop designed to do this
            3. no risk of running out of memory
   4. Components of the Value to Key Secondary Sort
      1. Mapper emits **a composite key** made up of the natural key and part of the value.
      2. Thus, we need a custom partitioner to partition data **by the natural key**.
         1. Otherwise records with the same natural key will go to different reducers.
         2. In the case of the max temperature problem, we want all records for the same year going to the same reducer.
         3. For the sort-by-last-name-then-first-name problem we want all records with the same last name going to the same reducer.
      3. Sort Comparator should sort by the composite key.
      4. Grouping Comparator is going to **group by the natural key** for the reduce() calls.
         1. Otherwise, the keys would be grouped by the composite key, which is not what you want.
         2. In the max temperature problem, we want them grouped by year.
   5. Mapper - Creating the Composite Key
      1. The composite key will be necessary for the actual comparison for sorting. Intermediate key should be a composite of the natural key and something from the value.  
           
         let map(k,v) = emit( new Pair(v.getPrimaryKey(), v.getSecondaryKey() ), v)



* + 1. SR
  1. Partitioner - use natural key to determine which reducer  
       
     let getPartition(Pair k, Text v, int numReducers) = return(k.getPrimaryKey().hashCode % numReducers)  
     
  2. Sorting Composite Keys
     1. Comparator classes are classes that compare objects
     2. compare(A,B) returns
        1. 1 if A > B
        2. 0 if A = B
        3. -1 if A < B
     3. Custom comparators can be used to sort composite keys
        1. Extend WritableComparator
        2. Override int compare()
     4. Two comparators are required.
        1. Sort Comparator
        2. Group Comparator
  3. Sort Comparator
     1. The Sort Comparator will sort the input to the Reducer.
     2. It uses the full composite key, using first the natural key, and then the secondary key if natural keys are equal.

let compare (Pair k1, Pair k2) = compare k1.getPrimaryKey(), k2.getPrimaryKey()

if equal then  
 compare k1.getSecondaryKey(), k2.getSecondaryKey()

Figure : Psuedocode

Johnson#8899 > Johnson#7231

Addams#4543 < Jones#3245

Pikachu#3422 < Pikachu#4399

Rarebit#1244 > Rarebit#007

Figure : Result of Sort Comparator sorting

* 1. Grouping Comparator
     1. The Grouping Comparator uses only the natural key in order to determine which keys and values are passed in a single call to the Reducer
     2. let compare(Pair k1, Pair k2) = compare k1.getPrimaryKey(), k2.getPrimaryKey() results in:  
        Johnson#8899 = Johnson#7231  
        Addams#4569 < Jones#1972
  2. Setting Comparators in Secondary Sort Job Driver
     1. Set both a grouping comparator and sort comparator in your driver class.
     2. Example:

public class MyDriver extends Configured implements Tool {

    public int run(String[] args) throws Exception {

        job.setSortComparatorClass(LastNameAndYearComparator.class);

        job.setGroupingComparatorClass(LastNameComparator.class);

    }

}

### Searching

1. Searching in Map-Reduce basically equates to a filter type operation
2. Assumptions
   1. The input is a set of files containing lines of text.
   2. The mapper has been passed a pattern to look for in the files.
3. Algorithm
   1. Mapper compares the line against the pattern
   2. If the pattern matches, then the Mapper outputs something like:
      1. emit (line, \_) OR
      2. emit (filename + line, \_)
   3. Else emit nothing
4. The reducer is simply the identity reducer that outputs each intermediate key

### Indexing

1. Inverted Index
   1. In computer science, an inverted index is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents.
   2. The purpose of an inverted index is to allow fast full text searches, at a cost of increased processing when a document is added to the database.
   3. Common Uses
      1. Document retrieval systems
      2. Search Engines
      3. Bioinformatics - searching DNA fragments against reference DNA sequence
   4. Algorithm
      1. Mapper: For each word in the line, emit (word, filename)
      2. Reducer
         1. Identify function
            1. Collect together all values for a given key (example: all filenames for a particular word)
            2. Emit (word, filename\_list)
         2. SR
   5. Inverted index data flow  
      

### Joins

1. In a relational database, joining sets of data together is useful and a common operation.
2. In a relational database, joining sets of data together is easy.
3. In Hadoop, joining sets of data together is a common operation as well.
4. In Hadoop, programming a map-reduce job to perform a join is NOT easy
5. Therefore:
   1. Avoid writing a map-reduce job to perform a join.
   2. Better to use Hive, Pig, or another tool to join data together.
6. Map-side Join
   1. Algorithm
      1. Load the smaller set of data into memory - typically using a hash table.
      2. The key of the hash table is your join key.
      3. Map over the other set of data and perform a lookup on the hash table using the join key.
      4. If the join key is found, then join the record and emit the record.
   2. Map side joins have obvious scalability issues. Avoid this method unless you have a very small amount of the in-memory data set.
   3. They tend to be faster than reduce-side joins, however.
   4. This one would generally be seen in the MR job chain.
7. Reducer-side Join
   1. This way is better than the Map-side join
   2. Algorithm
      1. Map over both sets of data.
      2. Emit a key/value pair for each record where key is the join key and value is the entire record.
      3. In the Reducer, perform the actual join.
   3. Note: Hadoop’s shuffle and sort will guarantee that values with the same key are brought together
   4. Slower than map-side joins, but more flexible. Can be applied to almost all scenarios.
   5. MultipleInput
8. Key points
   1. Joins are usually best done using Impala, Hive, or Pig
   2. Map-side joins are simple, but they don’t scale well
   3. Use reduce-side joins when both datasets are large.
      1. Mapper
         1. Merges both data sets into a common record type
         2. Use a composite key (custom WritableComparable) with join on key/record type
      2. Shuffle and Sort
         1. Secondary sort so that ‘primary’ records are processed first
         2. Cusom Partitioner to ensure records sent to correct Reducer OR …
         3. Hack the hashCode of the composite key
      3. Reducer
         1. Group by join key (custom grouping comparator)
         2. Write out ‘secondary’ records joined with ‘primary’ record data

# Hadoop Ecosystem

## Sqoop

1. Imports data from RDBMS to HDFS and vice versa.
   1. Uses MapReduce to import the data.
      1. No reducer is used, however.
      2. Throttles the number of mappers to avoid overloading the cluster.
   2. Can import just one table, all tables in a database, or just part of a table (i.e., supports the WHERE clause).
   3. Can import data into HDFS either as a comma-delimited text file (the default) or as SequenceFiles.
      1. Delimiters can be specified explicitly, as well as field-enclosing or escape characters to allow the presence of delimiters in field contents.
      2. SR
   4. Can be used for incremental imports
      1. First import retrieves all rows in a table.
      2. Subsequent imports retrieve all rows created since the last import.
   5. Generates a Java source file which can encapsulate a row of the imported data
      1. Useful for serializing and de-serializing data in subsequent map-reduce jobs.
      2. The import tool generates this file automatically.
      3. You can also generate this file by using the codegen tool:  
           
         > sqoop codegen –connect jdbc:mysql//localhost/hadoopguide –table widgets \  
          –class-name Widget
   6. Sqoop imports a table from a database by running a MapReduce job that extracts rows from the table, then writes the records to HDFS.
   7. Sqoop decides which database driver to use based on the connection URL.
      1. You may still have to download the JDBC driver yourself and install it on your Sqoop client.
      2. You can also specify which driver to use, when Sqoop cannot determine it.
2. Basically a mapper job without a reducer.
3. By default, uses 4 mappers, but this number is configurable.
4. Uses a JDBC interface
   1. Custom connectors exist for various RDBMSs.
      1. These use the RDBMS’s native protocols instead of JDBC.
      2. And that yields better performance.
   2. Custom connectors are not open source, but are free, and can be found at Cloudera’s site.
5. Syntax[[18]](#footnote-18)
   1. sqoop <tool-name> [tool-options]
   2. Tools include:
      1. import
      2. import-all-tables
      3. list-tables
      4. export
      5. create-hive-table
   3. Options include:
      1. --connect
      2. --username
      3. –password
      4. --table
   4. Specify the number of mappers to use with –m.
   5. Examples
      1. Import a table called employees from a database called personnel in a MySQL RDBMS:

$ sqoop import --username gina --password muff3n! \

--connect jdbc:mysql://database.example.com/personnel \

--table employees

* + 1. Import MySQL DB table employees from DB personnel with ID > 1000:

$ sqoop import --username gina --password muff3n! \

-- connect jdbc:mysql://database.example.com/personnel \

--table employees --where “id > 1000”

* + 1. Export files from HDFS /user/gina/data using 5 mappers and inject their contents into the student1 table in the testDb database:[[19]](#footnote-19)

$ sqoop export --connect jdbc:mysql://localhost/testDb \

--table student1 -m 5 --export-dir /uer/gina/data

* 1. SR

## Pig

1. Apache Pig is a tool for data analysis and processing on Hadoop.
2. Pig offers an alternative to writing MapReduce code directly.
   1. You write in “Pig Latin Script”—a data flow language.
   2. Hadoop Pig converts the Pig Latin into one or more MapReduce jobs.
3. Pig supports many features which allow developers to perform data analysis without having to write Java MapReduce code.
   1. Joining data sets
   2. Grouping data
   3. Loading non-delimited data
   4. Creation of user-defined functions one can write in Java.
4. Key Concepts
   1. In relational databases, we are used to dealing with relational data (rows, columns, fields).
   2. Pig uses similar concepts, but with different names:
      1. field - single element
      2. tuple
         1. Collection of values
         2. Analogous to a row in RDBMS
      3. bag
         1. Collection of tuples
         2. Analogous to a group of rows in RDBMS (though not analogous a table).  
            

Figure 1: Example of a bag, multiple rows in a table.

1. Pig has an interactive shell (called Grunt) where you can type statements.
   1. Pig interprets each Pig Latin statement as you type it.
   2. Execution is delayed until output is required.
   3. Useful for ad-hoc data inspection and debugging Pig Latin Scripts.
   4. To start Pig, type pig at the command line.
   5. Useful commands
      1. $ pig -help
      2. $ pig -version
      3. $ pig -execute
      4. $ pig *script.pig*
2. Using Pig. (There are 3 ways.)
   1. Scripts
      1. Pig can run a script file that contains Pig commands: e.g., pig script.pig
      2. For short scripts, you can use the –e option to run a string as a script on the command line.
   2. Grunt
      1. You start Grunt by calling pig without a file and without the –e option.
      2. You can also run Pig scripts from within Grunt using run or exec.
   3. Embedded
      1. You can run Pig programs from Java using the PigServer class, much like using JDBC to run SQL from Java.
      2. For programmatic access to Grunt, use the PigRunner class.

### Pig Latin

1. Pig Latin is not like a traditional programming language
2. Pig Latin is a data flow language - like a data workflow
   1. The flow of data is expressed as a sequence of statements.
   2. A Pig Latin program consists of a collection of statements, which can be thought of as operations or commands: e.g., “ls” or “group records by year”.
   3. Thus, it’s like a shell script or SQL script.
3. Statements usually end in semicolons.
   1. Some statements like the ls command, however, do not.
   2. As a guideline, statements or commands that are for interactive use in Grunt (e.g., interactive Hadoop commands or diagnostic operators like DESCRIBE) do not require semicolons.
   3. It is never an error to add a semicolon, so when in doubt, add one.
   4. Statements that end with a semicolon can span multiple lines for readability.
4. Comments come in 2 forms
   1. The double hyphen (--) for single-line comments.
   2. The /\* delimiters (C-style comments) for multi-line comments.
5. Operators and commands are not case-sensitive, but aliases and functions are.
6. The typical Pig Latin script starts by loading one or more datasets into bags, and then creates new bags by modifying those it already has.
7. As a Pig Latin script is executed, each statement is processed in turn.
   1. If there are errors, the interpreter halts and displays an error message.
   2. As it reads the script, the interpreter builds a logic plan for every operation, forming the core of a Pig Latin program.
      1. No data processing takes place at this point.
      2. The whole data flow must be defined before any processing can take place.
      3. The trigger for Pig to start execution is the DUMP statement.
         1. Only then is the program compiled into a physical plan (a series of MapReduce jobs) and executed.
         2. SR
8. Pig Latin lacks native control flow statements.
   1. That is, no conditionals or loops.
   2. This is by design.
   3. The recommended approach for writing programs that have conditional logic or loop constructs is to embed Pig Latin in another language, such as Python, JavaScript, or Java, and manage the control flow from there. [[20]](#footnote-20)

#### Pig Latin Operators



#### Pig Latin Diagnostic Operators



#### Pig Latin Macro and UDF Statements



#### Pig Latin Commands



* Note that commands do not process relations; therefore, they are not added to the logical plan and are executed immediately.[[21]](#footnote-21)
* Pig provides commands for interacting with the Hadoop filesystem, which is handy for moving data around before or after processing with Pig, and with MapReduce. They are very similar to the Hadoop fs commands.
* You can access all of the Hadoop fs commands with Pig’s fs.
  + For example, “fs –ls” will show you a file listing. “fs –help” will show you the available Hadoop commands.

#### Pig Latin Expressions



#### Pig Latin Functions



#### Pig Latin Types



## Hive

1. Introduction
   1. Apache Hive is another tool to manipulate data
   2. Hive is a higher level of abstraction on top of map-reduce
   3. Hive uses an SQL-like language called HiveQL
   4. Hive Generates map-reduce jobs that run on the Hadoop cluster
   5. Hive was originally developed by Facebook but is now an Apache open-source project
2. Hive Components
   1. Hive runs on the client machine.
   2. Converts HiveQL queries into map-reduce jobs that run on the cluster.
3. Hive more DB like than Pig, which runs on files
   1. Hive queries operated on tables like a traditional RDBMS
      1. A table is an HDFS directory containing one or more files.
      2. Hive supports many formats for data storage and retrieval.
   2. The user must specify the structure and location of tables when the tables are created.
   3. Hive stores table information in its “metastore” which is contained in an RDBMS such as MySQL.
   4. When performing a query:
      1. Hive first consults the metastore to get table information.
      2. Hive then queries the tables.
4. Hive Data
   1. Hive data is stored in Hive’s warehouse directory in HDFS
      1. Default path: /user/hive/warehouse/<table\_name>
      2. SR
   2. Tables represent subdirectories of the warehouse directory
   3. It is possible to create external tables if the data is already in HDFS and should not be moved from its current location
   4. Actual data is stored in flat files
      1. Control character-delimited text, or SequenceFiles
      2. Can be in arbitrary format with the use of a custom Serializer / Deserializer
      3. All data in a directory is considered to be part of the table data
5. Using the Hive shell
   1. While Pig is more batch oriented, Hive is meant to be interactive
   2. hive will get you into the shell, quit will get you out.
   3. Accessing Hive from the command line
      1. Can execute a file containing HiveQL code using -f option
      2. Can use HiveQL directly from the command line using the -e option
      3. Use the -S (silent) option to suppress informational messages.
         1. This can also be used with the -e or -f options
         2. SR
6. Loading Data Into Hive Tables
   1. Data is loaded into Hive with the LOAD DATA INPATH statement
      1. Assumes that the data is already in HDFS
      2. Example:   
         LOAD DATA INPATH “cust\_data” INTO TABLE customers;
   2. If the data is on the local filesystem, use LOAD DATA LOCAL INPATH
      1. This automatically loads the data into HDFS in the correct directory
      2. SR
7. Loading data into Hive
   1. You can use the Sqoop --hive-import which will automatically create a Hive table from the imported data.
      1. Imports the data.
      2. Generates the Hive CREATE TABLE statement based on the table definition in the RDBMS.
      3. Runs the statement.
   2. SR
8. User-defined functions
   1. Hive supports manipulation of data via User-Defined Functions (UDFs) which are written in Java.
   2. Hive also supports user-created scripts written in any language via the TRANSFORM operator.
   3. Leverages Hadoop Streaming[[22]](#footnote-22)
9. Limitations
   1. Hive is “SQL-Like,” so not all standard SQL is supported.
      1. Sub-queries are only supported in the FROM clause.
      2. No correlated sub-queries.
   2. No support for UPDATE.
   3. No support for DELETE.
   4. No support for INSERT on single rows.
   5. No transactions

## Flume

## Impala

1. Introduction
   1. Impala is a high performance SQL engine for ‘big data’
   2. Impala uses SQL
   3. Impala is 10 to 50 times faster than Hive, Pig, or MapReduce
   4. This means that Impala is feasible for interactive queries
   5. Achieves this speed by using a custom execution engine
   6. Does not have to translate query to Map-Reduce
   7. Impala was developed by Cloudera but is now 100% open source under Apache
2. Impala runs on Hadoop clusters  
   

Figure : Impala vs. Hive Architecture

* 1. Data is stored in HDFS
  2. Does not use map-reduce
  3. Uses the same Metastore as Hive
  4. Impala uses a lot of cluster resources

1. Using the Impala shell
   1. Impala built for speedy interactive query
   2. impala-shell will get you into the shell, exit gets you out
   3. SR
2. Limitations[[23]](#footnote-23)
   1. Impala does not currently support some features in Hive
   2. No complex data types (ARRAY, MAP, STRUCT)
   3. No support for a BINARY data type
   4. No custom file and row format (SerDes)
   5. There is no SQL-style authorization (privileges and roles)
3. Choosing between Impala, Hive, and Pig
   1. Use Impala when…
      1. You need near real-time responses to ad hoc queries.
      2. Or you have structured data with a defined schema.
   2. Use Hive or Pig when you need support for custom file types or complex data types
   3. Use Pig when…
      1. You have developers experienced with writing scripts.
      2. Your data is unstructured or semi-structured.
   4. Use Hive when you have very complex and long-running queries.
4. Pig, Hive, and Impala Comparison Chart

|  |  |  |  |
| --- | --- | --- | --- |
| **Features** | **Pig** | **Hive** | **Impala** |
| SQL-based query language | NO | YES | YES |
| Schema | Optional | Required | Required |
| Process data with external scripts | YES | YES | YES |
| Custom file format support | YES | YES | NO |
| Query Speed | Slow | Moderate | Fast |
| Accessible via ODBC/JDBC | NO | YES | YES |

1. Can Pig, Hive, and Impala now replace the traditional RDBMS?
   1. Relational databases are optimized for:
      1. Small amounts of data.
      2. Immediate results.
      3. In-place modification of data.
      4. Transactional support.
   2. Pig, Hive and Impala are optimized for:
      1. Large amounts of read-only data.
      2. Extensive scalability at low cost.
   3. Pig and Hive are better suited for batch processing.
   4. Impala and an RDBMS are better for interactive use.

## Oozie

1. Defines workflows

# Appendix 1: MasterPeace Lab

## Logging In

1. In the terminal, ssh to [pford@lab.ttlgroup.com](mailto:pford@lab.ttlgroup.com)
2. To login to a node:
   1. ssh cdh4-1-<name>0<number>
   2. EX: ssh cdh4-1-data01

## To add a local file to HDFS

1. Use scp to transfer the file to the lab.
2. Then load the file into HDFS using either:
   1. hadoop fs –copyFromLocal <local filename> [destinationDir]
   2. hadoop fs –put <local filename> [destinationDir]

# Appendix 2: Running MapReduce Jobs

1. Package the job in a shaded JAR
   1. The nodes in the cluster will not be guaranteed to have all of the job’s dependencies in their classpaths.
   2. Thus, we need to create an uber JAR to ensure that the job has all of its dependencies.
2. Push the JAR to the cluster and load it in HDFS.
3. Type the following to run the job:  
     
   hadoop jar <jar name>.jar <driver classname> <input file> <output dir>
4. SR

# Appendix 3: Setting Up A MapReduce Project

Maven Dependencies

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.4</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-hdfs</artifactId>

<version>2.6.0</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-auth</artifactId>

<version>2.6.0</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-common</artifactId>

<version>2.6.0</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-mapreduce-client-core</artifactId>

<version>2.6.0</version>

</dependency>

* Tom White’s book mentions hadoop-core, but in Hadoop 2, that has been replaced with hadoop-common and hadoop-mapreduce-client-core.
* MRUnit’s dependencies are mrunit, junit, and mockito.
* You’ll also want the Maven Shade Plugin so that you can create an uber JAR to distribute to the cluster: the job’s dependencies may not be present in the classpaths of the various nodes in the cluster.

# Appendix 4: Hadoop Streaming and Hadoop Pipes

1. Not covered on the exam.
2. Designed to make Hadoop language-independent.
3. Hadoop Streaming
   1. Uses standard Unix I/O
   2. Connects to MR programs written in Python or Ruby.
4. Hadoop Pipes
   1. Uses Socket Programming
   2. Connects to MR programs written in C++.

1. This is explained in more detail below under [HDFS Architecture](#hdfs_architecture), plus HDFS Federation and High Availablity. [↑](#footnote-ref-1)
2. <http://www.dummies.com/how-to/content/input-splits-in-hadoops-mapreduce.html> [↑](#footnote-ref-2)
3. Combiners and partitioners would be used after the mappers and before the shuffle-and-sort stage in the execution flow, and combiners come before partitioners in the flow. [↑](#footnote-ref-3)
4. Tom White, Hadoop: The Definitive Guide, 3rd Edition, O’Reilly, Ch. 2 “Analyzing Data with Hadoop” [↑](#footnote-ref-4)
5. To learn more about configuration, see *Definitive Guide*, Chapter 5. [↑](#footnote-ref-5)
6. Example using configuration - ToolRunner project, AvgWordLength.java [↑](#footnote-ref-6)
7. See the setup method and how it grabs parameter from Configuration in LetterMapper.java [↑](#footnote-ref-7)
8. Hadoop offers a complete API for working with HDFS. See *Definitive Guide* Chapter 3 - The Java Interface. [↑](#footnote-ref-8)
9. Udemy Lecture 21. Also see <http://www.dummies.com/how-to/content/input-splits-in-hadoops-mapreduce.html>. (“The key to efficient MapReduce processing is that, whenever possible, data is processed **locally**—on the slave node where it is stored. [↑](#footnote-ref-9)
10. See the [earlier discussion on reducers](#number_of_reducers) for more information about how many reducers to use in a job. [↑](#footnote-ref-10)
11. For example: Configuration.set(“mapreduce.output.textoutputformat.separator”, “,”); [↑](#footnote-ref-11)
12. *Hadoop: The Definitive Guide*, Ch.2 [↑](#footnote-ref-12)
13. System.err.println(“my message is the variable set to: “ + myVar); [↑](#footnote-ref-13)
14. This happens because the values come from different map tasks that finish at different times from run to run. [↑](#footnote-ref-14)
15. White, Ch. 8, “Sorting.” Of course, the order of keys should remain the same between runs for the same data, but the order of values mapped to each key will vary. [↑](#footnote-ref-15)
16. Another use case is the max temperature problem. [↑](#footnote-ref-16)
17. According to the Udemy video, the preferred solution to the max temperature problem is find it within the reducer, not during the shuffle-and-sort, but it can be done during shuffle-and-sort, using Secondary Sort. [↑](#footnote-ref-17)
18. Knowing Sqoop syntax is very important on the exam. [↑](#footnote-ref-18)
19. Note that the target table must already exist in the database. [↑](#footnote-ref-19)
20. White, Ch.11, “Pig Latin” [↑](#footnote-ref-20)
21. White, Ch.11, “Pig Latin” [↑](#footnote-ref-21)
22. Don’t study Hadoop streaming for the test. [↑](#footnote-ref-22)
23. Apache is addressing these limitations and may include these in an upcoming release of Impala [↑](#footnote-ref-23)