Apache Hadoop

- An open-source project for reliable, scalable, distributed computing: http://hadoop.apache.org/
- Hadoop includes these subprojects:
  - Hadoop Common: Common utilities that support the other Hadoop subprojects
  - HDFS: A distributed file system that provides high throughput access to application data
  - Map-Reduce: A software framework for distributed processing of large data sets on compute clusters

Data Flow

- A MapReduce job:
  - Unit of work that a programmer wants to perform
    - Consists of the input data, the MapReduce program, and configuration information
- Hadoop runs the job by dividing it into tasks
  - map tasks and reduce tasks
  - The tasks are scheduled using YARN and run on nodes in the cluster
- Input
  - A set of key-value pairs
  - **InputFormat** maps an HDFS dataset to $< k, v >$ pairs
  - Hadoop divides the input into fixed size splits
  - Creates one map task for each split, which runs the user-defined map function for each record in the split
  - The goal is to run the map task on a node where the input data resides in HDFS (data locality)
- Map tasks
  - Write their output to the local disk
  - Usually, the input to a single reduce task is the output from many mappers
- Reduce tasks
  - Map outputs must be transferred across the network to the node where the reduce task is running
  - The outputs are merged and then passed to the user-defined reduce function
  - The output of the reduce is usually stored in HDFS
- Output
  - A set of key-value pairs
  - An **OutputFormat** writes $< k'', v'' >$ pairs to HDFS

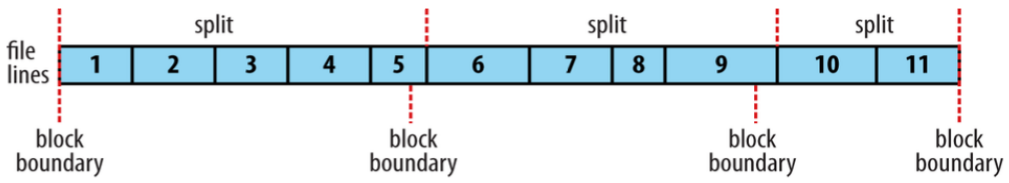
Variations

- I/O
  - Google's MapReduce may use GFS/BigTable for input source and output destination
  - Hadoop can also be integrated with existing MPP (massively parallel processing) relational databases
    - Allows a programmer to write MapReduce jobs over database rows

        - Then output into a new database table
    - In some cases, MapReduce jobs
        - May not consume any input at all
            - E.g., computing π
        - May only consume a small amount of data
            - E.g., input parameters to many instances of processor-intensive simulations running in parallel
- MapReduce
    - MapReduce program with no mappers is not possible
        - Although in some cases it is useful for the mapper to implement the identity function and simply pass input key-value pairs to the reducers
            - Has the effect of sorting and regrouping the input for reduce-side processing
    - MapReduce programs can contain no reducers
        - Mapper output is directly written to disk (one file per mapper)
        - In some cases it is useful for the reducer to implement the identity function
            - The program simply sorts and groups mapper output
    - Finally, running identity mappers and reducers has the effect of regrouping and resorting the input data

Map-Reduce Data

- Key-value pairs
    - Keys and values can be integers, float, strings, or any arbitrary data structures
    - Keys may uniquely identify a record or may be completely ignored
    - Keys can be combined in complex ways to design various algorithms
- Map Reduce algorithm does not work on physical blocks of the file
    - It works on logical input splits
    - HDFS breaks down very large files into large blocks (128MB), and stores three copies of these blocks on different nodes in the cluster
        - Has no awareness of the content of these files
    - Hadoop uses a logical representation of the data stored in file blocks, known as input splits
    - Upper bound for input splits = HDFS blocksize
    - No lower bound, but can be set by programmer: $mapreduce.input.fileinputformat.split.minsize$



Example

- A single file is broken into lines, and the line boundaries do not correspond with the HDFS block boundaries
- Splits honor logical record boundaries (in this case, lines), so we see that the first split contains line 5, even though it spans the first and second block
- The second split starts at line 6
- Each map processes a single split
  - Each split is divided into records, and the map processes each record as a key-value pair in turn
- Splits and records are logical
  - Nothing requires them to be tied to files
- As a MapReduce application writer, you don't need to create splits manually, as they are created by an InputFormat
  - InputFormat is responsible for creating the input splits and dividing them into records
- The output dataset is also stored on HDFS
  - The output may consist of a number of distinct files, equal to the number of reducers
  - The programmer can specify the number of reducers using the *setNumReduceTasks* method (default is 1)

InputFormat

- Describes the input-specification for a Map-Reduce job
  - Provides a factory for RecordReader objects that read the file
- The Map-Reduce framework relies on the InputFormat of the job to
  - Validate the input-specification of the job
  - Split-up the input files into InputSplits
  - Provide the RecordReader implementation to be used to get input records from the InputSplit for processing by the mapper

FileInputFormat

- Base class for all implementations of InputFormat that use files as their data source
- It provides two things:
  - A place to define which files are included as the input to a job
  - Implementation for generating splits for the input files

Text Input

- Hadoop excels at processing unstructured text

TextInputFormat

- Default InputFormat
- Used for plain text files
- Files are broken into lines

- Either linefeed (\n) or carriage-return (\r) are used to signal end of line
- Keys are the position in the file and values are the line of text
- So, a file containing the following text:

  a b c d,
  e f g,

  is divided into one split of two records. The records are interpreted as the following key-value pairs:

  (0, a b c d,)
  (9, e f g,)

Binary Input

SequenceFileInputFormat

- Provides a persistent data structure for binary key-value pairs
  - Also works well as containers for smaller files
- It comes with the *sync*() method to introduce sync points to help managing InputSplits for MapReduce
- They support compression as a part of the format
- They can store arbitrary types using a variety of serialization frameworks

Multiple Inputs

- All of the input is interpreted by a single InputFormat and a single Mapper
- MultipleInputs class
  - Allows you to specify which InputFormat and Mapper to use on a per-path basis

OutputFormat

- For writing the reducer results to the output (HDFS)
- Analogous to InputFormat

TextOutputFormat

- Writes "key value", followed by a newline, to the output file
- Each key-value pair is separated by a tab character
  - Cay be changed using the *mapreduce.output.textoutputformat.separator* property

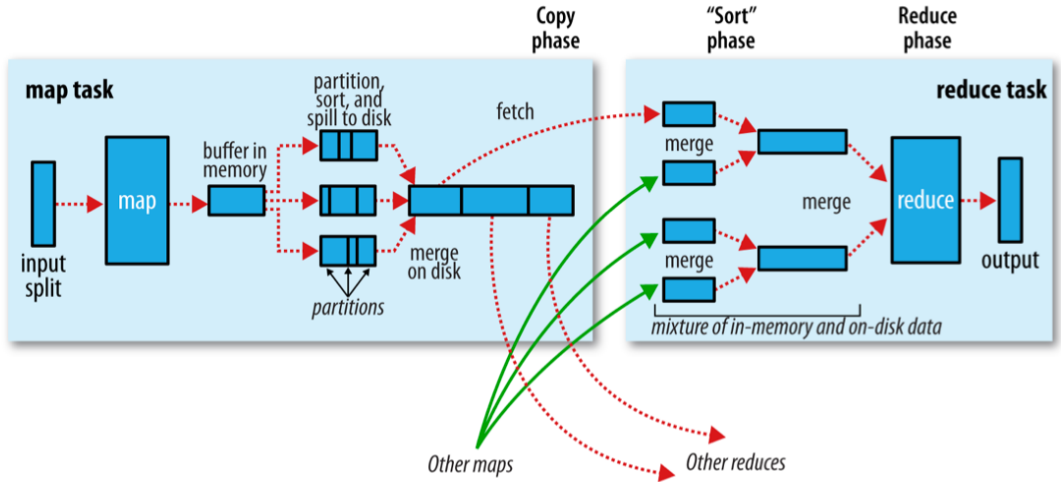
SequenceFileOutputFormat

- Uses a binary format to pack key-value pairs
- This is a good choice of output if it forms the input to a further MapReduce job, since it is compact and is readily compressed
- Compression is controlled via the static methods on SequenceFileOutputFormat

NullOutputFormat discards output

Shuffling:

- The process by which the system performs the sort and transfers the map outputs to the reducers as inputs
- Implicit stage between map and reduce



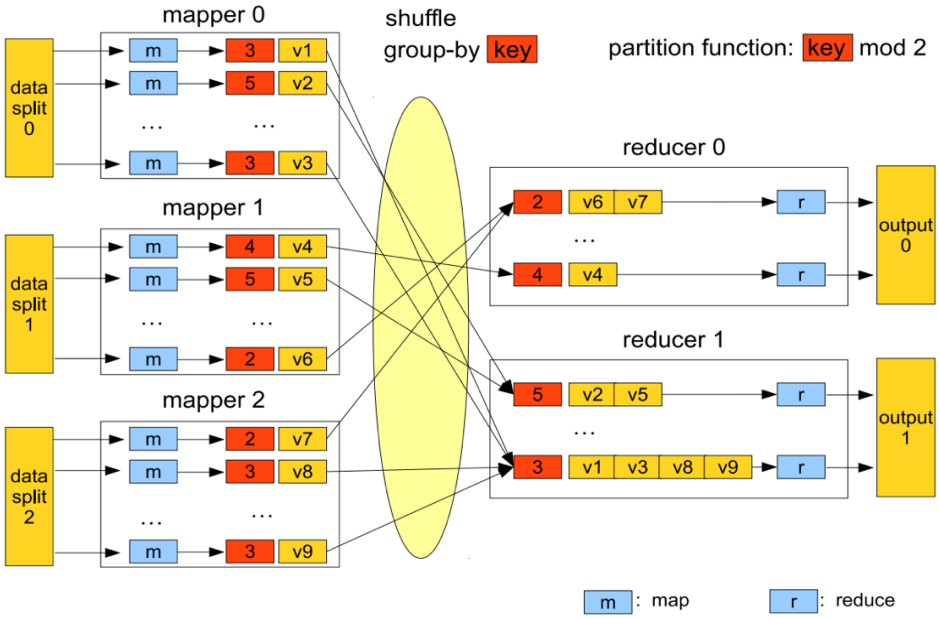
- Each map task has a circular memory buffer that it writes the output to
  - The buffer is 100 MB by default
- When the contents of the buffer reach a certain threshold size a background thread will start to spill the contents to disk
- Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers that they will ultimately be sent to
  - Within each partition, the background thread performs an in-memory sort by key
  - Sorting saves time for the reducer, helping it easily distinguish when a new reduce task should start
  - It simply starts a new reduce task, when the next key in the sorted input data is different than the previous
- Before the task is finished, the spill files are merged into a single partitioned and sorted output file
- The map tasks may finish at different times, so the reduce task starts copying their outputs as soon as each completes in parallel (copy phase)
  - If files are small, copied to JVM memory else on disk
- A background thread continues to merge them into larger sorted files
- Then it sort the merged files (sort phase)
- The number of reduce tasks is not governed by the size of the input, but instead is specified independently
- When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task
- There can be many keys (and their associated values) in each partition, but the *records for any given key are all in a single partition*

- Each reducer is associated with a different key space (a partition)
  - All values for the same key are sent to the same partition
  - Default partitioner: key-hash-value % number-of-reducers

Shuffling Implementation

- Let say that there are m mappers and n reducers
- At the mapper side:
  - Each mapper creates n partitions
  - Each $< k', v' >$ pair produced by the mapper goes to partitioner $k' \% n$
  - Each partition is sorted locally using the sorting function
  - If there is a combiner function, each partition is reduced by combining consecutive pairs with the same key
- At the reducer side
  - Each reducer fetches one partition from each of the m mappers
  - These m partitions are merged in stages
  - The reducer scans the merged data: consecutive pairs with the same key belong to the same group

- An Example with 3 Mappers and 2 Reducers



The *org.apache.hadoop.mapreduce.Mapper* Class

- The Mapper class is a generic type, with four formal type parameters that specify
  - The input key, input value, output key, and output value types of the map function
- Need to define the map method:
  public void map (KEYIN key, VALUEIN value, Context context) throws IOException, InterruptedException {…}
  - Called once for each key/value pair in the input split
- Use **context.write(k, v)** to write the (k, v) pair to the map output
  - Optionally, override these methods:

    public void setup (Context context)
    - Called once at the beginning of the task
    - Can be used to configure parameters that are used during the map operation
    - Can be used set up database connections
    - Can be used to initialize any class-level variables

    public void cleanup (Context context)
    - Called once at the end of the task
    - Can be used for closing connections, releasing resources, performing cleanup activities
    - Can be used for writing some information into the output
      - For instance, it can be used for writing summary data or for calculating and writing final statistics after the map operations

The *org.apache.hadoop.mapreduce.Reducer* Class

- Four formal type parameters are used to specify the input and output types
- The input types of the reduce function must match the output types of the map function
- Need to define the reduce method:
  public void reduce (KEYIN key, Iterable values, Context context) throws IOException, InterruptedException {…}
  - Called once for each key
- The **Iterable** values contain all values associated with the same key
- Also contain optional setup() and cleanup()

org.apache.hadoop.mapreduce.Partitioner Class

- Allows to implement a custom partitioner
- Need to override the following function
  getPartition(K key, V value, int numPartitions)

org.apache.hadoop.mapreduce.Job Class

- It allows the user to configure the job, submit it, and monitor its progress
- Represents a packaged Hadoop job for submission to cluster
- Need to specify input and output paths
- Need to specify input and output formats
- Need to specify mapper, reducer, combiner, partitioner classes
- Need to specify intermediate/final key/value classes
- Need to specify number of reducers
  - But not mappers!
- Default values are not practical!

setJarByClass()

- Rather than explicitly specifying the name of the JAR file, we can pass a class in the Job's *setJarByClass*() method, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class

setInputPaths(), *addInputPath*()

- An input path is specified by calling these methods, and it can be a single file, a directory (in which case, the input forms all the files in that directory), or a file pattern
- As the name suggests, *addInputPath*() can be called more than once to use input from multiple paths

- The output path (of which there is only one) is specified by the static *setOutputPath*() method on FileOutputFormat
  - It specifies a directory where the output files from the reduce function are written
  - The directory shouldn't exist before running the job because Hadoop will complain and not run the job
- Map and reduce are set using *setMapperClass*() and *setReducerClass*() methods
- The *setOutputKeyClass*() and *setOutputValueClass*() methods control the output types for the reduce function, and must match what the Reduce class produces
  - The map output types default to the same types, so they do not need to be set if the mapper produces the same types as the reducer
  - However, if they are different, the map output types must be set using the *setMapOutputKeyClass*() and *setMapOutputValueClass*() methods
- The input types are controlled via the input format, which we have not explicitly set because we are using the default TextInputFormat
- After setting the classes that define the map and reduce functions, we are ready to run the job

- The *waitForCompletion*() method on Job submits the job and waits for it to finish
  - The single argument to the method is a flag indicating whether verbose output is generated
    - When true, the job writes information about its progress to the console
  - The return value of the *waitForCompletion*() method is a Boolean indicating success (true) or failure (false), which we translate into the program's exit code of 0 or 1
- Here is an example on how to submit a job:

```
// Create a new Job
Job job = Job.getInstance ();
job.setJarByClass(MyJob.class);
// Specify various job-specific parameters job.setJobName("myjob");
job.setInputPath(new Path("in"));
job.setOutputPath(new Path("out"));
job.setMapperClass(MyJob.MyMapper.class);
job.setReducerClass(MyJob.MyReducer.class);
// Submit the job, then poll for progress until the job is complete
job.waitForCompletion(true);
```

Serialization

- Need to be able to transform any object into a byte stream
  - Transmit data over the network (RPC)
  - Store data on HDFS
- Hadoop uses its own serialization interface
- Provides a number of **WritableComparable** classes
  - **IntWritable, DoubleWritable, Text,** etc
  - **Text** is a **Writable** String
- Custom values must implement **Writable**
  - Two methods
    - **write()** for serialization
      - For writing its state to a **DataOutput** binary stream
    - **readFields()** for deserializaiton
      - For reading its state from a **DataInput** binary stream

```
class MyClass implements Writable {
      public void write(DataOutput out) throws IOException {…}
      public void readFields(DataInput in) throws IOException {…}
}
```

- Custom keys must implement **WritableComparable**
  - It is a **Writable** with the additional method:

    ```
    public int compareTo (MyClass o) { … }
    ```
  - Why? because must be able to sort data by the key
  - If you want to use a custom serializer, you may specify your own implementation of

org.apache.hadoop.io.serializer.Serialization and set *io.serialization* in the Hadoop configuration

Serialization Example

```
class Employee implements Writable {
   public String name;
   public int dno;
   public String address;

   public void write (DataOutput out) throws IOException {
         out.writeInt (dno);
         out.writeUTF(name);
         out.writeUTF(address);
   }

   public void readFields ( DataInput in ) throws IOException {
         dno = in.readInt ();
         name = in.readUTF();
         address = in.readUTF();
   }
}
```

Note:

- When you access values, you get the same object but with different values
  - To reduce the number of objects being created and garbage collected
- If you want to add all the values to a vector v, if you just use $v.add(a)$ in your loop, you're adding the same object to the vector over and over again
- Since Hadoop reuses this object for each value, at the end you will have a vector with n copies of the last value that you read
- Wrong:

```
Vector v = new Vector();
for (VALUEIN a: values)
      v.add(a);
```

- By using $v.add(a.clone())$, you're creating a new copy of the value each time you add it to the Vector, so each value is preserved properly

```
Vector v = new Vector();
for (VALUEIN a: values)
      v.add(a.clone());
```

- Or you can create a new instance of `a` and then add it the vector `v`

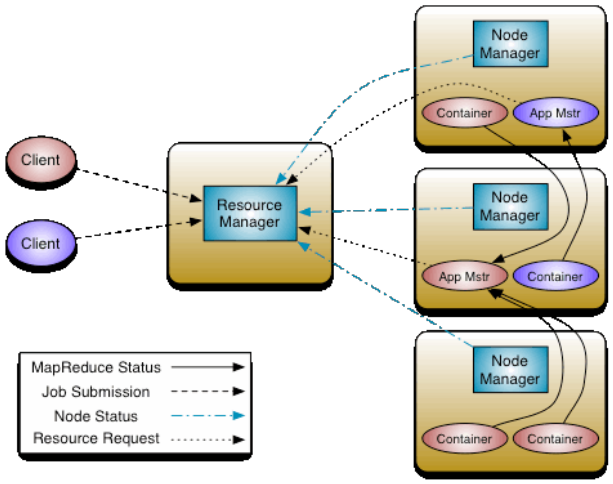
Chaining Multiple Map-Reduce Jobs Together

```
public class MyClass extends Configured implements Tool {
      @Override
      public int run (String [] args) throws Exception {
            Configuration conf = getConf();
            Job job = Job.getInstance(conf, ”Job1”);
            …
            job.waitForCompletion(true);
            Job job2 = Job.getInstance(conf, ”Job2”);
            …
            job2.waitForCompletion(true);
            return 0;
      }
      public static void main (String [] args) throws Exception {
            ToolRunner.run(new Configuration(), new MyClass(),args);
      }
}
```

Hadoop Java API:
https://hadoop.apache.org/docs/stable/api/index.html

Apache Hadoop YARN

- Split up the functionalities of resource management and job scheduling/monitoring into separate daemons
  - ResourceManager (*RM*)
  - ApplicationMaster (*AM*)



ResourceManager (*RM*)

- The ResourceManager allocates resources among all the applications in the system
  - The NodeManager is the per-machine framework
    - Responsible for containers, monitoring their resource usage (cpu, memory, disk, network)
    - Reporting to the ResourceManager/Scheduler
- Components:
  - Scheduler
    - Allocates resources to the running applications subject to constraints of capacities, queues etc.
  - ApplicationsManager
    - Accepts job-submissions, negotiates the first container for executing the application specific ApplicationMaster
    - Provides the service for restarting the ApplicationMaster container on failure

ApplicationMaster (*AM*)

- An application is either a single job or a DAG of jobs
- Negotiates resources from the ResourceManager
- Works with the NodeManager(s) to execute and monitor the tasks