YAHOO!

HowTo Hadoop

Devaraj Das

Hadoop

http://hadoop.apache.org/core/

Hadoop Distributed File System

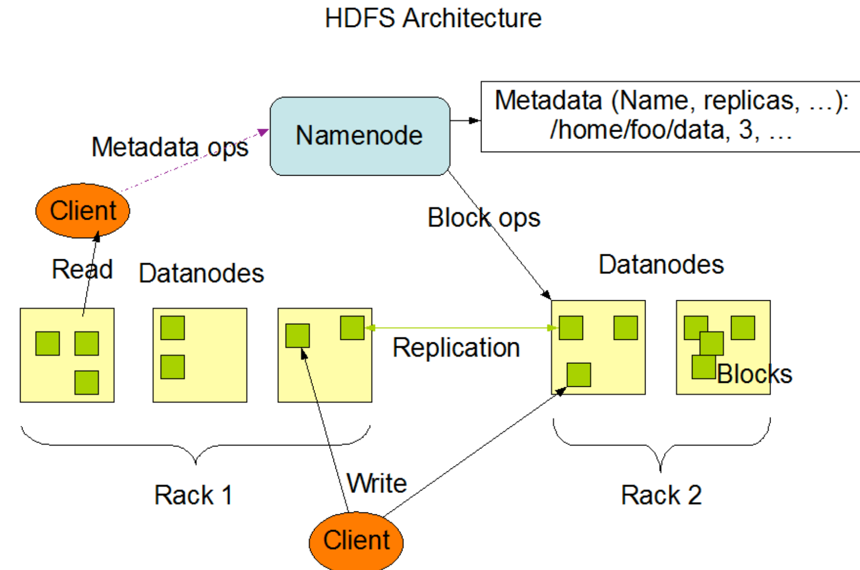
- Fault tolerant, scalable, distributed storage system
- Designed to reliably store very large files across machines in a large cluster
- Data Model
  - Data is organized into files and directories
  - Files are divided into uniform sized blocks and distributed across cluster nodes
  - Blocks are replicated to handle hardware failure
  - Filesystem keeps checksums of data for corruption detection and recovery
  - HDFS exposes block placement so that computes can be migrated to data

HDFS Architecture

- Master-Worker architecture
- HDFS Master "Namenode"
  - Manages the filesystem namespace
  - Controls read/write access to files
  - Manages block replication
  - Checkpoints namespace and journals namespace changes for reliability
- HDFS Workers "Datanodes"
  - Serve read/write requests from clients
  - Perform replication tasks upon instruction by Namenode

HDFS Terminology

- Namenode
- Datanode
- DFS Client
- Files/Directories
- Replication
- Blocks
- Rack-awareness



Block Placement

- Default is 3 replicas, but settable
- Blocks are placed (writes are pipelined):
  - On same node
  - On different rack
  - On the other rack
- Clients read from closest replica
- If the replication for a block drops below target, it is automatically re-replicated.

Data Correctness

- Data is checked with CRC32
- File Creation
  - Client computes checksum per 512 byte
  - DataNode stores the checksum
- File access
  - Client retrieves the data and checksum from DataNode
  - If Validation fails, Client tries other replicas

Interacting with HDFS

```
hadoop fs [-fs <local | file system URI>] [-conf <configuration file>]
        [-D <property=value>] [-ls <path>] [-lsr <path>] [-du <path>]
        [-dus <path>] [-mv <src> <dst>] [-cp <src> <dst>] [-rm <src>]
        [-rmr <src>] [-put <localsrc> <dst>] [-copyFromLocal <localsrc> <dst>]
        [-moveFromLocal <localsrc> <dst>] [-get <src> <localdst>]
        [-getmerge <src> <localdst> [addnl]] [-cat <src>]
        [-copyToLocal <src><localdst>] [-moveToLocal <src> <localdst>]
        [-mkdir <path>] [-report] [-setrep [-R] [-w] <rep> <path/file>]
        [-touchz <path>] [-test -[ezd] <path>] [-stat [format] <path>]
        [-tail [-f] <path>] [-text <path>]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-chgrp [-R] GROUP PATH...]
        [-help [cmd]]
```

Interacting with the HDFS

- Uploading files
  - hadoop fs -put foo mydata/foo
  - cat ReallyBigFile | hadoop fs -put - mydata/ReallyBigFile
- Downloading files
  - hadoop fs -get mydata/foo foo
  - hadoop fs -get - mydata/ReallyBigFile | grep “the answer is”
  - hadoop fs -cat mydata/foo
- File Types
  - Text files
  - SequenceFiles
    - Key/Value pairs formatted for the framework to consume
    - Per-file type information (key class, value class)

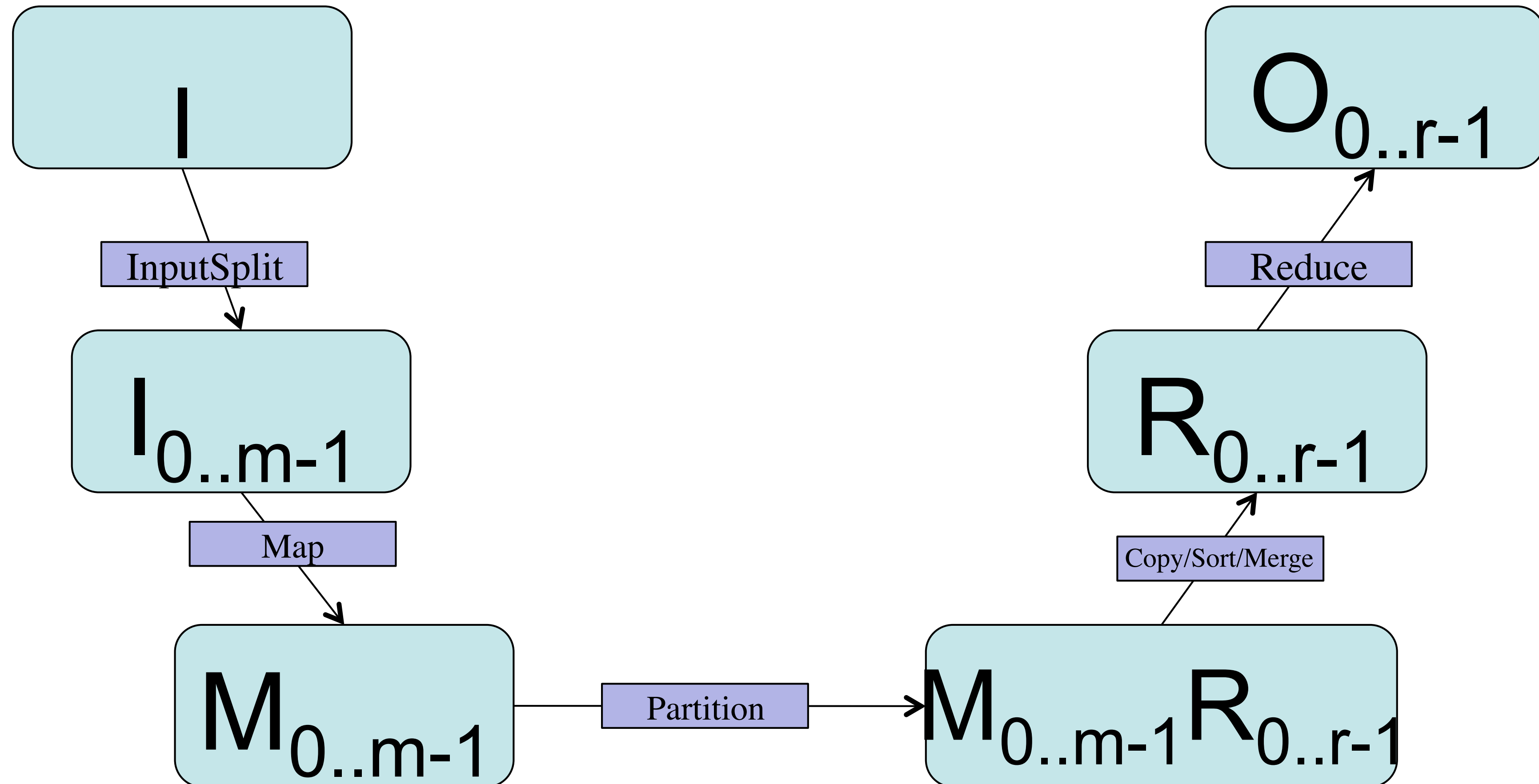
HDFS API

- Most common file and directory operations supported:
  - create, open, close, read, write, seek, tell, list, delete etc.
- Files are write once and have exclusively one writer
  - Append/truncate coming soon
- Some operations peculiar to HDFS:
  - set replication, get block locations
- Owners, permissions supported now !

Hadoop Map/Reduce

- Simple: Transform & Aggregate
  - But, Sort/merge before aggregate is (almost) always needed
  - Operates at transfer rate
- Simple programming metaphor:
  - input | map | shuffle | reduce > output
  - cat * | grep | sort | uniq -c > file
- Pluggable user code runs in generic reusable framework
  - A natural for log processing, great for most web search processing
  - A lot of SQL maps trivially to this construct (see PIG)
- Distribution & reliability
  - Handled by framework
- Several interfaces:
  - Java, C++, text filter (a.k.a. Streaming)

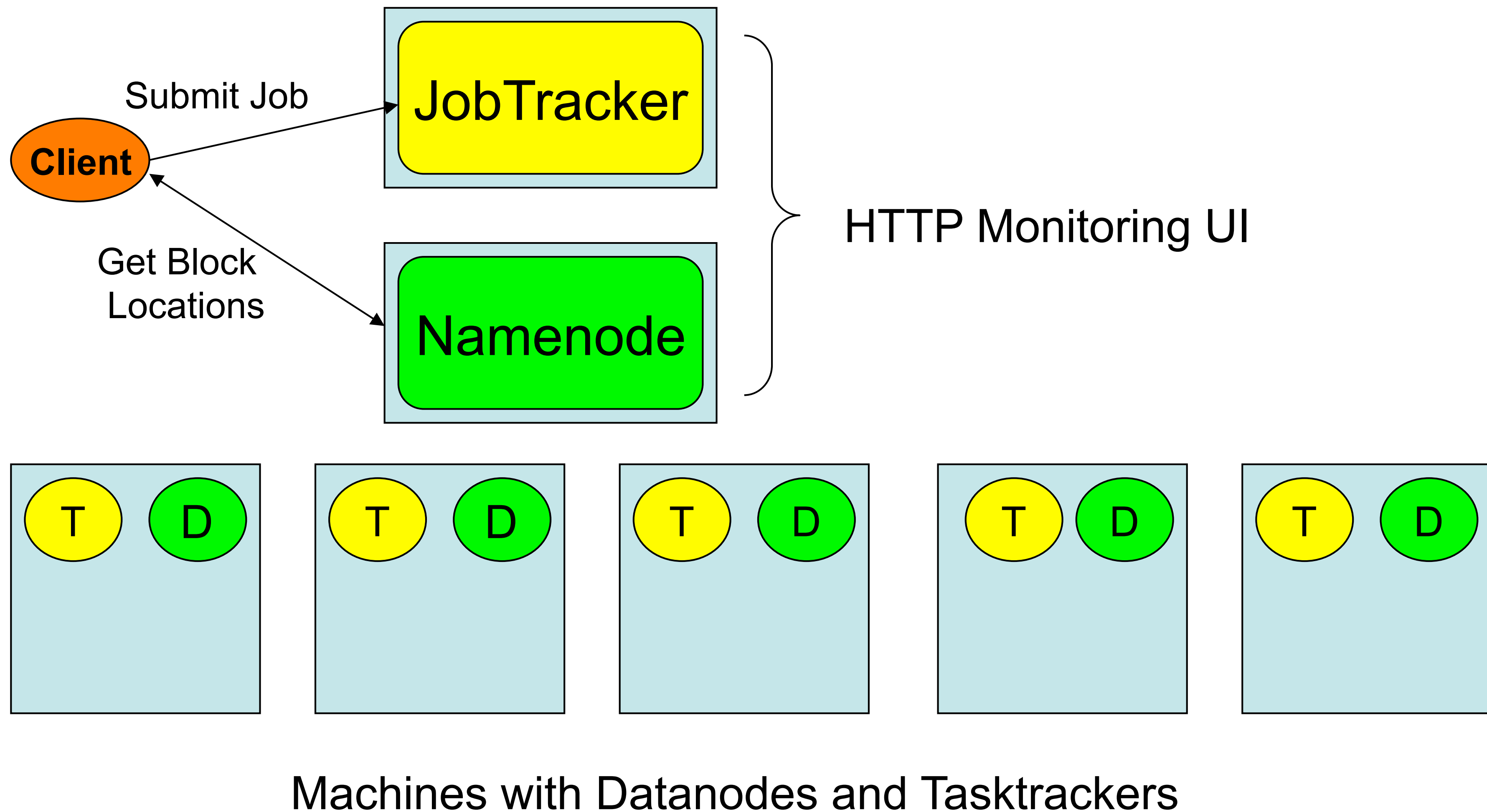
Hadoop MR Dataflow



Hadoop MR - Terminology

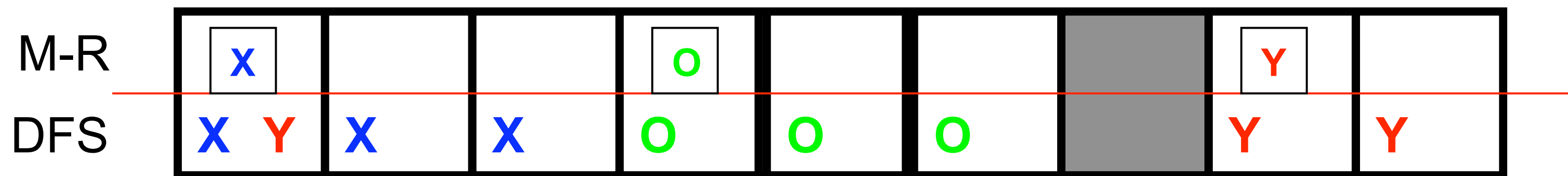
- Job
- Task
- JobTracker
- TaskTracker
- JobClient
- Splits
- InputFormat/RecordReader

Hadoop HDFS + MR cluster



Hadoop: Two Services in One

Cluster Nodes run both DFS and M-R



WordCount: Hello World of Hadoop

- Input: A bunch of large text files
- Desired Output: Frequencies of Words

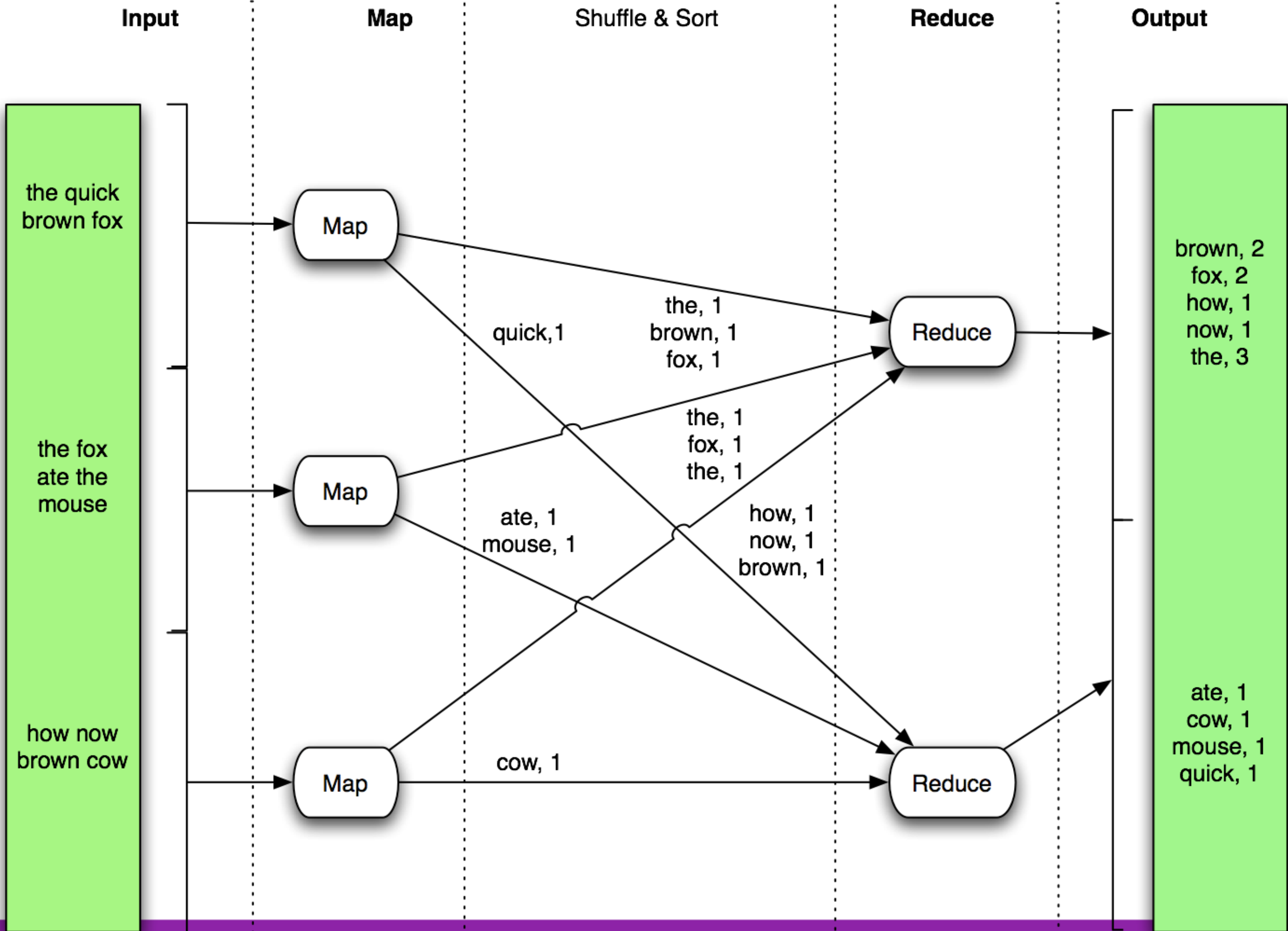
Word Count Example

- Mapper
  - Input: value: lines of text of input
  - Output: key: word, value: 1
- Reducer
  - Input: key: word, value: set of counts
  - Output: key: word, value: sum
- Launching program
  - Defines the job
  - Submits job to cluster

Map Output -> Reduce Input

- Map output is stored across local disks of task tracker
- So is reduce input
- Each task tracker machine also runs a Datanode
- In our config, datanode uses “upto” 85% of local disks
- Large intermediate outputs can fill up local disks and cause failures
  - Non-even partitions too

Word Count Dataflow



Configuring a Job

- Jobs are controlled by configuring *JobConfs*
- JobConfs are maps from attribute names to string value
- The framework defines attributes to control how the job is executed.

```
conf.set("mapred.job.name", "MyApp");
```

- Applications can add arbitrary values to the JobConf

```
conf.set("my.string", "foo");
conf.setInteger("my.integer", 12);
```

- JobConf is available to all of the tasks

Putting it all together

- Create a launching program for your application
- The launching program configures:
  - The *Mapper* and *Reducer* to use
  - The output key and value types (input types are inferred from the *InputFormat)*
  - The locations for your input and output
- The launching program then submits the job and typically waits for it to complete

Putting it all together

```
public class WordCount {
……
public static void main(String[] args) throws IOException {
    JobConf conf = new JobConf(WordCount.class);
    // the keys are words (strings)
    conf.setOutputKeyClass(Text.class);
    // the values are counts (ints)
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(MapClass.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputPath(new Path(args[0]);
    conf.setOutputPath(new Path(args[1]);
    JobClient.runJob(conf);
…..
```

Some handy tools

- Input/Output Formats
- Partitioners
- Combiners
- Compression
- Counters
- Speculation
- Zero reduces
- Distributed File Cache
- Tool

Input and Output Formats

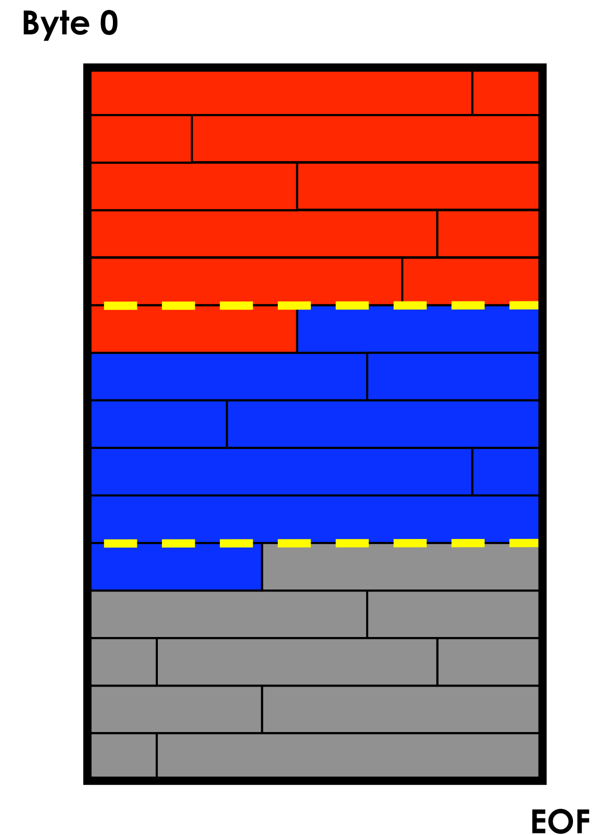
- A Map/Reduce may specify how it's input is to be read by specifying an *InputFormat* to be used
- A Map/Reduce may specify how it's output is to be written by specifying an *OutputFormat* to be used
- These default to *TextInputFormat* and *TextOutputFormat,* which process line-based text data
- Another common choice is SequenceFileInputFormat and SequenceFileOutputFormat for binary data
- These are file-based, but they are not required to be

Input -> InputSplits

- Input specified as collection of paths (typically on HDFS)
- JobClient asks the specified InputFormat to provide description of splits
- Default: FileSplit
  - Each split is approximately DFS's block
    - mapred.min.split.size overrides this
  - Gzipped files are not split
  - A "split" does not cross file boundary
- Number of Splits = Number of Map tasks

InputSplit -> RecordReader

- Record = (Key, Value)
- InputFormat
  - TextInputFormat
  - Unless 1st, ignore all before 1st separator
  - Read-ahead to next block to complete last record



How many Maps and Reduces

- Maps
  - Usually as many as the number of HDFS blocks being processed, this is the default
  - Else the number of maps can be specified as a hint
  - The number of maps can also be controlled by specifying the *minimum split size*
  - The actual sizes of the map inputs are computed by:
    - *max(min(block_size, data/#maps), min_split_size)*
- Reduces
  - Unless the amount of data being processed is small
    - *0.95*num_nodes*mapred.tasktracker.tasks.maximum*

Example: Parameter Sweeps

- Usually an external program processes a file based on command-line parameters
- E.g. ./prog in.txt –params="0.1,0.3,0.7"
  - Produces out.txt
- Objective: Run several instances of "prog" for varying parameters over parameter space
- Number of Mappers =Number of different combinations of these parameters

Partitioners

- Partitioners are application code that define how keys are assigned to reduces
- Default partitioning spreads keys evenly, but randomly
  - Uses *key.hashCode() % num_reduces*
- Custom partitioning is often required, for example, to produce a total order in the output
  - Should implement *Partitioner* interface
  - Set by calling `conf.setPartitionerClass(MyPart.class)`
  - To get a total order, sample the map output keys and pick values to divide the keys into roughly equal buckets and use that in your partitioner

Partitioner

- Default partitioner evenly distributes records
  - hashcode(key) mod NR
- Partitioner could be overridden
  - When Value should also be considered
    - a single key, but values distributed
  - When a partition needs to obey other semantics
    - Al URLs from a domain should be in the same file
- Interface Partitioner
  - int getPartition(K, V, nPartitions)

Producing Fully Sorted Output

- By default each reducer gets input sorted on key
- Typically reducer output order is the same as input
- Each part file is sorted
- How to make sure that Keys in part i are all less than keys in part i+1 ?
- Fully sorted output

Fully sorted output (contd.)

- Simple solution: Use single reducer
- But, not feasible for large data
- Insight: Reducer input also must be fully sorted
- Key to reducer mapping is determined by partitioner
- Design a partitioner that implements fully sorted reduce input
  - sample the map output keys and pick values to divide the keys into roughly equal buckets and use that in your partitioner

Performance Analysis of Map-Reduce

- MR performance requires
  - Maximizing Map input transfer rate
  - Pipelined writes from Reduce
  - Small intermediate output
  - Opportunity to Load Balance

Map Input Transfer Rate

- Input locality
  - HDFS exposes block locations
  - Each map operates on one block
- Efficient decompression
  - More efficient in Hadoop 0.18
- Minimal deserialization overhead
  - Java serialization is very verbose
  - Use Writable/Text

A Counter Example

- Bob wanted to count lines in text files totaling several terabytes
- He used
  - Identity Mapper (input copied directly to output)
  - A single Reducer that counts the lines and outputs the total
- What is he doing wrong ?
- This really happened!
- Take home message is that Hadoop is **powerful** and can be dangerous in the wrong hands…

Intermediate Output

- Almost always the most expensive component
  - M * R Transfers over the network
  - Merging and Sorting
- How to improve performance:
  - Avoid shuffling/sorting if possible
  - Minimize redundant transfers
  - Compress

Avoid shuffling/sorting

- Set number of reducers to zero
  - Known as map-only computations
  - Filters, Projections, Transformations
- Beware of number of files generated
  - Each map task produces a part file
  - Make map produce equal number of output files as input files
    - How?

Combiners

- When *maps* produce many repeated keys
  - It is often useful to do a local aggregation following the *map*
  - Done by specifying a *Combiner*
  - Goal is to decrease size of the transient data
  - Combiners have the same interface as Reduces, and often are the same class.
  - Combiners must **not** have side effects, because they run an indeterminate number of times.
  - In *WordCount*, *conf.setCombinerClass(Reduce.class);*

Compression

- Compressing the outputs and intermediate data will often yield huge performance gains
  - Can be specified via a configuration file or set programatically
  - Set *mapred.output.compress* to *true* to compress job output
  - Set *mapred.compress.map.output* to *true* to compress map outputs
- Compression Types *(mapred.output.compression.type)*
  - “block” - Group of keys and values are compressed together
  - “record” - Each value is compressed individually
  - Block compression is almost always best
- Compression Codecs *(mapred(.map)?.output.compression.codec)*
  - Default (zlib) - slower, but more compression
  - LZO - faster, but less compression

Opportunity to Load Balance

- Load imbalance inherent in the application
  - Imbalance in input splits
  - Imbalance in computations
  - Imbalance in partition sizes
- Load imbalance due to heterogeneous hardware
  - Over time performance degradation
- Give Hadoop an opportunity to do load-balancing

Configuring Task Slots

- mapred.tasktracker.map.tasks.maximum
- mapred.tasktracker.reduce.tasks.maximum
- Tradeoffs:
  - Number of cores
  - Amount of memory
  - Number of local disks
  - Amount of local scratch space
  - Number of processes
- Also consider resources consumed by Tasktracker & Datanode

Speculative execution

- The framework can run multiple instances of slow tasks
  - Output from instance that finishes first is used
  - Controlled by the configuration variable *mapred.speculative.execution*
  - Can dramatically bring in long tails on jobs

Performance Summary

- Is your input splittable?
  - Gzipped files are NOT splittable
- Are partitioners uniform?
- Buffering sizes (especially io.sort.mb)
- Do you need to Reduce?
- Only use singleton reduces for very small data
  - Use Partitioners and cat to get a total order
- Memory usage
  - Please do not load all of your inputs into memory!

Counters

- Often Map/Reduce applications have countable events
- For example, framework counts records in to and out of Mapper and Reducer
- To define user counters:

```
static enum Counter {EVENT1, EVENT2};
reporter.incrCounter(Counter.EVENT1, 1);
```

- Define nice names in a MyClass_Counter.properties file

```
CounterGroupName=My Counters
EVENT1.name=Event 1
EVENT2.name=Event 2
```

Deploying Auxiliary Files

- -file auxFile.dat
- Job submitter adds file to job.jar
- Unjarred on the task tracker
- Available as $cwd/auxFile.dat
- Not suitable for more / larger / frequently used files

Using Distributed Cache

- Sometimes, you need to access “side” files (such as “in.txt”)
- Read-only Dictionaries (such as for porn filtering)
- Libraries dynamically linked to streaming mapper/reducer
- Tasks themselves can fetch files from HDFS
  – Not Always ! (Hint: Unresolved symbols)
- Performance bottleneck

Distributed File Cache

- Define list of files you need to download in JobConf
- Add to launching program:

```
DistributedCache.addCacheFile(new URI("hdfs://nn:8020/foo"),
    conf);
```

- Add to task:

```
Path[] files =
    DistributedCache.getLocalCacheFiles(conf);
```

Caching Files Across Tasks

- Specify “side” files via –cacheFile
- If lot of such files needed
  - Jar them up (.tgz coming soon)
  - Upload to HDFS
  - Specify via –cacheArchive
- TaskTracker downloads these files “once”
- Unjars archives
- Accessible in task’s cwd before task even starts
- Automtic cleanup upon exit

Tool

- Handle “standard” Hadoop command line options:
  - -conf file - load a configuration file named file
  - -D prop=value - define a single configuration property prop
- Class looks like:

```
public class MyApp extends Configured implements Tool {
  public static void main(String[] args) throws Exception {
    System.exit(ToolRunner.run(new Configuration(),
                    new MyApp(), args));
  }
  public int run(String[] args) throws Exception {
    .... getConf() ...
  }
}
```

Non-Java Interfaces

- Streaming
- Pipes (C++)
- Pig

Streaming

- What about non-programmers?
  - Can define Mapper and Reducer using Unix text filters
  - Typically use grep, sed, python, or perl scripts
- Format for input and output is: **key \t value \n**
- Allows for easy debugging and experimentation
- Slower than Java programs

```
bin/hadoop jar hadoop-streaming.jar -input in-dir -output out-dir
  -mapper streamingMapper.sh -reducer streamingReducer.sh
```

- Wordcount Mapper: `sed -e 's| |\n|g' | grep .`
- Wordcount Reducer: `uniq -c | awk '{print $2 "\t" $1}'`

Pipes (C++)

- C++ API and library to link application with
- C++ application is launched as a sub-process of the Java task
- Keys and values are std::string with binary data
- Word count map looks like:

```
class WordCountMap: public HadoopPipes::Mapper {
public:
  WordCountMap(HadoopPipes::TaskContext& context){}
  void map(HadoopPipes::MapContext& context) {
    std::vector<std::string> words =
      HadoopUtils::splitString(context.getInputValue(), " ");
    for(unsigned int i=0; i < words.size(); ++i) {
      context.emit(words[i], "1");
    }}};
```

Pig

- Scripting language that generates Map/Reduce jobs
- User uses higher level operations
  - Group by
  - Foreach
- Word Count:

```
input = LOAD 'in-dir' USING TextLoader();
words = FOREACH input GENERATE
   FLATTEN(TOKENIZE(*));
grouped = GROUP words BY $0;
counts = FOREACH grouped GENERATE group,
   COUNT(words);
STORE counts INTO 'out-dir';
```

Hadoop Streaming

- Not everyone is a Java programmer
- Python, Perl, Shell scripts
- Most languages support
  - Reading from <stdin>
  - Writing to <stdout>
- Mapper & Reducer: External Programs
- Framework serializes/deserializes I/O to/ from Strings