Map-Reduce framework

Background

- Google, Yahoo, etc. deal with
  - very large amounts of data (many terabytes)
  - need to process data fairly quickly (within a day, e.g.)
  - use very large numbers of commodity machines (thousands)

A cluster at Yahoo.

- Google developed an infrastructure consisting of
  - the Google distributed file system GFS
  - the MapReduce computational model
- Other implementations include Hadoop from Apache.

Requirements and Constraints

- Want to run on 1,000–10,000 nodes.
- With that many nodes
  - some will fail
  - some will go down for maintenance

  Fault tolerance is essential.
- Want to work on petabytes of data
- Data will need to be distributed across many disks.
- Data access speeds will depend on location:
  - local disk will be fastest
  - same rack may be faster than different rack
- Replication is needed for performance and fault tolerance.

Requirements and Constraints

- Want an infrastructure that takes care of management tasks
  - distribution of data
  - management of fault tolerance
  - collecting results
- For a specific problem
  - user writes a few routines
  - routines plug into the general interface
- Goal: identify a class of computations that is
  - general enough to cover many problems
  - structured enough to allow development of an infrastructure
  - reasonably easy to tailor to specific problems
- MapReduce seems to fit this goal reasonably well.

Google MapReduce

- Related to two concepts from functional programming:
  - *Mapping*: applying a function to each element of a structure and returning a comparable structure of results.
  - *Reducing* or *folding*: Applying a binary operation, usually associative, often commutative, to an initial element and every successive element of a structure to produce a single reduced result, e.g. a sum.
- some functional programming primitives, including `Map` and `Reduce`.

- The names come from the Lisp world.
- A useful running example: Counting word frequencies in a collection of documents.

Google MapReduce

- MapReduce operations work with key/value pairs, e.g.
  - document name/document content
  - word/count
- A general MapReduce computation has several components:
  - Input reader: reads input files and divides into chunks for the map function
  - `map` function: receives key/value pair and emits 0 or more key value pairs.
  - Partition function: allocates output of maps to particular reduce functions.
  - Comparison function: used in sorting map output by keys.
  - `reduce` function: takes a key and a collection of values and produces a key/value pair.
  - Output writer: writes results to storage.
- All components can be customized.

Google MapReduce

- Google's MapReduce is implemented as a C++ library.
- Operates on commodity hardware and standard networking.
- Input data, intermediate results, and final results are stored in GFS.
- A master scheduler process distributes map, reduce tasks to workers.
- Fault tolerance:
  - The master pings workers periodically.
  - Workers that do not respond are marked as failed.
  - Jobs assigned to failed workers are rerun.
  - Master failure aborts the computation.

Introduction

- Map-Reduce is a programming model designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks.
- Map-Reduce programs are written in a particular style influenced by *functional programming* constructs, specifically idioms for processing lists of data.
- This module explains the nature of this programming model and how it can be used to write programs which run in the Hadoop environment.

Map-reduce Basics

- 1. **List Processing**
- Conceptually, Map-Reduce programs transform lists of input data elements into lists of output data elements.

- A Map-Reduce program will do this twice, using two different list processing idioms: *map*, and *reduce*.

- These terms are taken from several list processing languages such as LISP, Scheme, or ML

Map-reduce Basics

- **2.Mapping Lists**
- The first phase of a Map-Reduce program is called *mapping*.
- A list of data elements are provided, one at a time, to a function called the *Mapper*, which transforms each element individually to an output data element.

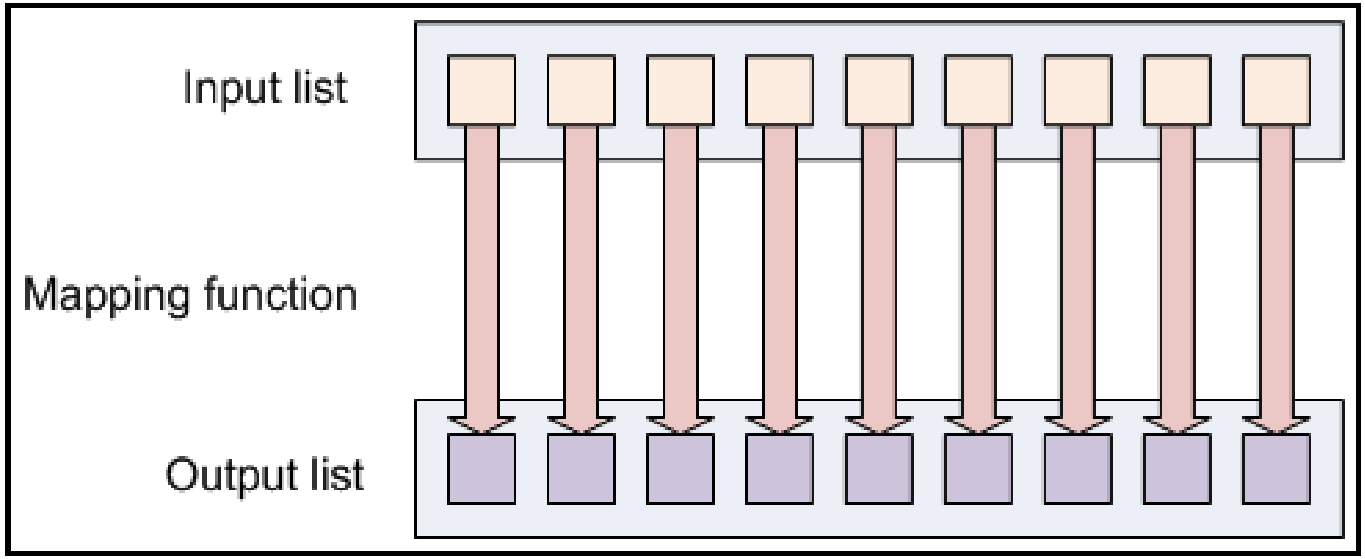


Figure 4.1: Mapping creates a new output list by applying a function to individual elements of an input list.

Map-reduce Basics

- **3.Reducing List**
- Reducing lets you aggregate values together.
- A *reducer* function receives an iterator of input values from an input list.

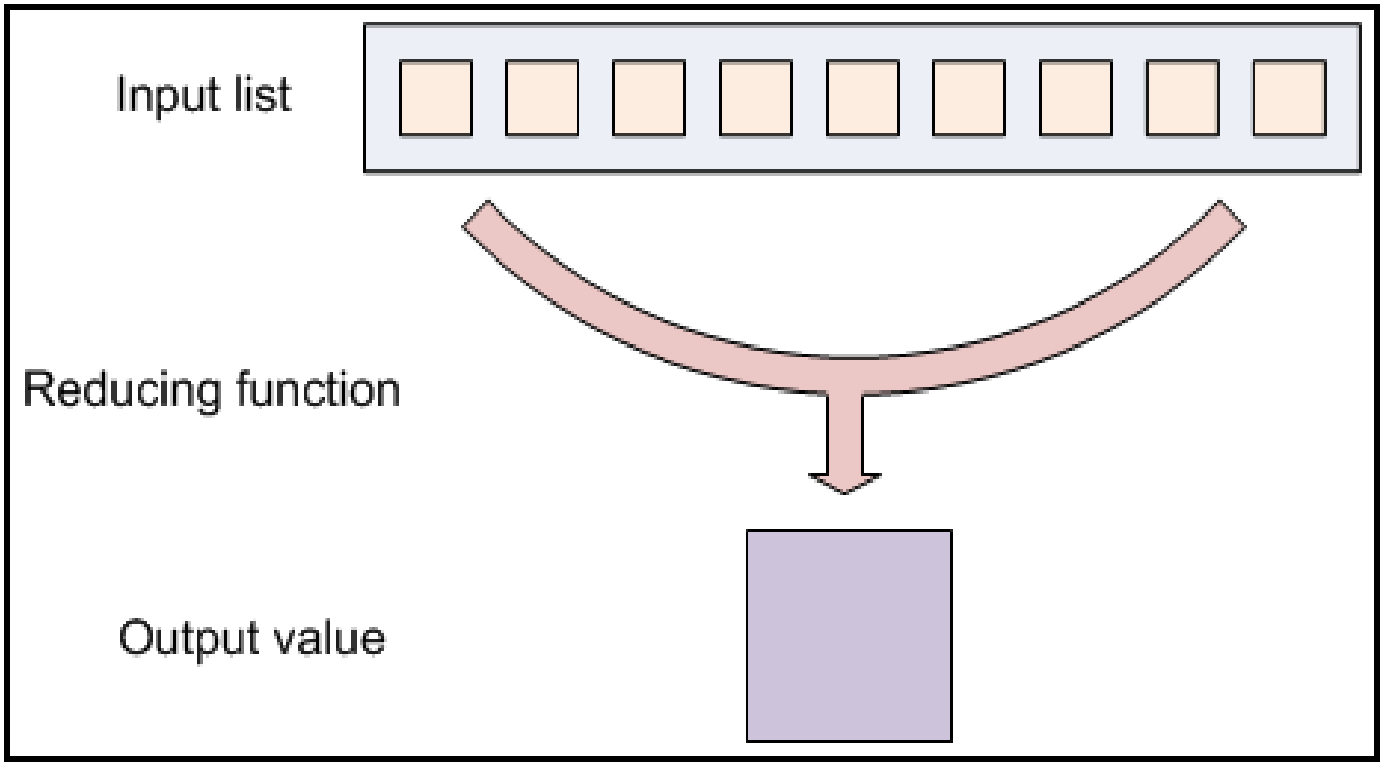


Figure 4.2: Reducing a list iterates over the input values to produce an aggregate value as output.

Map-reduce Basics

- **4.Putting Them Together in Map-Reduce:**
- The Hadoop Map-Reduce framework takes these concepts and uses them to process large volumes of information.
- A Map-Reduce program has two components:
  - Mapper
  - And reducer.

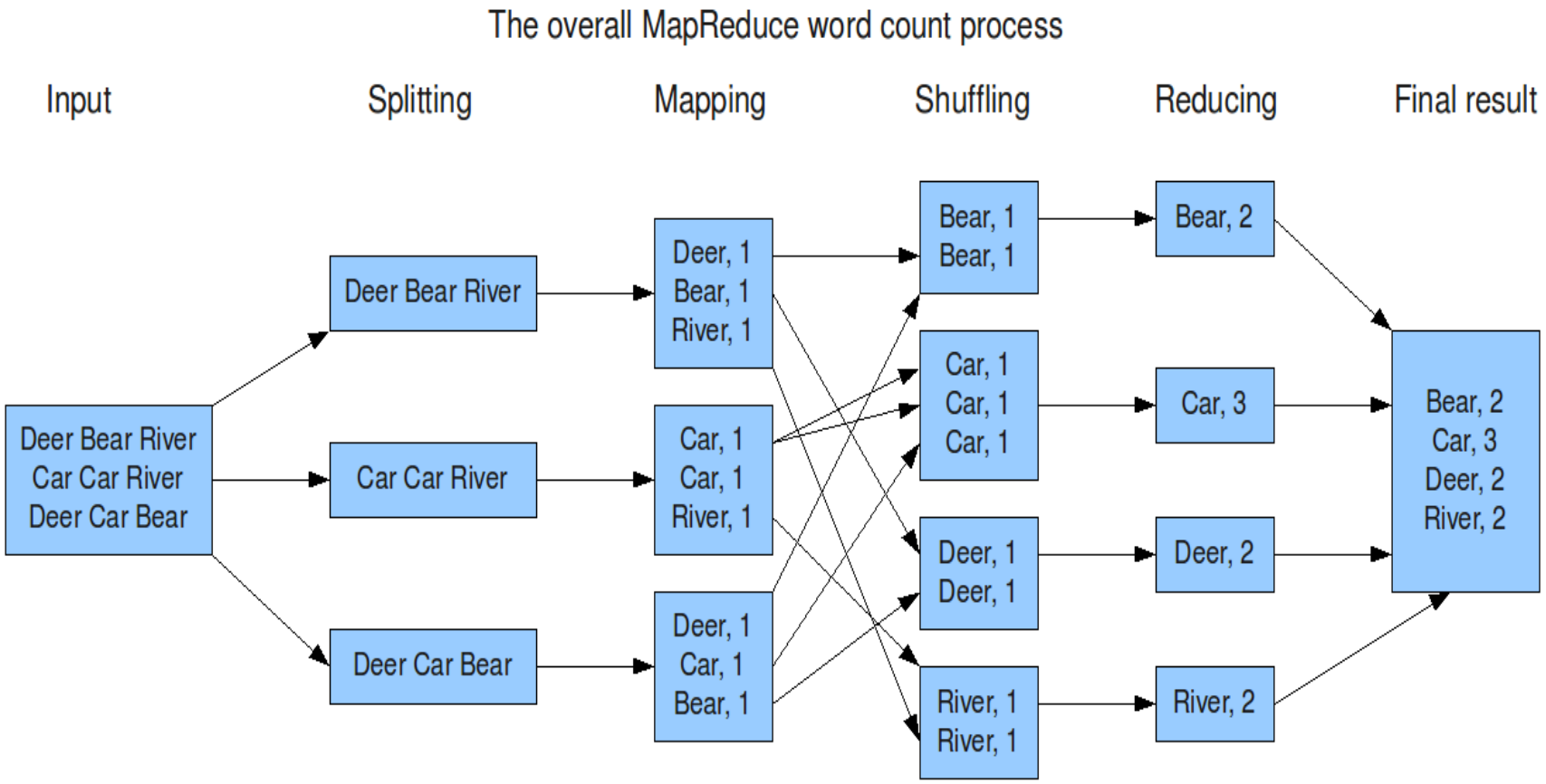
  The Mapper and Reducer idioms described above are extended slightly to work in this environment, but the basic principles are the same.

Example (word count)

- mapper (filename, file-contents):
  - **for each** word **in** file-contents:
    - **emit** (word, 1)

- reducer (word, values):
  - sum = 0
  - **for each** value **in** values:
  - sum = sum + value
  - **emit** (word, sum)

Example (word count)



Map-Reduce Data Flow

Now that we have seen the components that make up a basic MapReduce job, we can see how everything works together at a higher level:

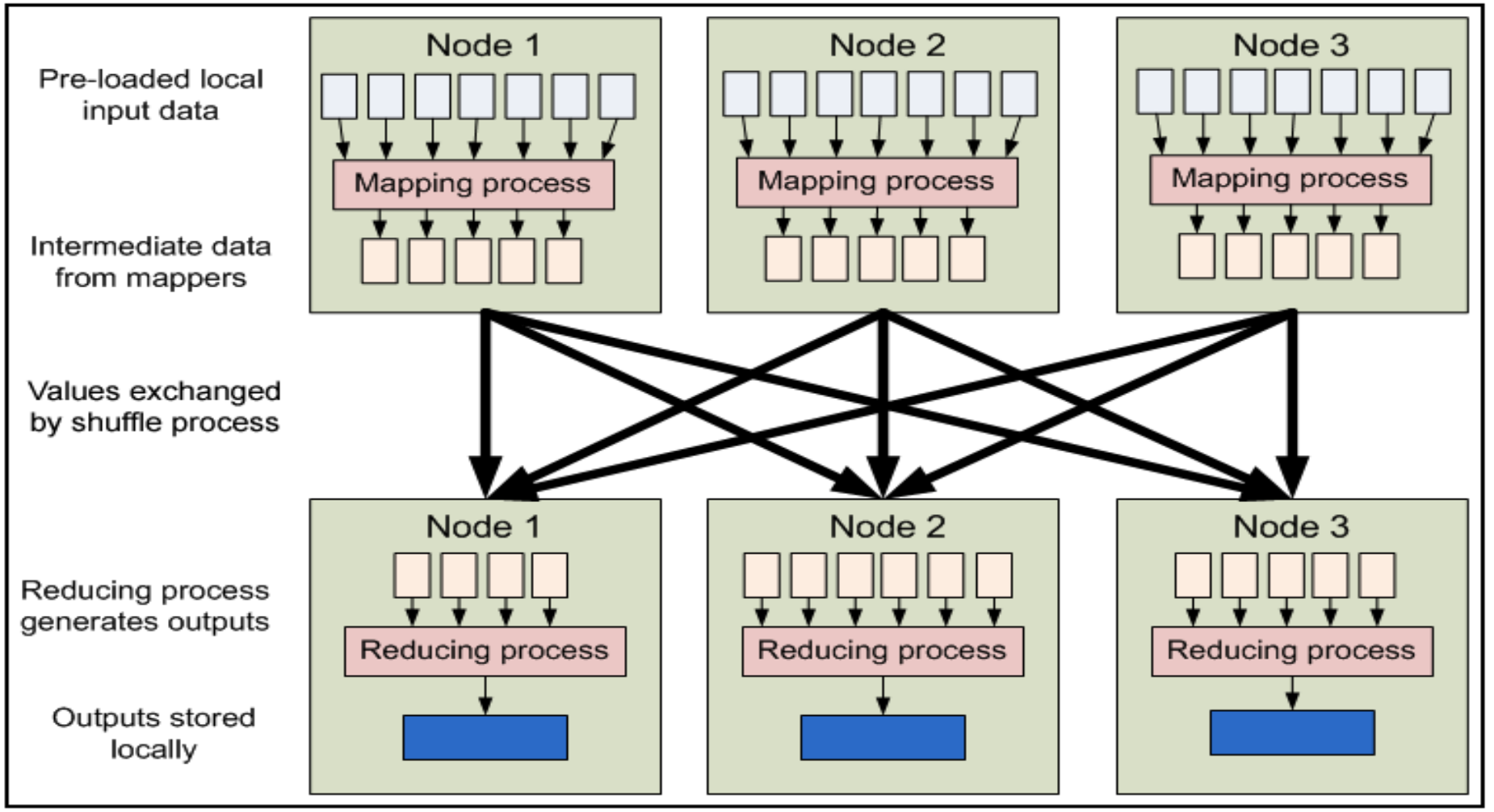


Figure 4.4: High-level MapReduce pipeline

Data flow

- Map-Reduce inputs typically come from input files loaded onto our processing cluster in HDFS.
- These files are distributed across all our nodes.
- Running a Map-Reduce program involves running mapping tasks on many or all of the nodes in our cluster.
- Each of these mapping tasks is equivalent:
  - no mappers have particular "identities" associated with them.
  - Therefore, any mapper can process any input file. Each mapper loads the set of files local to that machine and processes them.

Data flow

- When the mapping phase has completed, the intermediate (key, value) pairs must be exchanged between machines to send all values with the same key to a single reducer.
- The reduce tasks are spread across the same nodes in the cluster as the mappers.
- **This is the only communication step in MapReduce.**
- Individual map tasks do not exchange information with one another, nor are they aware of one another's existence.
- Similarly, different reduce tasks do not communicate with one another.

Data flow

- *The user never explicitly marshals information from one machine to another; all data transfer is handled by the Hadoop Map-Reduce platform itself, guided implicitly by the different keys associated with values.*
- *This is a fundamental element of Hadoop Map-Reduce's reliability.*
- *If nodes in the cluster fail, tasks must be able to be restarted.*
- *If they have been performing side-effects, e.g., communicating with the outside world, then the shared state must be restored in a restarted task.*
- *By eliminating communication and side-effects, restarts can be handled more gracefully.*

Input files

- This is where the data for a Map-Reduce task is initially stored.
- While this does not need to be the case, the input files typically reside in HDFS.
- The format of these files is arbitrary; while line-based log files can be used, we could also use a binary format, multi-line input records, or something else entirely.
- It is typical for these input files to be very large -- tens of gigabytes or more.

InputFormat

- These input files are split up and read is defined by the InputFormat.
- An InputFormat is a class that provides the following functionality:
  - Selects the files or other objects that should be used for input
  - Defines the *InputSplits* that break a file into tasks
  - Provides a factory for *RecordReader* objects that read the file
- Several InputFormats are provided with Hadoop.
- An abstract type is called *FileInputFormat*; all InputFormats that operate on files inherit functionality and properties from this class.
- When starting a Hadoop job, FileInputFormat is provided with a path containing files to read.
- The FileInputFormat will read all files in this directory. It then divides these files into one or more InputSplits each.
- You can choose which InputFormat to apply to your input files for a job by calling the setInputFormat() method of the *JobConf* object that defines the job.

- The default **InputFormat** is the ***TextInputFormat***.
  - This is useful for **unformatted** data or line-based records like log files.
- A more interesting input format is the ***KeyValueInputFormat***.
  - This format also treats each line of input as a separate record. While the **TextInputFormat** treats the entire line as the value,
  - the **KeyValueInputFormat** breaks the line itself into the key and value by searching for a tab character.
  - This is particularly useful for reading the output of one **MapReduce** job as the input to another
- Finally, the ***SequenceFileInputFormat*** reads special binary files that are specific to Hadoop.
  - These files include many features designed to allow data to be rapidly read into Hadoop mappers.
  - Sequence files are block-compressed and provide direct serialization and deserialization of several arbitrary data types (not just text).
  - Sequence files can be generated as the output of other **MapReduce** tasks and are an efficient intermediate representation for data that is passing from one MapReduce job to anther.

InputSplits

- *An **InputSplit** describes a unit of work that comprises a single map task in a **MapReduce** program.*
- *A **MapReduce** program applied to a data set, collectively referred to as a Job, is made up of several (possibly several hundred) tasks.*
- *Map tasks may involve reading a whole file; they often involve reading only part of a file.*
- *By default, the **FileInputFormat** and its descendants break a file up into 64 MB chunks (the same size as blocks in HDFS).*

RecordReader

- The **InputSplit** has defined a slice of work, but does not describe how to access it.
- The ***RecordReader*** class actually loads the data from its source and converts it into (key, value) pairs suitable for reading by the Mapper.
- The **RecordReader** instance is defined by the **InputFormat**.
- The default InputFormat, ***TextInputFormat***, provides a
  - ***LineRecordReader***, which treats each line of the input file as a new value.
  - The key associated with each line is its byte offset in the file.
  - The **RecordReader** is invoke repeatedly on the input until the entire **InputSplit** has been consumed. Each invocation of the **RecordReader** leads to another call to the **map**() method of the Mapper.

Mapper

- The **Mapper** performs the interesting user-defined work of the first phase of the **MapReduce** program.
- Given a key and a value, the map() method emits (key, value) pair(s) which are forwarded to the **Reducers**.
- A new instance of **Mapper** is instantiated in a separate Java process for each map task (**InputSplit**) that makes up part of the total job input. The individual mappers are intentionally not provided with a mechanism to communicate with one another in any way.
- This allows the reliability of each map task to be governed solely by the reliability of the local machine.
- The **map**() method receives two parameters in addition to the key and the value:

Mapper

- The ***OutputCollector*** object has a method named **collect**() which will forward a (key, value) pair to the reduce phase of the job.
- The ***Reporter*** object provides information about the current task; its **getInputSplit**() method will return an object describing the current **InputSplit**.
- It also allows the map task to provide additional information about its progress to the rest of the system.
- The **setStatus**() method allows you to emit a status message back to the user. The **incrCounter**() method allows you to increment shared performance counters.
- Each mapper can increment the counters, and the **JobTracker** will collect the increments made by the different processes and aggregate them for later retrieval when the job ends.

1.Partition & Shuffle (Mapper)

- After the first **map** tasks have completed, the nodes may still be performing several more map tasks each.
- But they also begin **exchanging** the intermediate outputs from the **map** tasks to where they are required by the **reducers**.
- This process of moving **map** outputs to the reducers is known as ***shuffling***.
- A different subset of the intermediate key space is assigned to each reduce node; these subsets (known as "**partitions**") are the inputs to the reduce tasks.
- Each map task may emit (key, value) pairs to any partition; all values for the same key are always reduced together regardless of which **mapper** is its origin.
- Therefore, the map nodes must all agree on where to send the different pieces of the intermediate data.
- The ***Partitioner*** class determines which partition a given (key, value) pair will go to.
- The default **partitioner** computes a hash value for the key and assigns the partition based on this result.

2. Sort (Mapper)

- Each reduce task is responsible for reducing the values associated with several intermediate keys.

- The set of intermediate keys on a single node is automatically sorted by Hadoop before they are presented to the Reducer.

Reduce

- A **Reducer** instance is created for each **reduce** task.
- This is an instance of user-provided code that performs the second important phase of job-specific work.
- For each **key** in the partition assigned to a **Reducer**, the Reducer's **reduce**() method is called once.
- This receives a key as well as an **iterator** over all the values associated with the key.
- The values associated with a key are returned by the **iterator** in an undefined order.
- The **Reducer** also receives as parameters ***OutputCollector*** and ***Reporter*** objects; they are used in the same manner as in the **map**() method.

1. OutputFormat (Reducer)

- The (key, value) pairs provided to this **OutputCollector** are then written to output files.
- The way they are written is governed by the ***OutputFormat***.
- The **OutputFormat** functions much like the **InputFormat** class.
- The instances of **OutputFormat** provided by **Hadoop** write to files on the local disk or in HDFS; they all inherit from a common ***FileOutputFormat***.

2. RecordWriter (Reducer)

- *Much like how the **InputFormat** actually reads individual records through the **RecordReader** implementation, the **OutputFormat** class is a factory for **RecordWriter** objects; these are used to write the individual records to the files as directed by the **OutputFormat**.*

3. Combiner (Reducer)

- The pipeline showed earlier omits a processing step which can be used for optimizing bandwidth usage by **MapReduce** job.
- Called the ***Combiner,*** this pass runs after the **Mapper** and before the **Reducer**.
- Usage of the **Combiner** is optional. If this pass is suitable for your job, instances of the **Combiner** class are run on every node that has run **map** tasks.
- The **Combiner** will receive as input all data emitted by the **Mapper** instances on a given node. The output from the **Combiner** is then sent to the **Reducers**, instead of the output from the **Mappers**.
- The Combiner is **a "mini-reduce" process which operates only on data generated by one machine.**

More Tips about map-reduce

- **Chaining Jobs**
- Not every problem can be solved with a **MapReduce** program, but fewer still are those which can be solved with a single **MapReduce** job. Many problems can be solved with **MapReduce**, by writing several **MapReduce** steps which run in series to accomplish a goal:
- E.g
  - Map1 -> Reduce1 -> Map2 -> Reduce2 -> Map3…

Listing and Killing Jobs:

- It is possible to submit jobs to a Hadoop cluster which malfunction and send themselves into infinite loops or other problematic states.
- In this case, you will want to manually kill the job you have started.
- The following command, run in the Hadoop installation directory on a Hadoop cluster, will list all the current jobs:

- $ bin/hadoop job -list
- currently running JobId StartTime UserName job_200808111901_0001 1 1218506470390 aaron

- $ bin/hadoop job -kill *jobid*

Conclusions

- This module described the **MapReduce** execution platform at the heart of the Hadoop system. By using **MapReduce**, a high degree of parallelism can be achieved by applications.
- The **MapReduce** framework provides a high degree of fault tolerance for applications running on it by limiting the communication which can occur between nodes, and requiring applications to be written in a "**dataflow-centric**" manner.

Assignment

- Parallel efficiency of map-reduce.

- Q&A/Feedback?