Big Data
- Map Reduce

Dr. Matt Zhang
ITU

Outline

- Motivation
- How M/R Works
  - Programming Model
- Jobs
- M/R Types and Formats
- M/R Features
- Develop M/R Application
  - Word Count Example

Motivation

- Data-intensive computing has arrived:
  - Both user-facing services and batch data processing
  - Data analysis is key
- Need massive scalability and easy parallelism
  - PB's of data, millions of files, 1000's of nodes, millions of users.

Motivation (cont.)

- Need to do this cost effectively and reliably
  - Data warehouse too expensive
    - Teradata maintenance support fee > millions of $ per year
  - Use commodity hardware where failure is the norm
  - Share resources among multiple projects

MapReduce to the rescue!

MR Features

- Simple data-parallel programming model and framework:
  - Designed for scalability and fault-tolerance
  - Automatic parallelization and distribution
  - Status and monitoring tools
  - Abstracts all the internal work away from developers
    - Can focus simply on writing Map and Reduce functions

MR Applications

- Pioneered by Google
  - Processes 20 PB of data per day
  - Popularized by open-source Hadoop project
  - Used at Yahoo!, Facebook, Amazon, etc.

MR Applications

- **Google:** index construction for search, article clustering for Google news, statistical machine translation
- **Yahoo!:** web-map and spam detection for Yahoo! mail
- **Facebook:** Ad optimization and spam detection

MR and Data Analytics

- Data cleaning:
  - Preprocess data in order to reduce noise and handle missing values – goal is to improve learning
- Relevance analysis:
  - Remove the irrelevant or redundant attributes using correlation analysis

MR and Data Analytics

- Data transformation and reduction:
  - Generalize to higher-level concepts, and/or
  - Normalize data (an attribute value is scaled to be between 0.0 – 1.0), especially if neural networks or distance measurements are used in the learning step.
  - Data can be Reduced by applying methods ranging from wavelet transformation to discretization techniques

Programming Model

- APIs: http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/

- Input data type: file of K/V records
- Map function: $(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$
- Reduce function: $(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$
- Example:

```
def mapper(line):
        foreach  word in line.split();
                output(word, 1);          // (K_inter, V_inter)

def  reducer(key, values):
      output(key, sum(values));           // list(K_out, V_out)
```

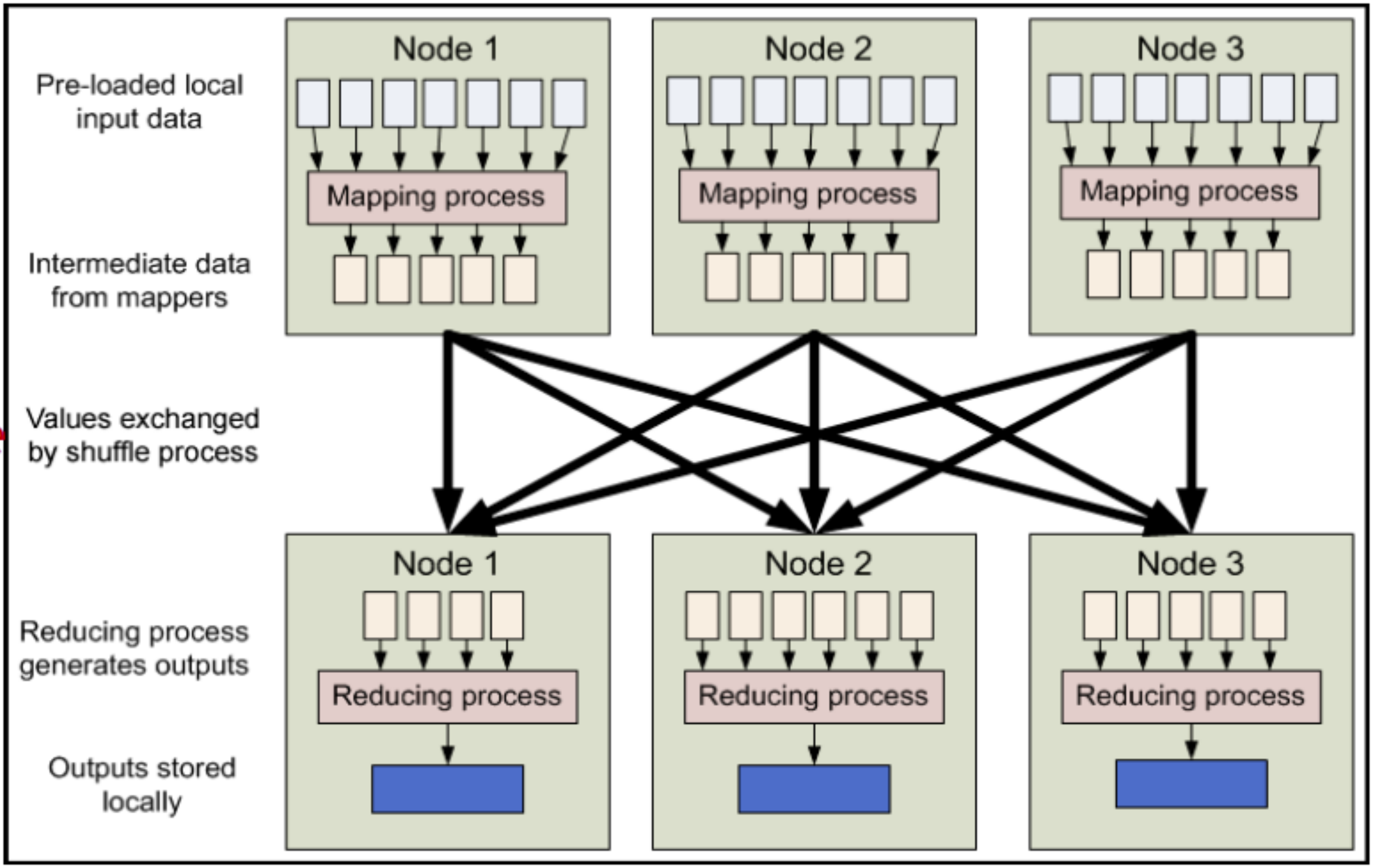
Programming Model

- Push: input split into large chunks and placed on local disks of cluster nodes
- Map: chunks (map tasks) are served to “mapper”
  - Prefer mapper that has data locally
  - Mappers save outputs to local disk before serving them to reducers; allows recovery
- Reduce: reducers execute reduce tasks only when map phase is completed

Programming Model

The Big Picture

Disk I/O Intensive



High degree of parallelism: Master/Slave architecture

Partitioning

- Partitioning/Shuffling: divide intermediate key space across reducers:
  - K reduce tasks ➔ K partitions (simple hash function)
  - E.g., K =3, keys {1,2}, {3,4}, {5,6}

JobTracker

- Software daemons control MR jobs
- Resides on master node
  - Client submit MR jobs to JobTracker
  - It assigns MR tasks to other nodes
- Each slave node has a TaskTracker daemon
  - Instantiating the M/R tasks
  - Status reporting to JobTracker

Job Terminology

- Job
  - user program
- Task
  - Execution of a single M/R over a slice of data
  - If one fails, JobTracker will start another one
  - Speculative execution

Mapper

- Reads data of key/value pairs
- Run in parallel, each processing a portion of input
- Output also key/value pairs
- Mappers run on nodes with data locality
  - Minimize network traffic

Reducer

- Process the output intermediate key/value pairs from Mapper, and output results
- Intermediate values for each key are combined into a list
  - Same key goes to same reducer
  - Sorted key order – “shuffle and sort”
- Output zero or more final key/value pairs
  - Write to HDFS

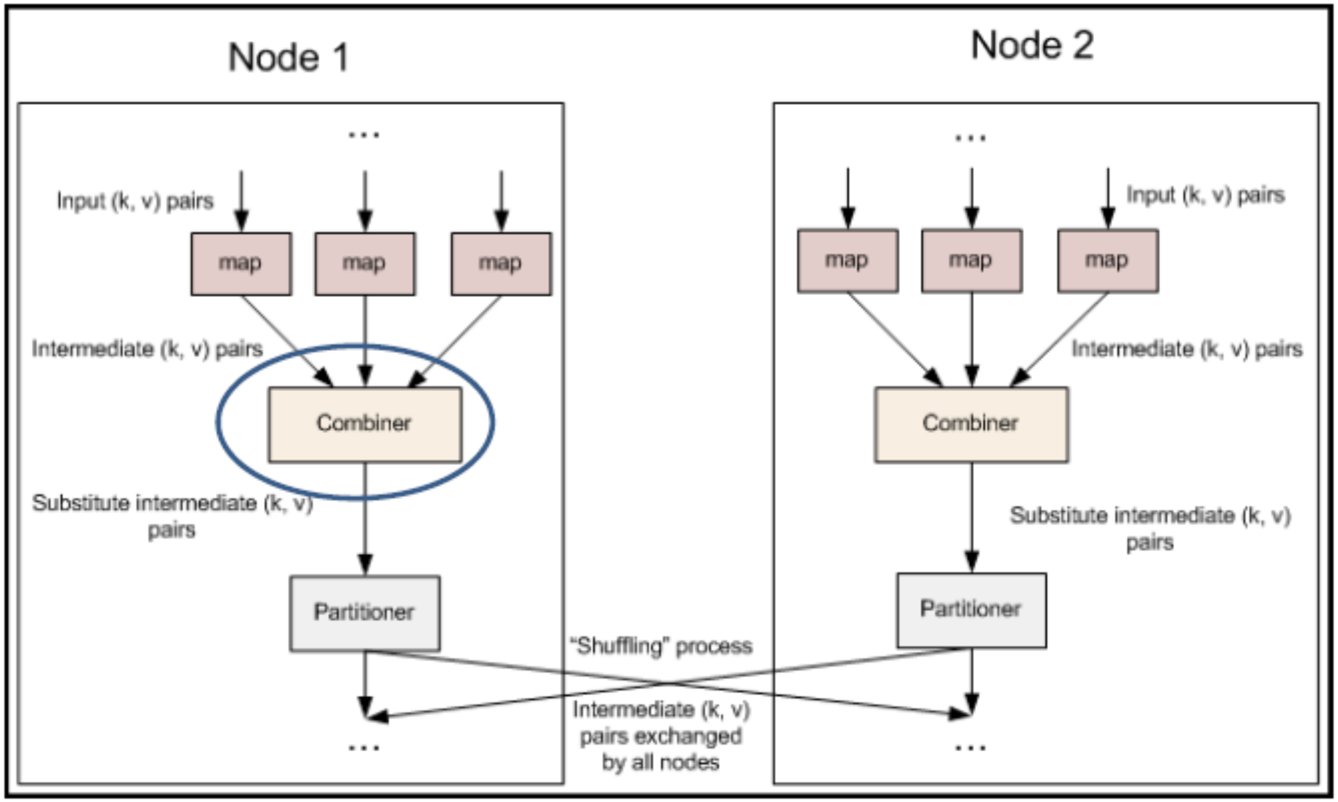
Shuffling

- Shuffle and sort:
  - All mappers typically have all intermediate keys
    - All to all communication
- Bottleneck?
  - Reducers cannot start until all Mappers finish
  - In practice, Mappers transfer data after finishing
    - avoids data transfer at the same time

Combiner

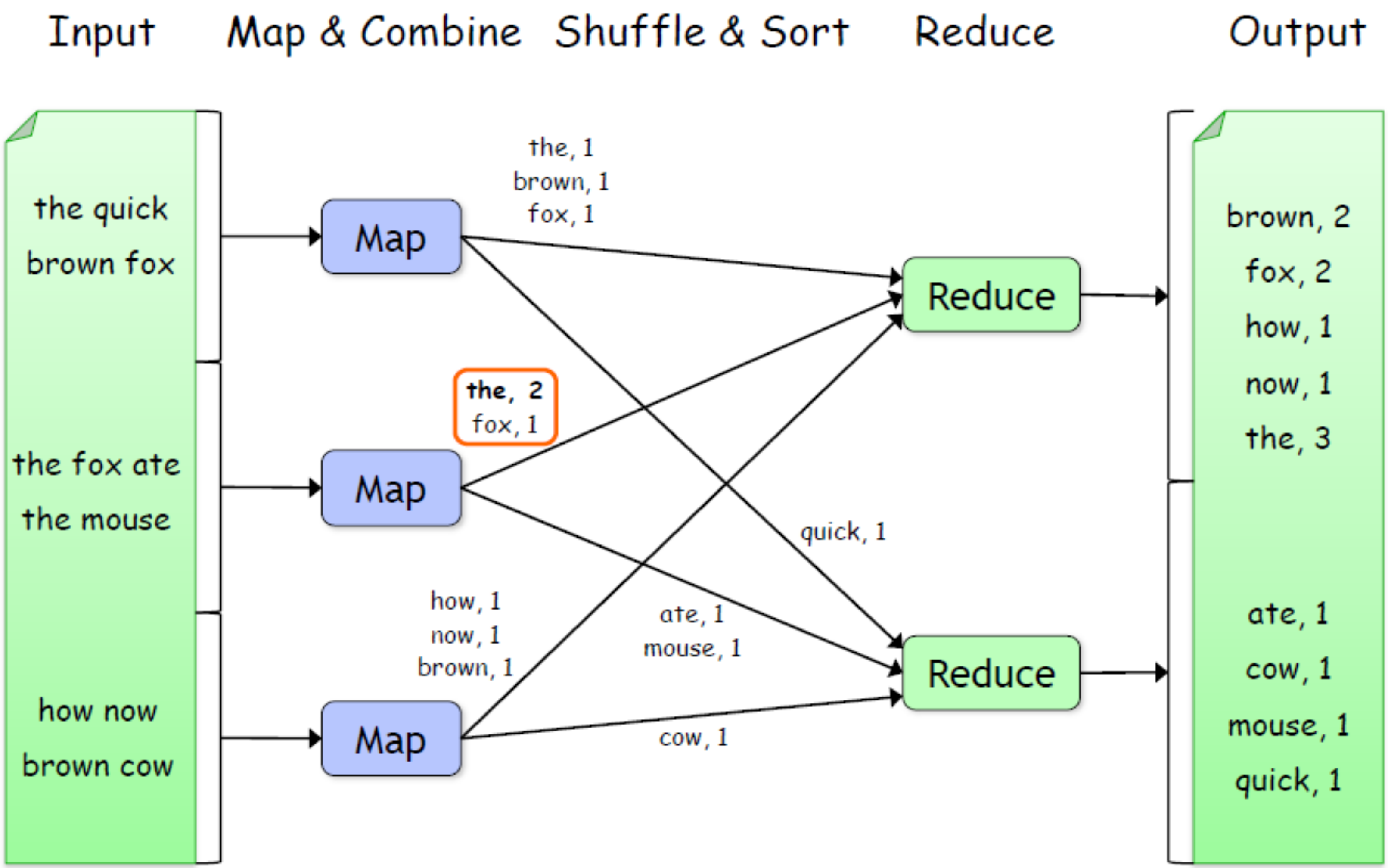
- Mappers produce large amount of intermediate data
  - Lots of network traffic
- Pre-aggregation
  - Mini-reducer
  - Runs on single Mapper's output
  - Input/output data types for Combiner/Reducer must be the same
  - Code is often the same as reducer

Combiner



- A combiner is a local aggregation function for repeated keys produced by the same mapper

Combiner Example



Word Count with Combiner

Fault Tolerance in MapReduce

- If a task crashes:
  - Retry on another node
    - OK for a map because it has no dependencies
    - OK for reduce because map outputs are on disk
- If a node crashes:
  - Re-launch its current tasks on other nodes
  - Re-run any maps the node previously ran to get output data

Fault Tolerance in MapReduce (cont.)

- If a task is going slowly:
- Launch second copy of task on another node (“speculative execution”)

Example: Inverted Index

- **Input**: (filename, text) records
- **Output**: list of files containing each word

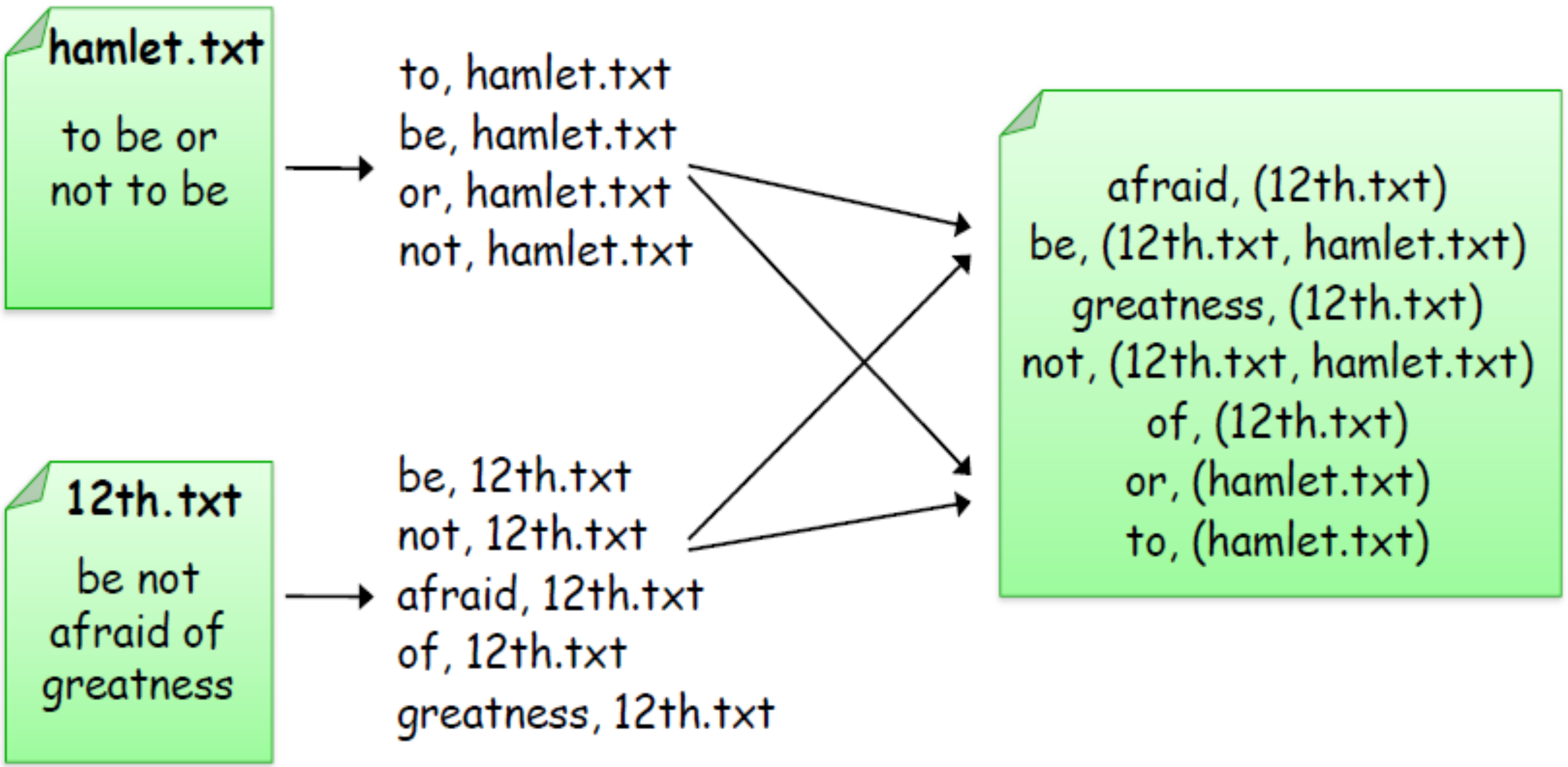
- **Map**:

```
foreach  word  in text.split():
        output(word, filename)
```

- **Reduce**:

```
def  reduce(word, filenames):
    output(word, sort(filenames))
```

Example: Inverted Index

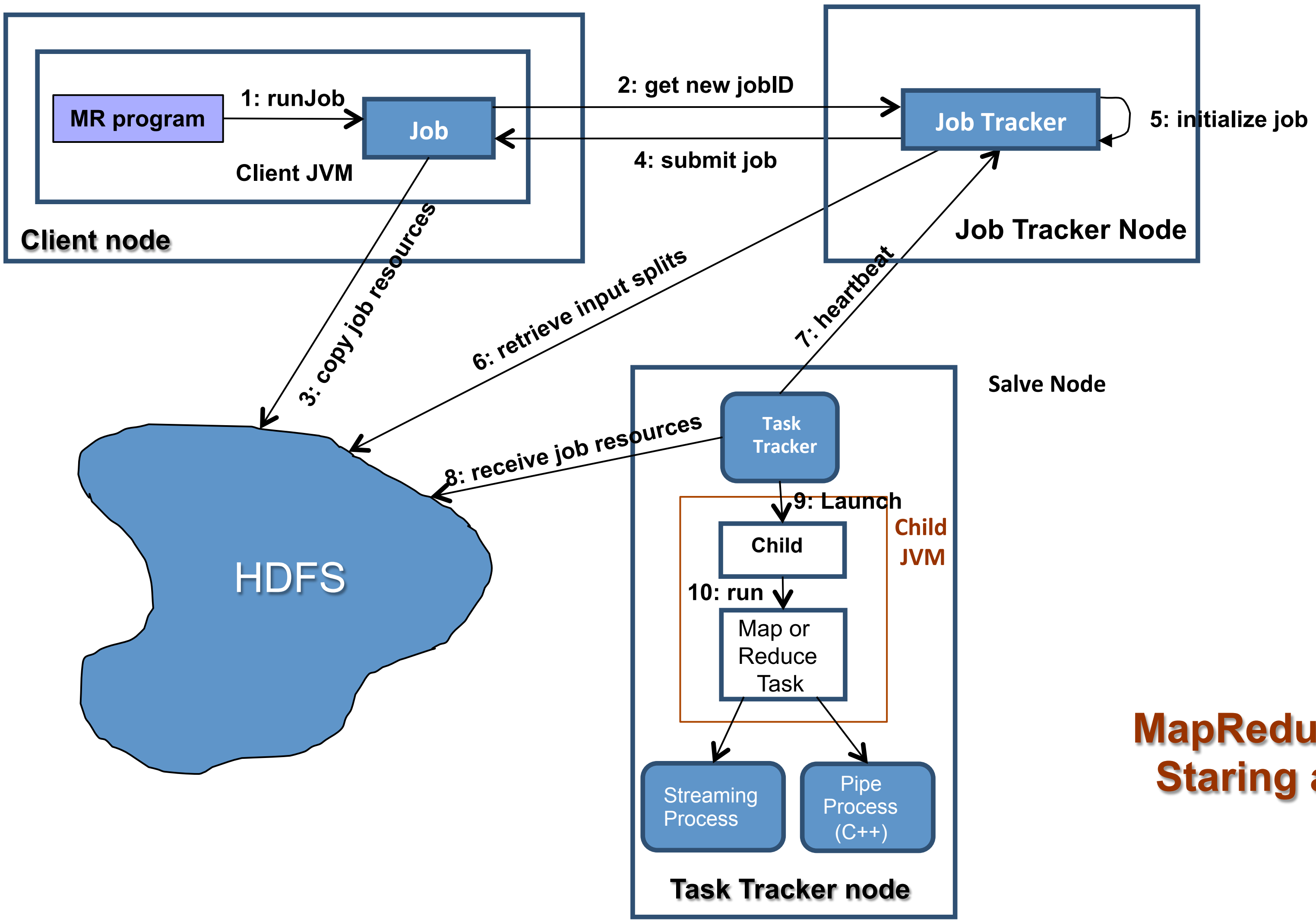


Inverted Index Example

Applications

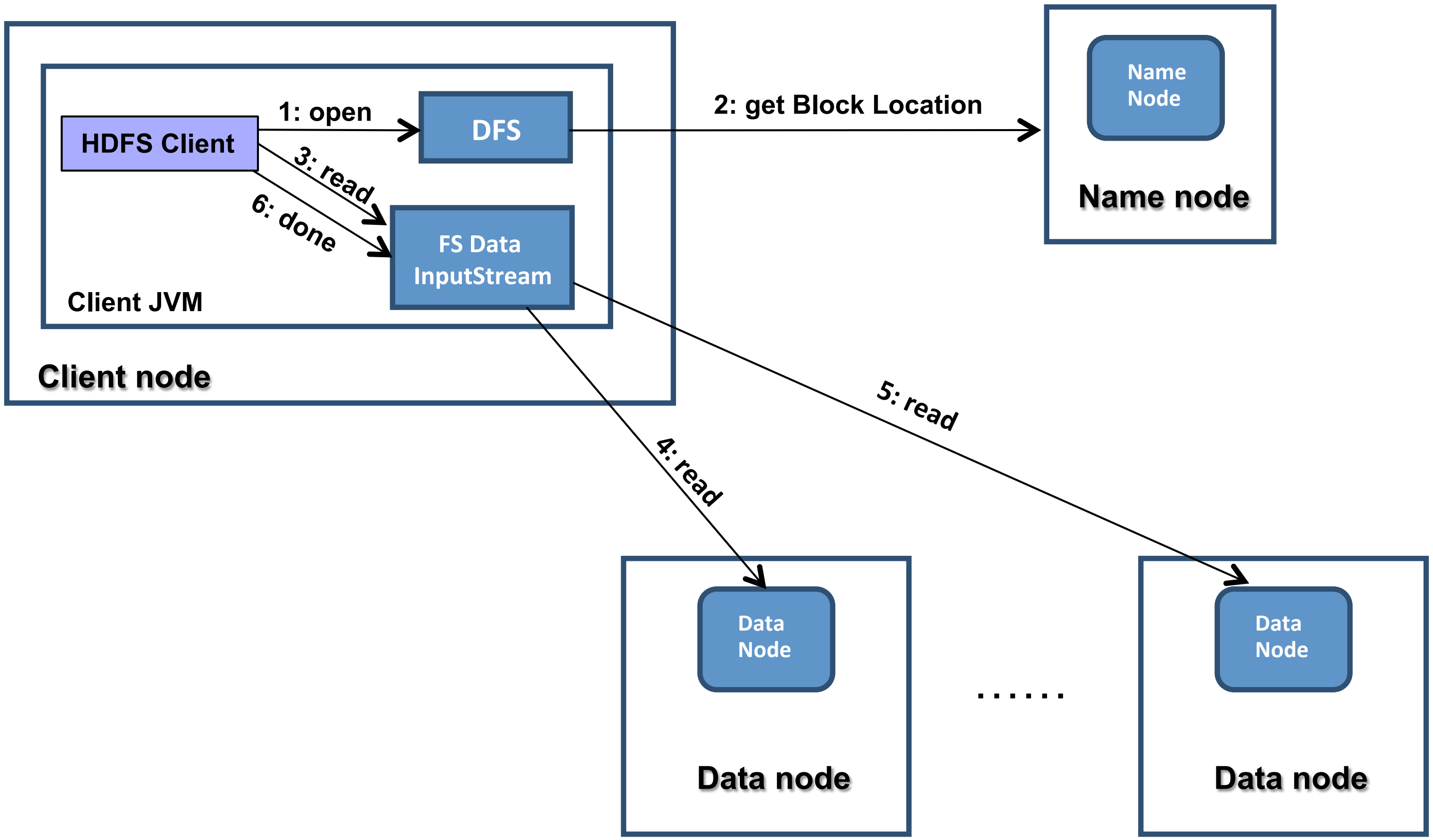
- What applications may perform well?
  - Modest computing relative to data
  - Data-independent processing of maps
  - Data-independent processing of keys
  - Smaller ballooning of map output relative to input

Starting a Job

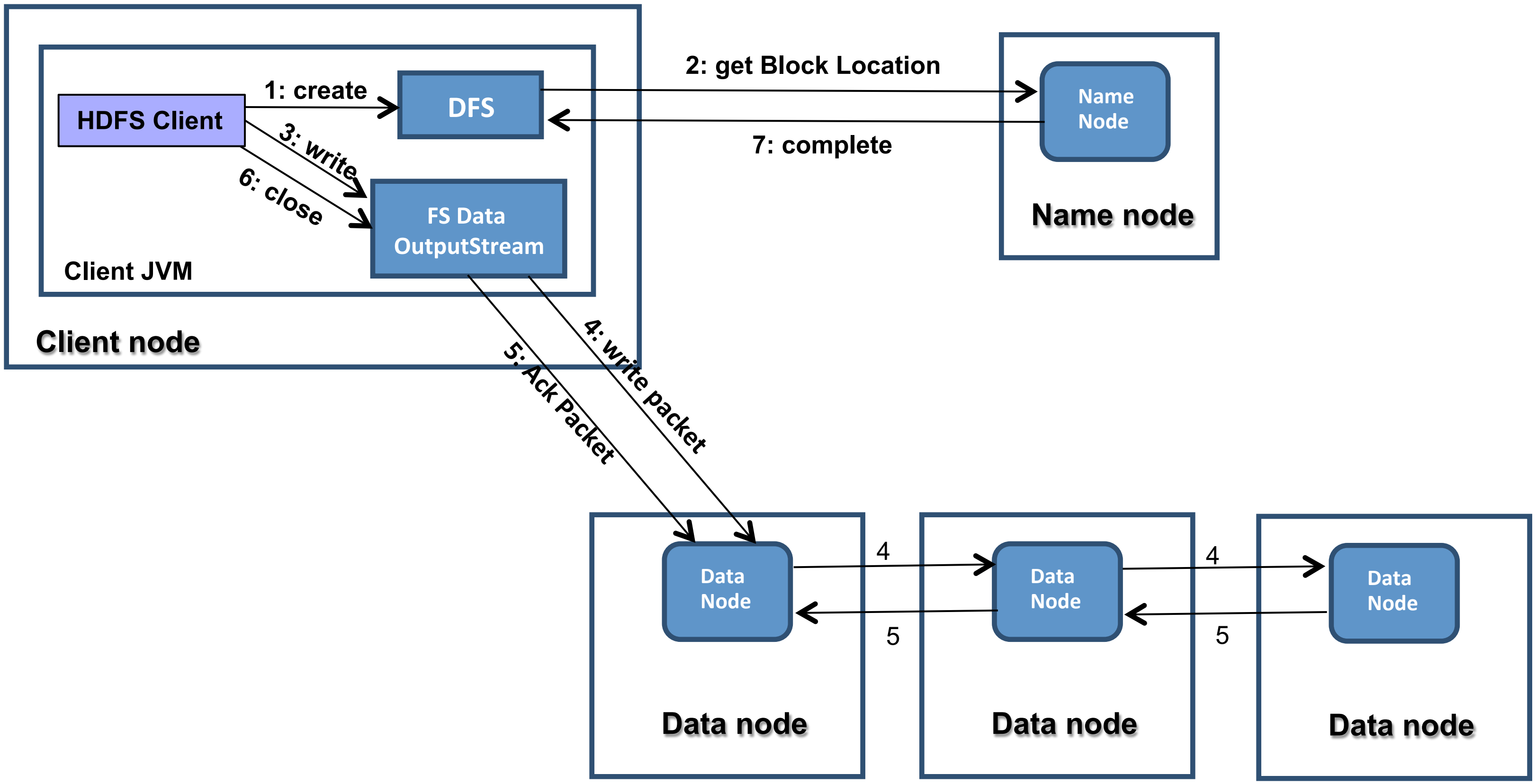


MapReduce-v1 Staring a Job

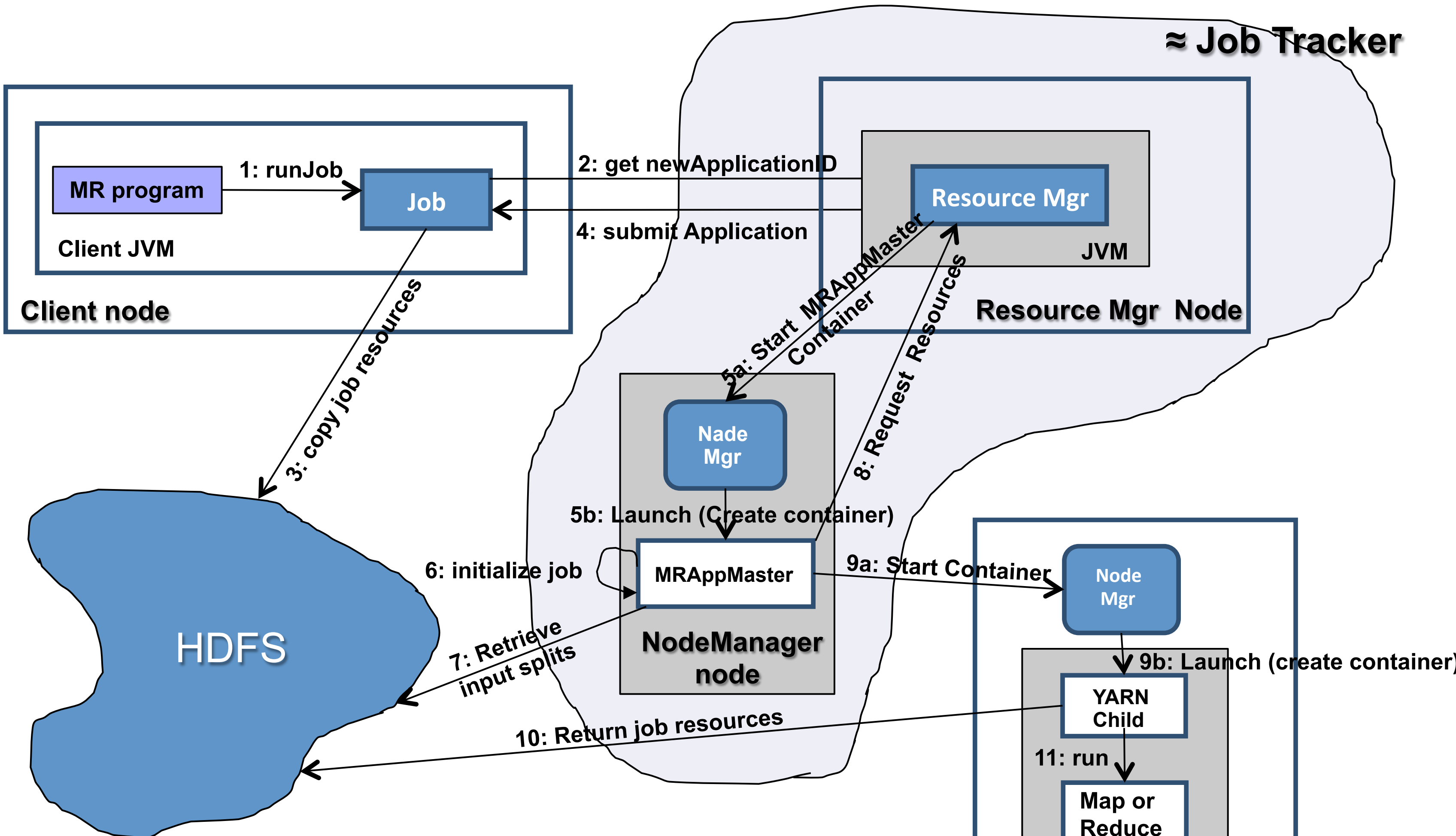
Read Anatomy



Write Anatomy



Starting a Job (Yarn)



MapReduce-v2
Staring a Job

Mapred-site.xml:
Mapreduce.framework.name = yarn

Developing M/R Applications

Developing MapReduce Application

- Write Map and Reduce functions and test them independently. MRUnit (http://incubator.apache.org/mrunit) is a library used to test the mapper() or reducer() as stand-alone function.
- MRUnit is used with JUnit to test MR Jobs as part of your IDE environment.
- Write a driver program to run a job.

Developing MapReduce Application

- Run the job from your IDE using a small subset of the data
- Debug using the IDE debugger.
- Run against the full dataset and in a cluster environment
- May expose issues that did not show up in the IDE testing.

Developing MapReduce Application

- After the program is working in a cluster, it is time for tuning through profiling.
  - Before developing MapReduce job, we need to set up and configure the development environment.
  - For details refer to Chapter-10, Hadoop: The Definitive Guide, 4th Edition

Word Count Example: Mapper

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class WordCountMapper extends MapReduceBase
    implements Mapper <LongWritable, Text, Text,
    IntWritable>
{
    // hadoop supported data types
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
```

Word Count Example: Mapper

```
// map method that performs the tokenizer job and
// framing the initial key value pairs
public void map(LongWritable key, Text value,
OutputCollector<Text, IntWritable> output, Reporter
reporter)  throws  IOException
{
    // taking one line at a time and tokenizing the same
    String line = value.toString();
    StringTokenizer   tokenizer = new
                      StringTokenizer(line);
```

Word Count Example: Mapper

```
 // iterating through all the words available in that line
 // and forming the key value pair
while (tokenizer.hasMoreTokens())
{
  word.set(tokenizer.nextToken());
  // send to output collector which in turn passes the
  // same to reducer
  output.collect(word, one);
}
    }
}
```

Word Count Example: Reducer

```
import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
```

Word Count Example: Reducer

```
public class WordCountReducer  extends
    MapReduceBase  implements Reducer<Text,
    IntWritable, Text, IntWritable>
{
    // reduce method accepts the Key Value
    // pairs from mappers, do the aggregation
    // based on keys and produce the final output
```

Word Count Example: Reducer

```
public void reduce(Text  key,
Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output,
Reporter reporter)  throws  IOException
 {
      int sum = 0;
     /* iterates through all the values available
          with a key and add them together and
          give the final result as the key and sum
          of its values */
```

Word Count Example: Reducer

```
      while (values.hasNext())
      {
          sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
   }
}
```

Word Count Example: Driver

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
```

Word Count Example: Driver

```
public class WordCount extends
Configured implements Tool {
    public int run(String[] args) throws
Exception
    {
        //creating a JobConf object and
        // assigning a job name for
        // identification purposes
        // Class JobConf -
```

Word Count Example: Driver

/*http://hadoop.apache.org/docs/r2.3.0/api/org/apache/hadoop/mapred/JobConf.html */

```
JobConf  conf = new
   JobConf(getConf(),
   WordCount.class);

conf.setJobName("WordCount");
```

Word Count Example: Driver

```
// Setting configuration object with the
// Data Type of output Key and Value
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(
        IntWritable.class);
```

Word Count Example: Driver

```
// Providing the mapper and reducer
// class names
conf.setMapperClass(
        WordCountMapper.class);
conf.setReducerClass(
        WordCountReducer.class);
```

Word Count Example: Driver

```
// the hdfs input and output directory
// to be fetched from the command
// line.
// Class Path - https://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/fs/Path.html
FileInputFormat.addInputPath(
   conf, new Path(args[0]));
FileOutputFormat.setOutputPath(
   conf, new Path(args[1]));
```

Word Count Example: Driver

```
    JobClient.runJob(conf);
    /* Class JobClient –
       http://hadoop.apache.org/docs/r2.2.0/
       api/org/apache/hadoop/
       mapred/JobClient.html */
    return 0;
}
```

Word Count Example: Driver

```
public static void main(String[] args)
  throws Exception
    {
          int res = ToolRunner.run(
             new Configuration(),
             new WordCount(), args);
          System.exit(res);
    }
}
```

Summary

- We introduced the programming model of MapReduce
- We discussed the components of MapRudece
- We covered the steps recommended for developing MapReduce Applications
  - Word Count Example