CPSC 436C
Cloud Computing for Data Science

MapReduce
Simplified Data Processing on Large Clusters

Maryam R.Aliabadi
mraiyata@cs.ubc.ca

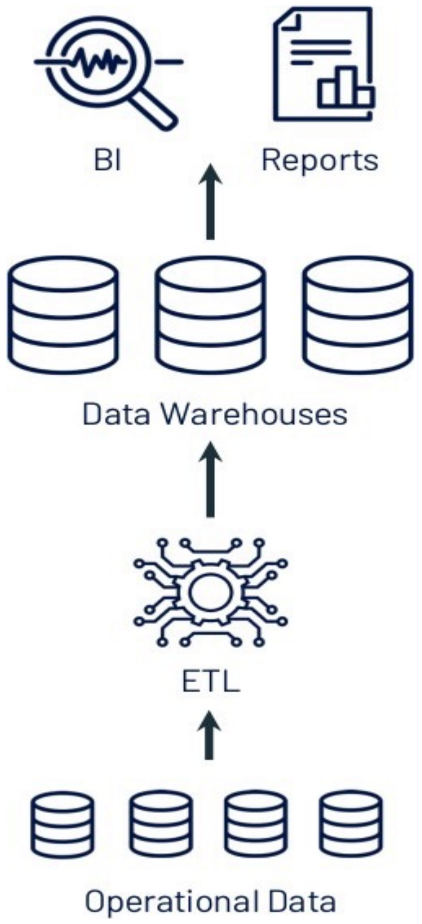
Fall 2023

Last Class‘ Review

- Data Management Systems
  - Data Warehouse
  - Data Lake
  - Lake House

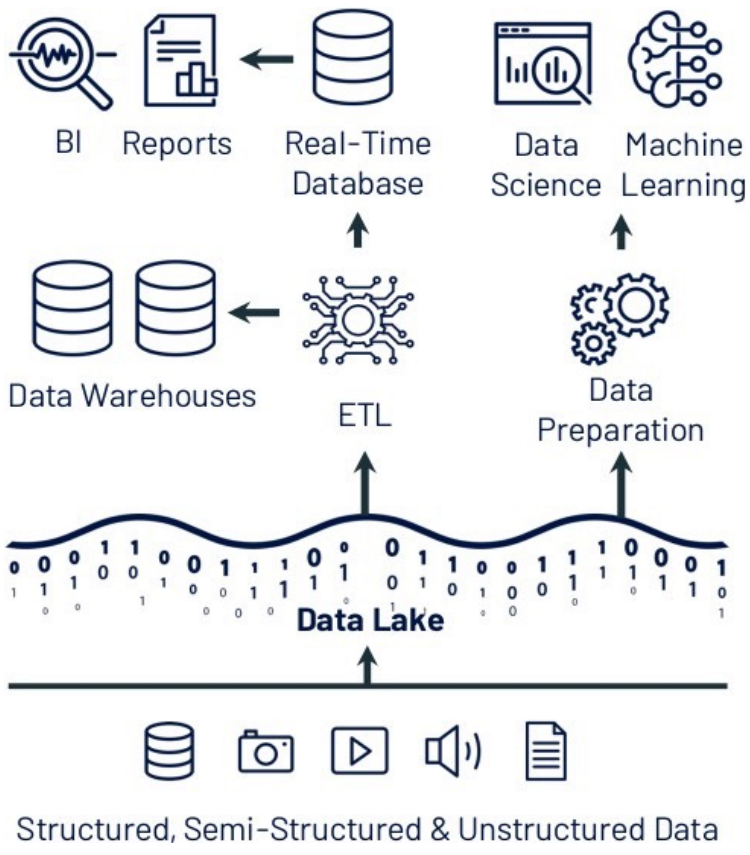
Data Warehouses (1980s)

- ETL (Extract, Transform, Load) data directly from operational database systems.
- Purpose-built for SQL analytics and BI: schemas, indexes, caching, etc.
- Powerful management features such as ACID transactions and time travel
- Data Warehouse defines the schema before data is stored (Schema on write).



Data Lakes (2010s)

- Low-cost storage to hold all raw data, e.g., Amazon S3, and HDFS.
- ETL jobs then load specific data into warehouses, possibly for further ELT.
- Directly readable in ML libraries (e.g., TensorFlow and PyTorch) due to open file format.



Raw Versus Conformed Data

- Raw data is information stored in its original format
  - For example, JSON stored as a document
  - Relational systems can store and query this kind of raw, semi-structured data

- Conformed data is information that fits a specific schema, requiring transformation of raw data.

Data Warehouse and Data Lake

- Data warehouses
  - Store only conformed data
  - Transforms all data to a set schema as it is written
  - Performs additional tasks on the data, such as validation and metadata extraction.

- Data Lakes
  - Contain data in its raw format.
  - Performs the transformation on an as-needed basis, when the data is read by users.

- The trade-offs of conforming data include time and cost.

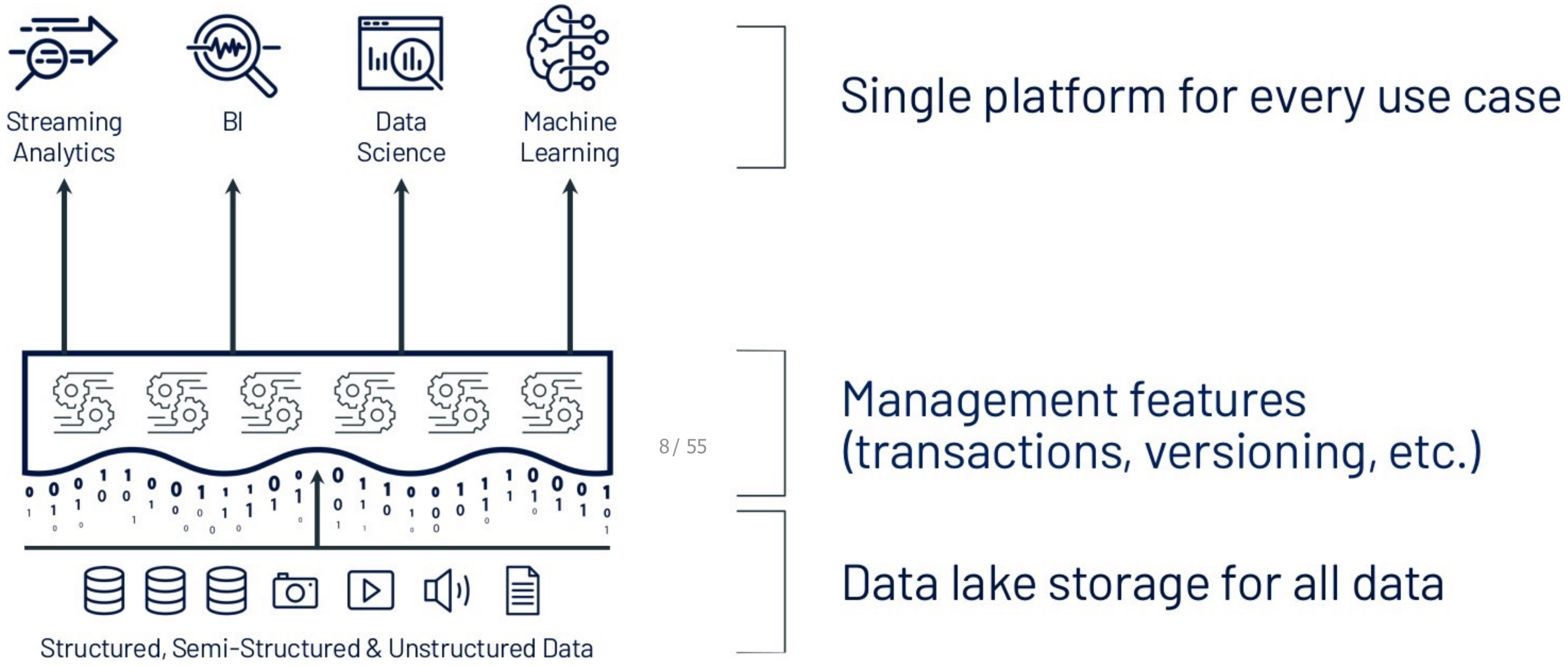
Schematization

- The trade-offs of ETL versus ELT systems is a difference in when the raw data is schematized.

- **Schema on read** is the paradigm of ELT systems, where raw data can be queried in its native format.

- **Schema on write** is the ETL paradigm, where the schema is applied when data is written into the data platform.

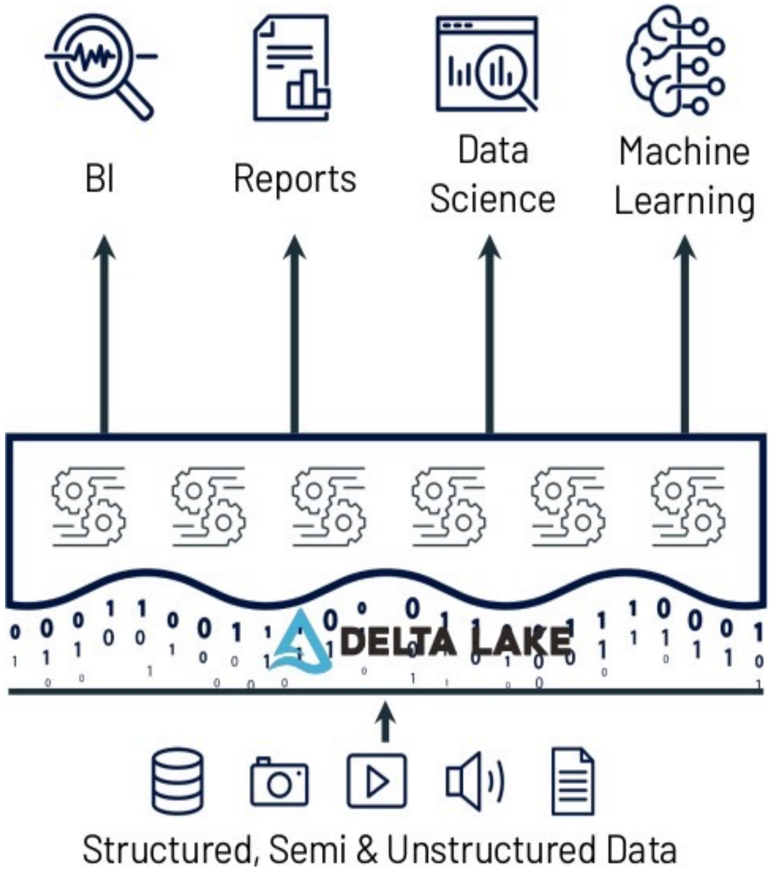
Lakehouse (2020)



- Lakehouse combines the benefits of Data Warehouses and Data Lakes while simplifying enterprise data architectures.

Lakehouse = Data Lake + Delta Lake

- Delta Lake is an open source storage layer that brings **reliability** to Data Lakes.
- Provides ACID transactions.
- Provides scalable metadata handling.
- Provides time travel and versioning.
- Unifies streaming and batch data processing.



How to Choose the Best trade-off

- The best trade-off is selected based on the requirements and the downstream processing needs of the application:
  - Performance
  - Cost
  - Complexity
  - Data quality
  - Type of ingested data
  - Frequency of ingested data
  - Type of analysis on target data

What Data Management System is the Best Fit?

- **Scenario 1: E-commerce Sales Analytics**

What Data Management System is the Best Fit?

- **Scenario 2: Real-time Social Media Analytics**

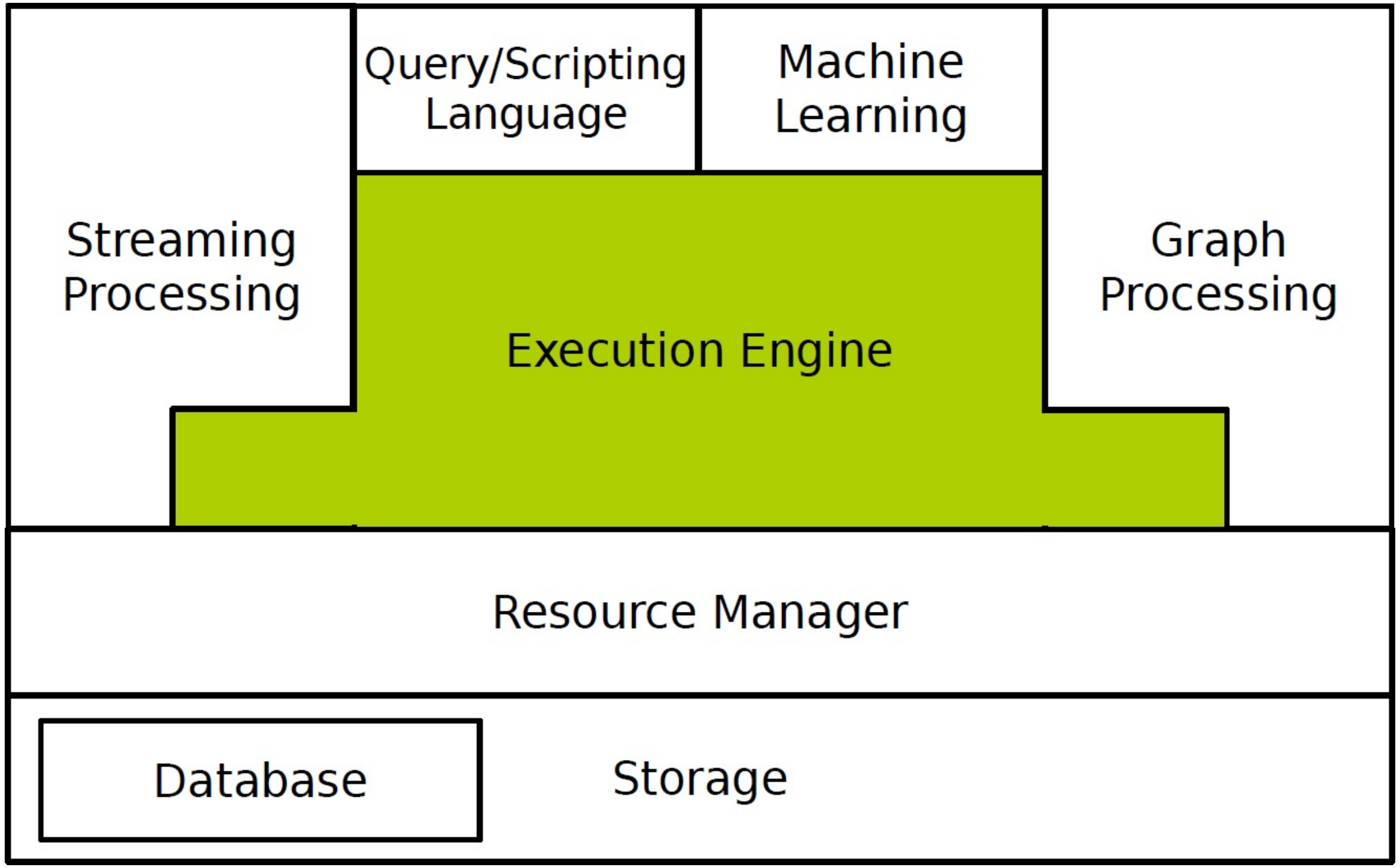
What Data Management System is the Best Fit?

- **Scenario 3: Healthcare and Medical Research Data**

Today's topic:

Data Processing - MapReduce

Data Processing



What do we do when there is too much data to process?

Scale Up vs. Scale Out

- Scale up or scale vertically: adding resources to a single node in a system.
- Scale out or scale horizontally: adding more nodes to a system.

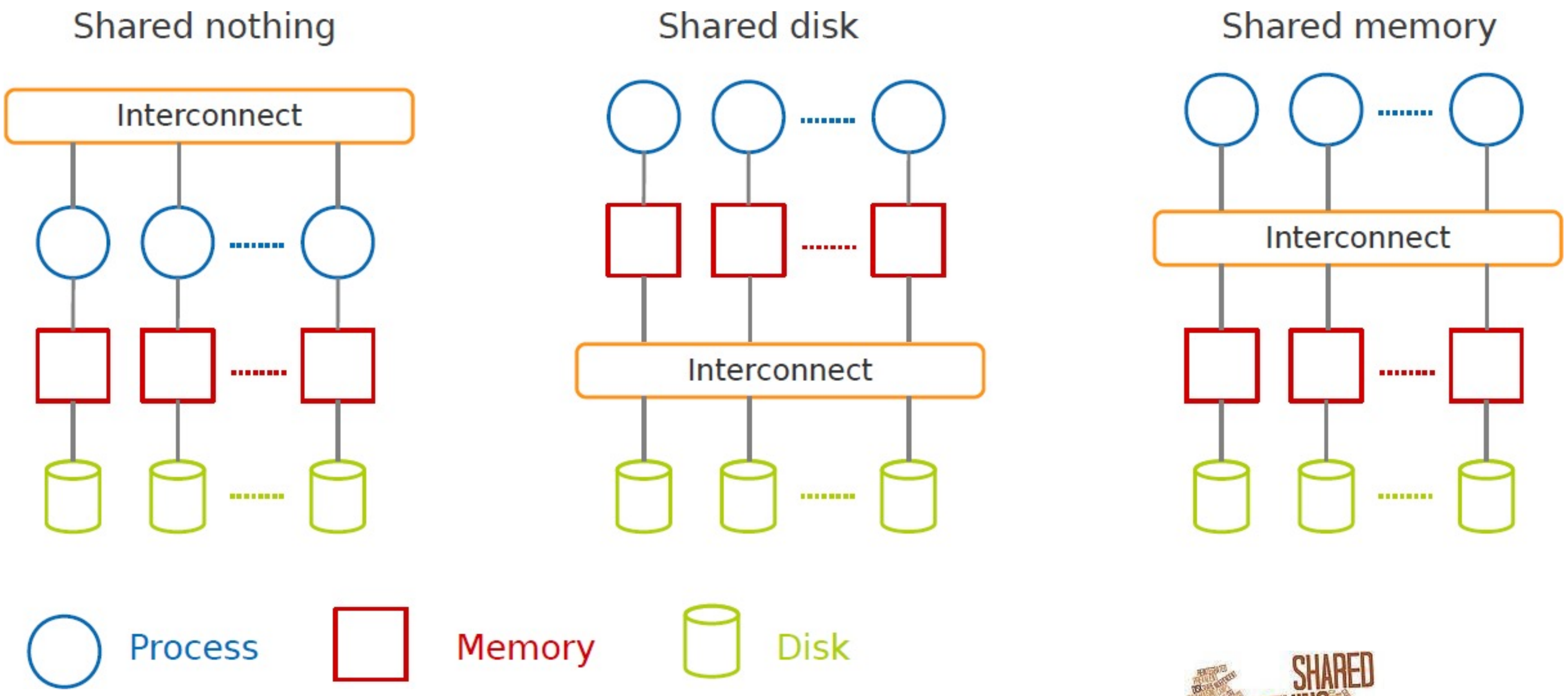
Scale Up vs. Scale Out

- Scale up: more expensive than scaling out.
- Scale out: more challenging for fault tolerance and software development.

Challenges

- How to distribute computation?
- How can we make it easy to write distributed programs?
- Machines failure.

Taxonomy of Parallel Architectures



DeWitt, D. and Gray, J. “Parallel database systems: the future of high performance database systems”. ACM Communications, 35(6), 85-98, 1992.

MapReduce

- A shared nothing architecture for processing large data sets with a parallel/distributed algorithm on clusters of commodity hardware.

MapReduce Resolves the Challenges

- Provides
  - data distribution
  - fault tolerance
  - load balancing

- Hides system-level details from programmers.

How MapReduce Resolves the Challenges?

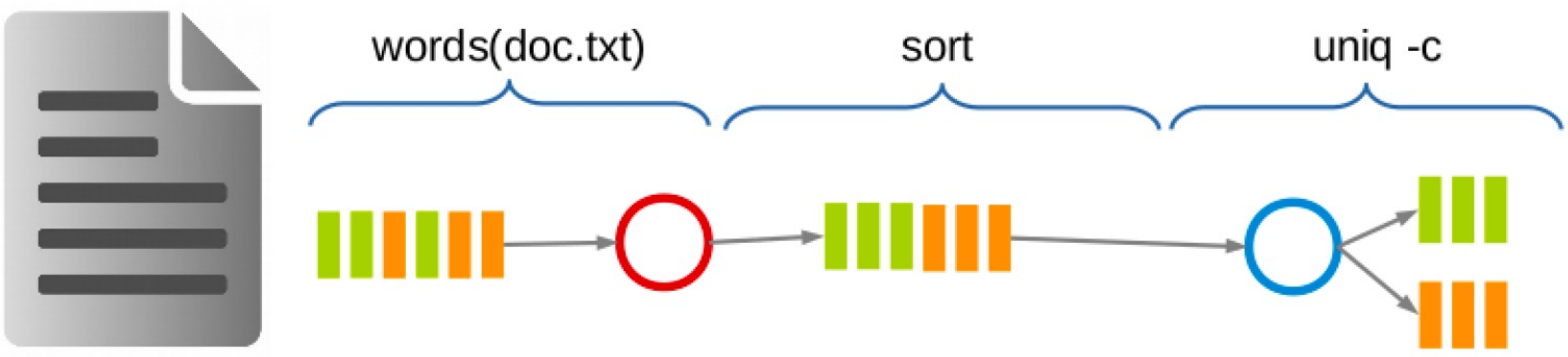
- A programming model: to batch process large data sets (inspired by functional programming).

- An execution framework: to run parallel algorithms on clusters of commodity hardware.

Programming Model

Word Count

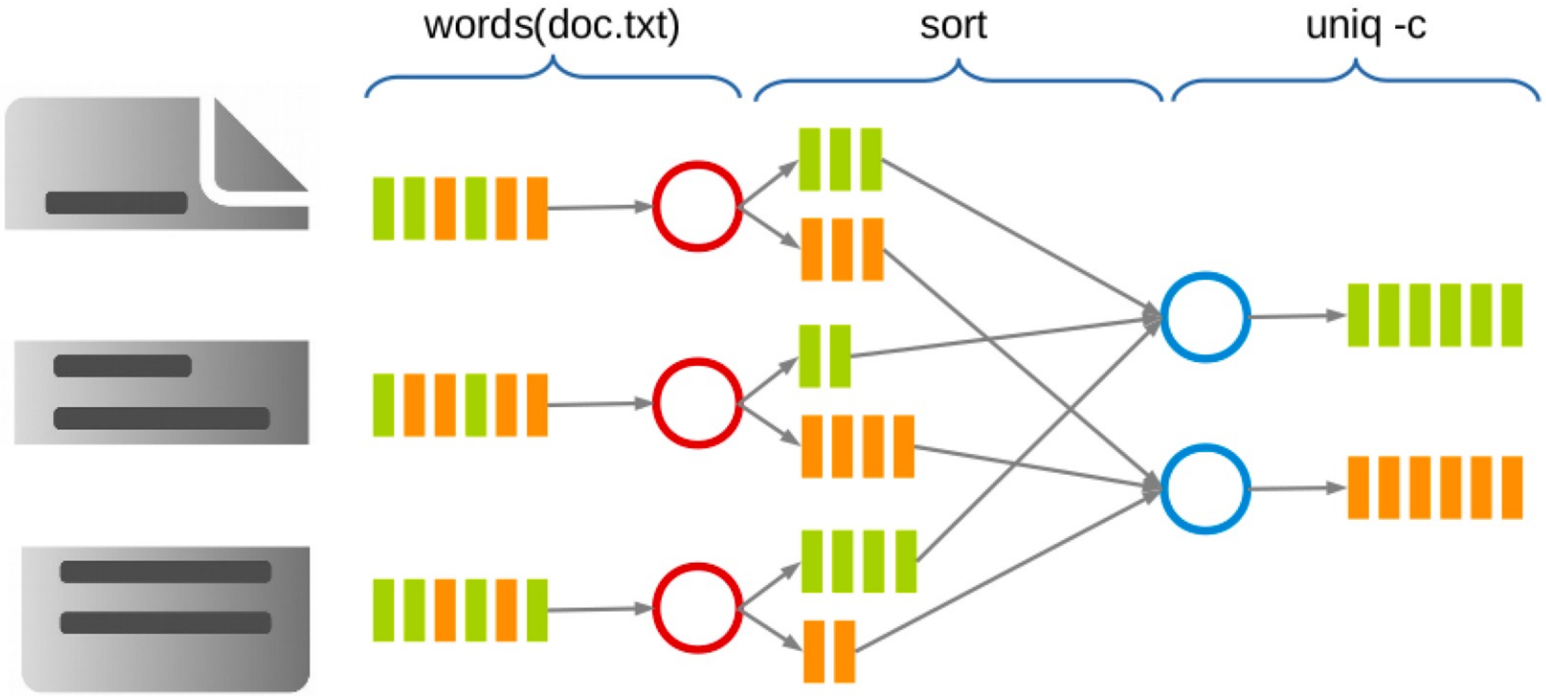
- Count the number of times each distinct word appears in the file
- If the file fits in memory: `words(doc.txt) | sort | uniq -c`



- If not?

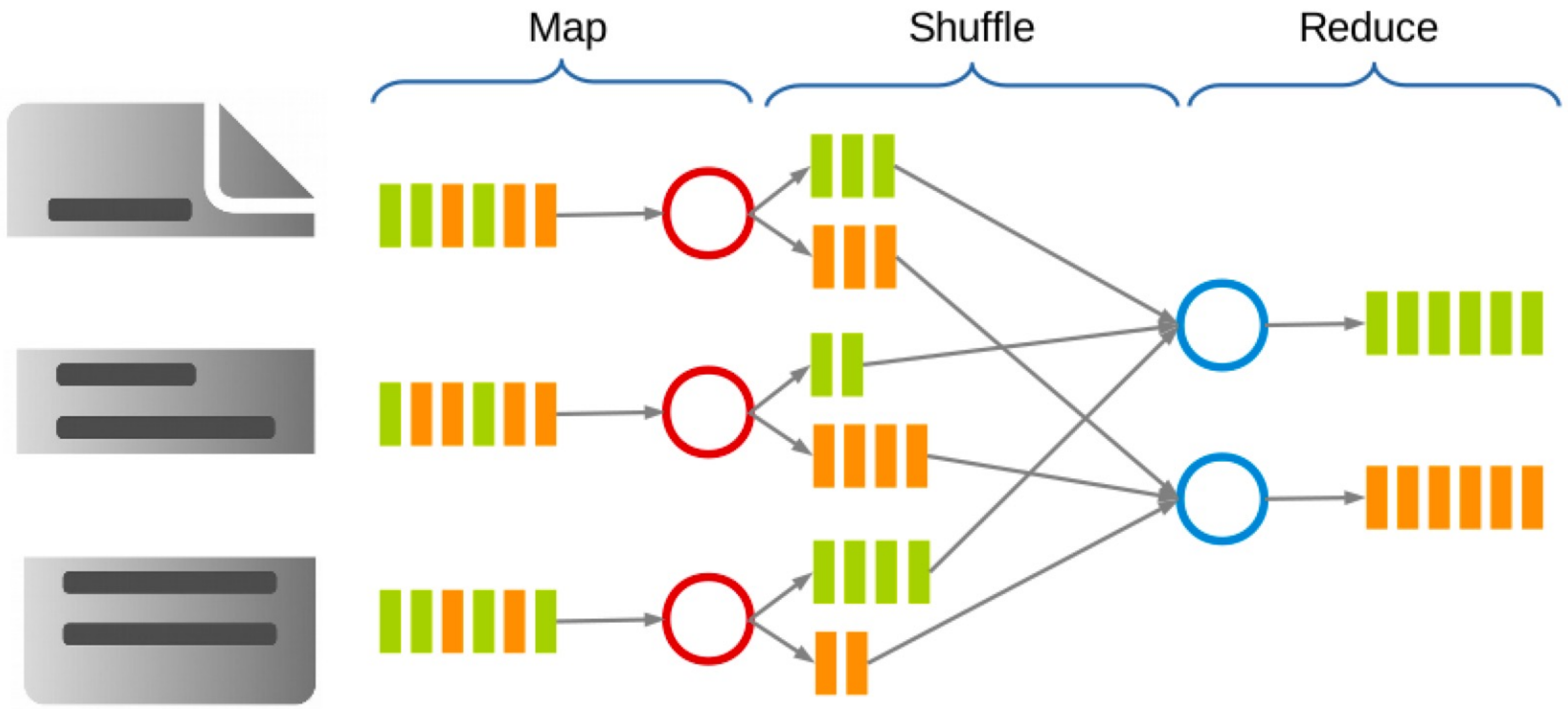
Data Parallel Processing

- Parallelizes data and processing



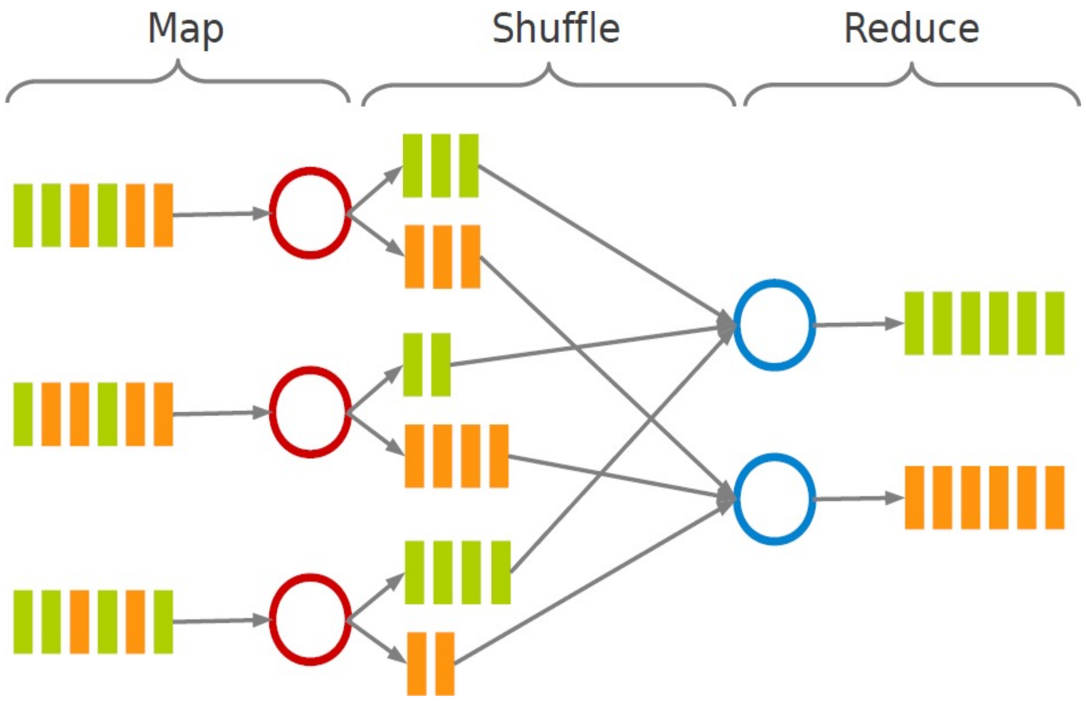
Data Parallel Processing

- MapReduce



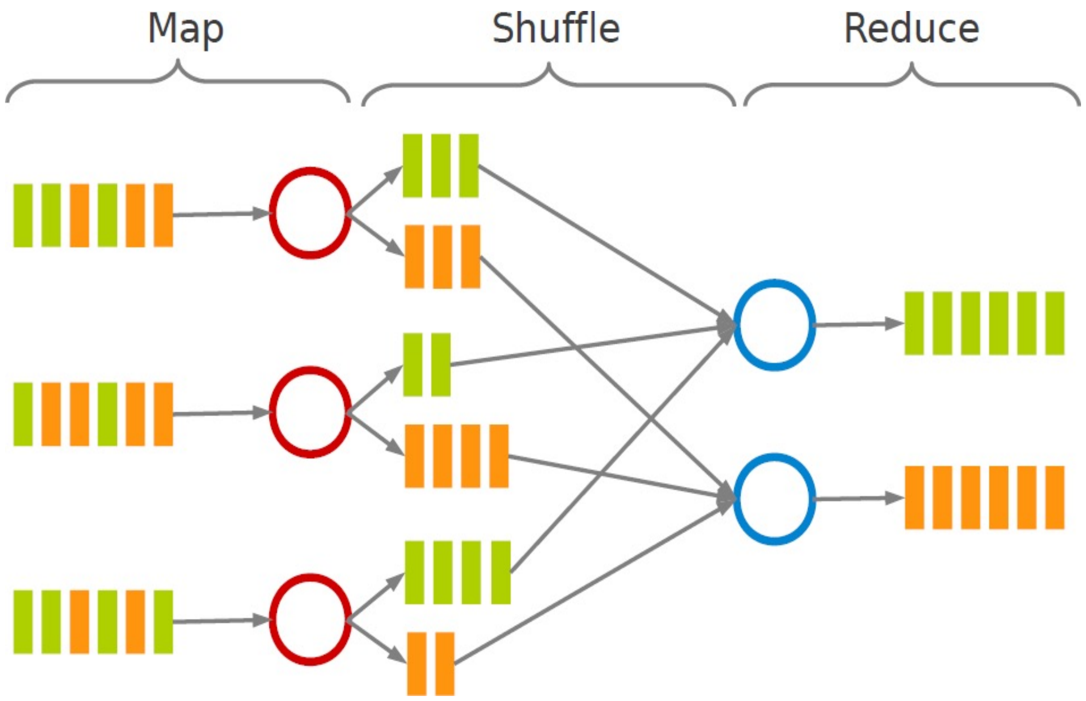
MapReduce Stages - Map

- Each Map task (typically) operates on a single HDFS block.
- Map tasks (usually) run on the node where the block is stored.
- Each Map task generates a set of intermediate key/value pairs.



MapReduce Stages – Shuffle and Sort

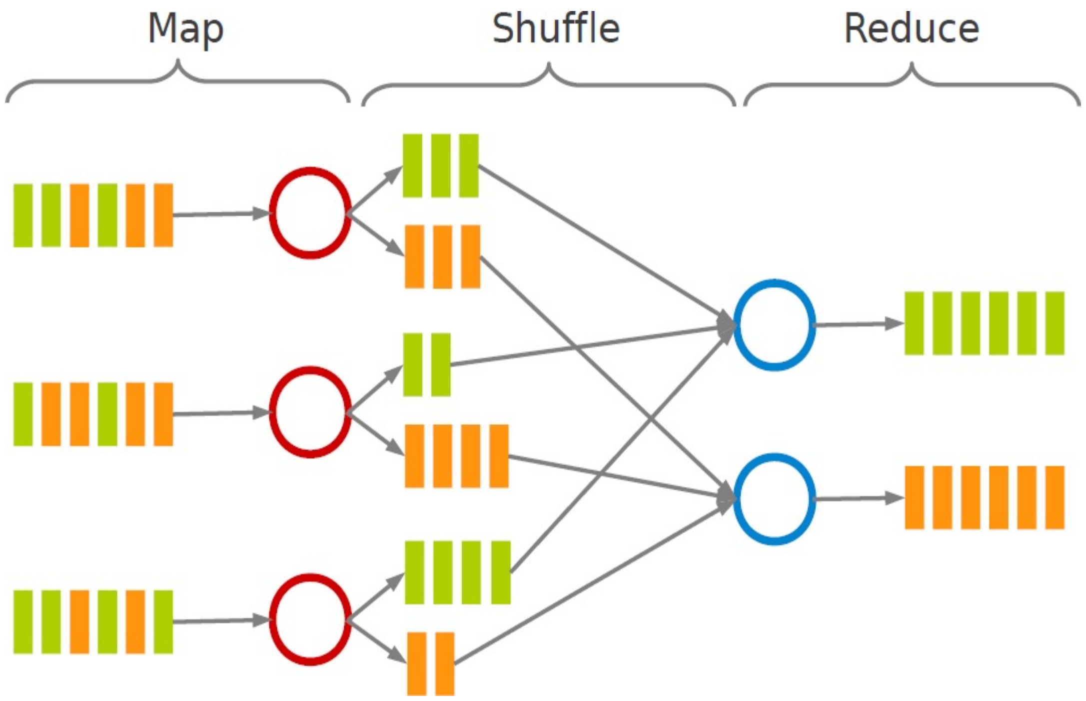
- Sorts and consolidates intermediate data from all mappers.
- Happens after all Map tasks are complete and before Reduce tasks start.



MapReduce Stages - Reduce

- Each Reduce task operates on all intermediate values associated with the same intermediate key.

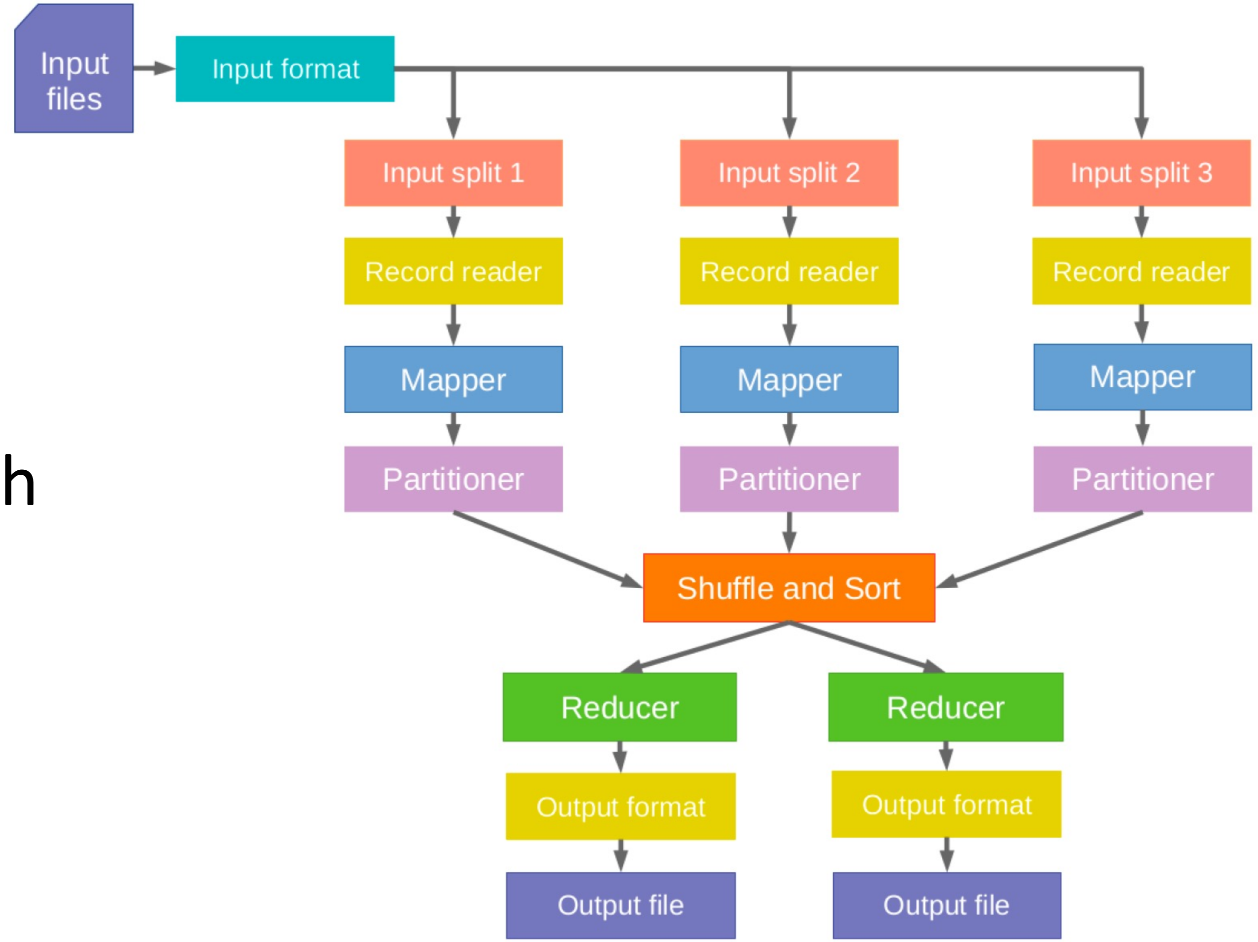
- Produces the final output.



MapReduce Data Flow

- **map** function: processes data and generates a set of intermediate key/value pairs.

- **reduce** function: merges all intermediate values associated with the same intermediate key.

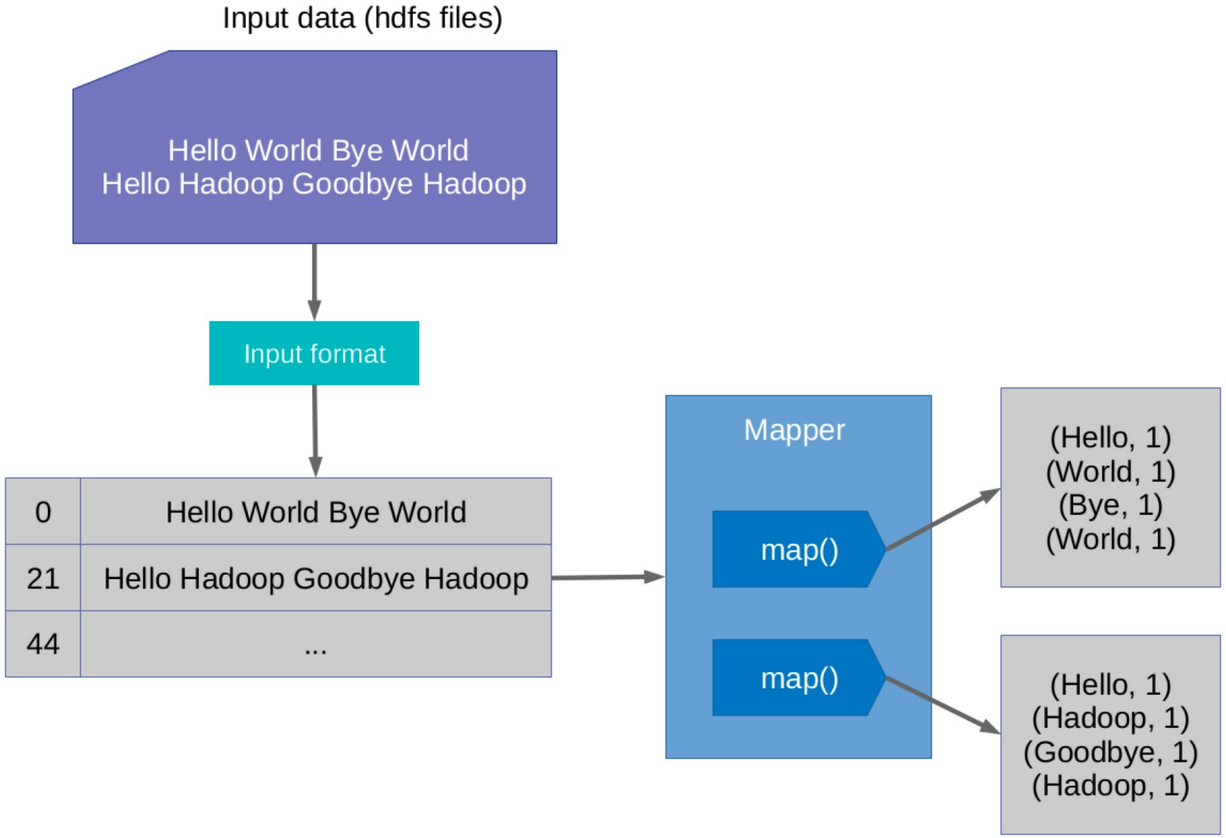


Example: Word Count

- Consider doing a word count of the following file using MapReduce:

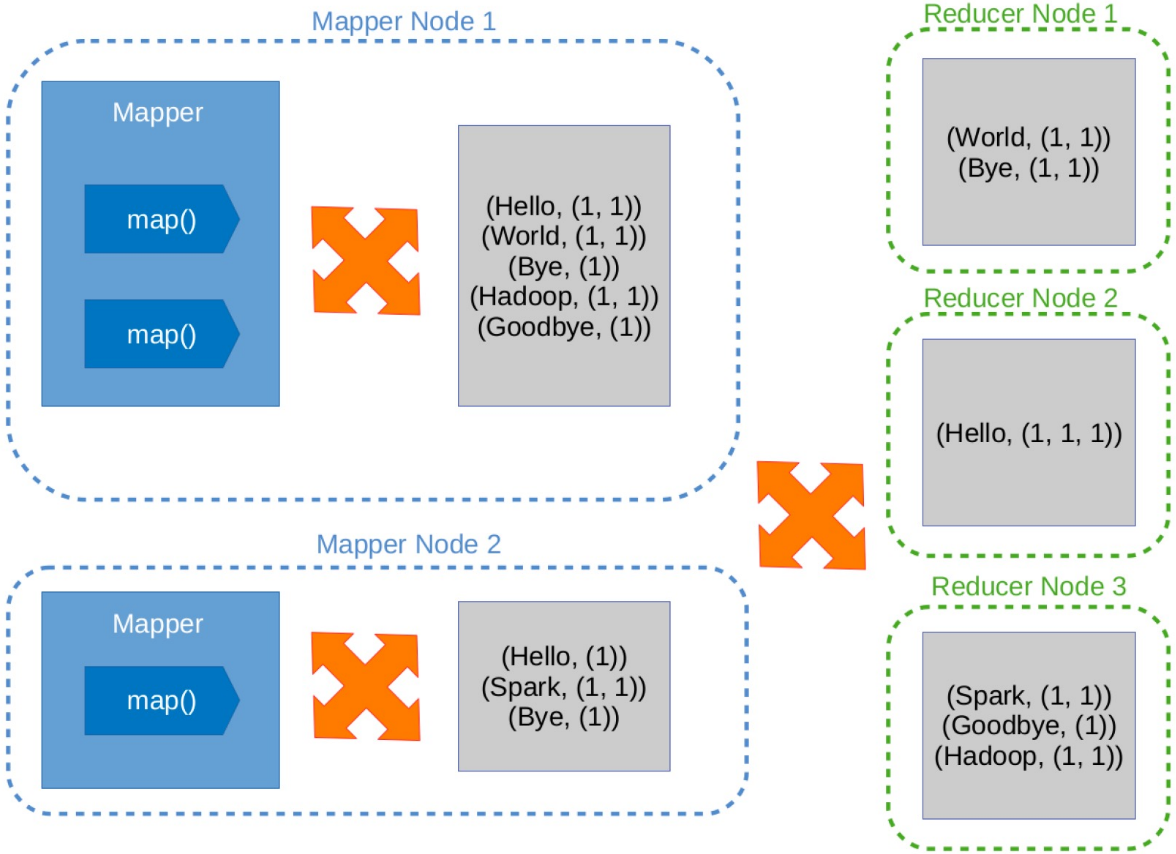
Example: Word Count - *Map*

- The map function reads in words one a time and outputs (word, 1) for each parsed input word.



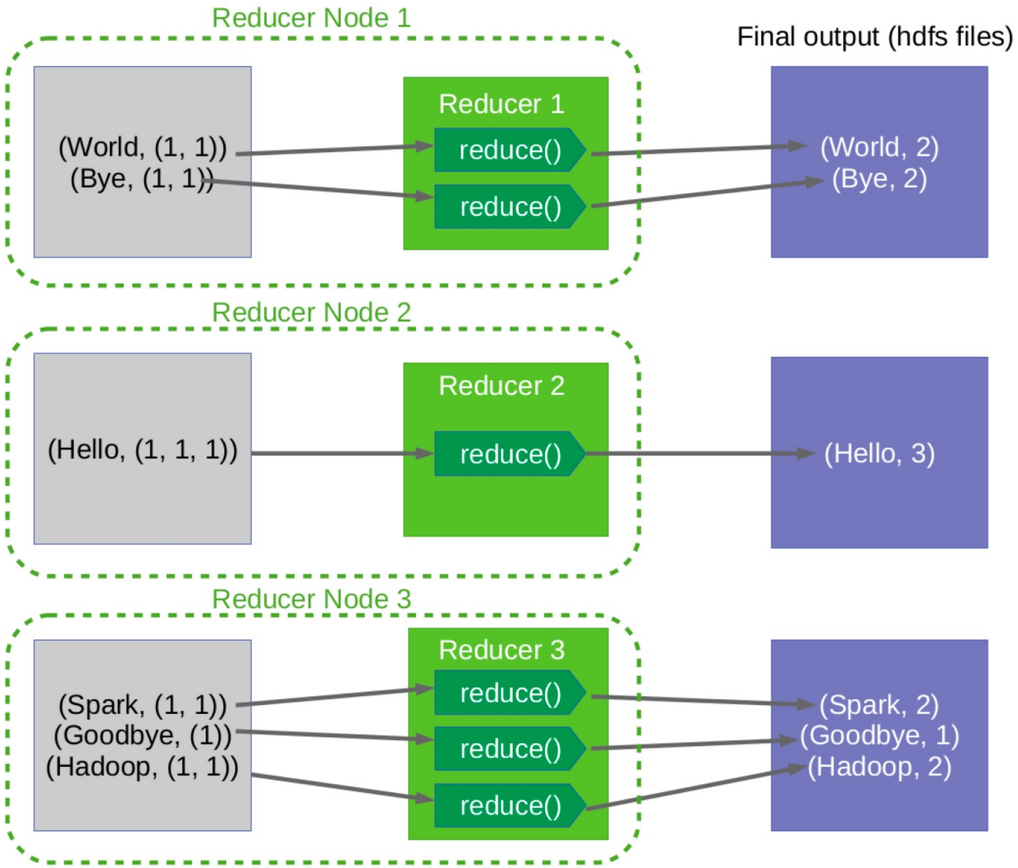
Example: Word Count - *Shuffle*

- The shuffle phase between map and reduce phase creates a list of values associated with each key.



Example: Word Count - *Reduce*

- The reduce function sums the numbers in the list for each key and outputs (word, count) pairs.



Example: Word count- *Map*

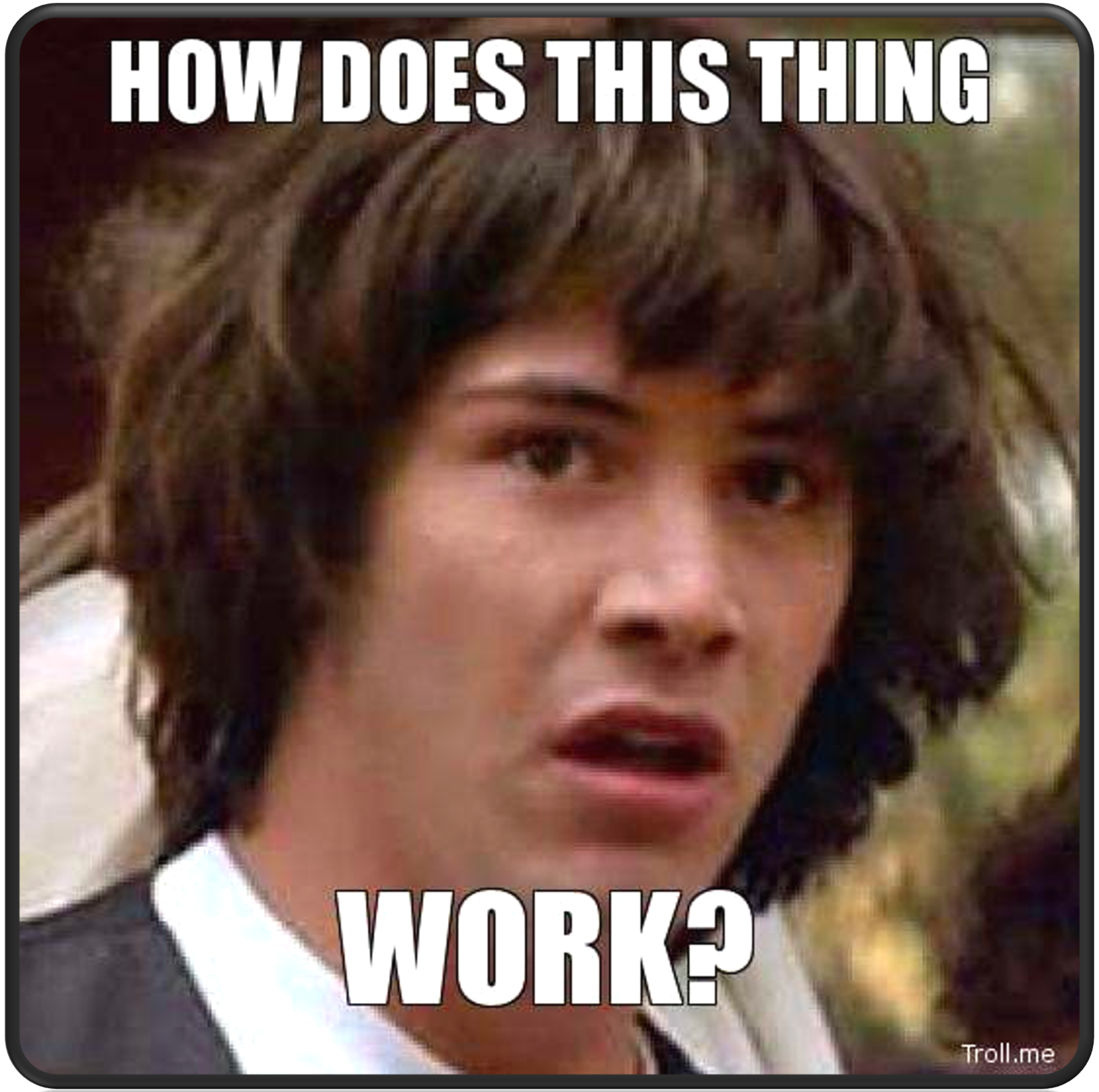
```
public static class MyMap extends Mapper<...> {
  private final static IntWritable one = new IntWritable(1); private Text
  word = new Text();

  public void map(LongWritable key, Text value, Context context) throws
    IOException, InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);

    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(word, one);
    }
  }
}
```

Example: Word count- *Reduce*

```
public static class MyReduce extends Reducer<...> {
  public void reduce(Text key, Iterator<...> values, Context context)
    throws IOException, InterruptedException {
    int sum = 0;

    while (values.hasNext())
      sum += values.next().get();

    context.write(key, new IntWritable(sum));
  }
}
```

Example: Word count- *Driver*

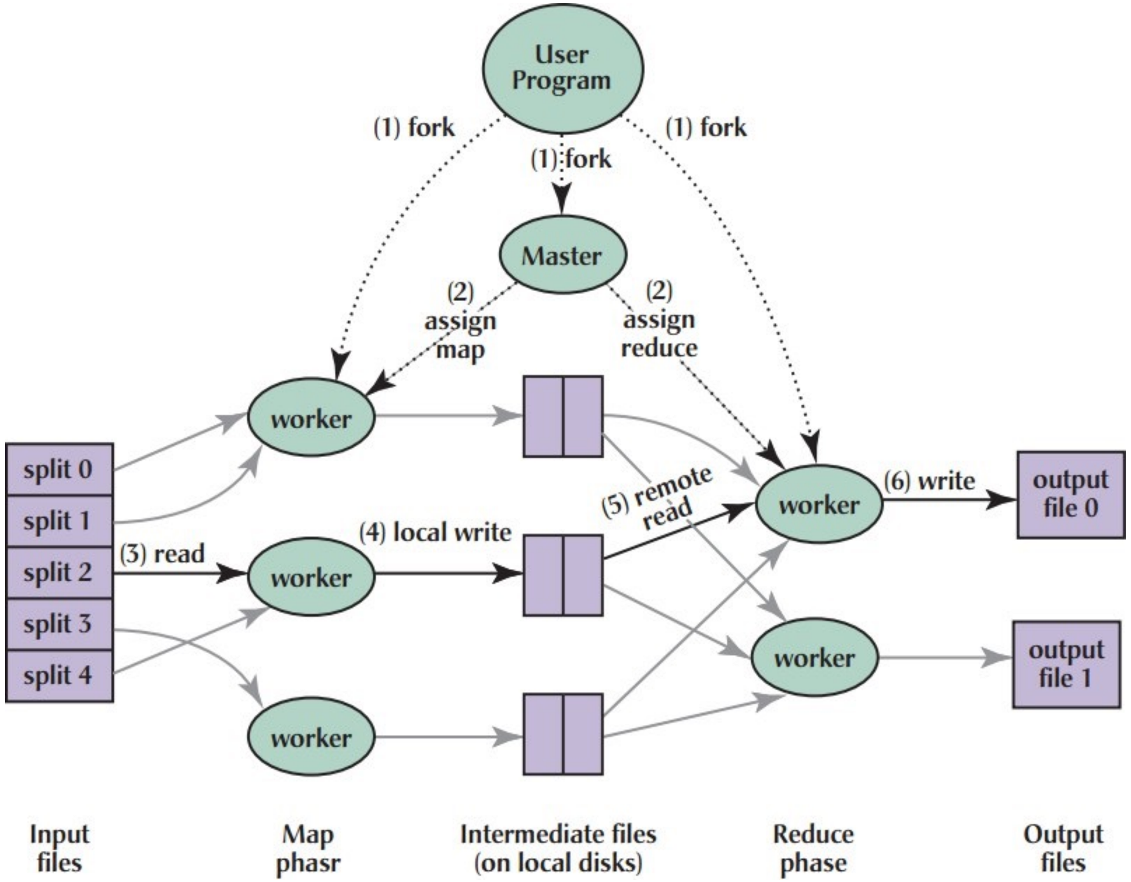
```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = new Job(conf, "wordcount");

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  job.setMapperClass(MyMap.class);
  job.setCombinerClass(MyReduce.class);
  job.setReducerClass(MyReduce.class);

  job.setInputFormatClass(TextInputFormat.class);
  job. setOutputFormatClass( TextOutputFormat. class) ;

  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.waitForCompletion(true);
}
```


HOW DOES THIS THING
WORK?
Troll.me

MapReduce
Execution Engine

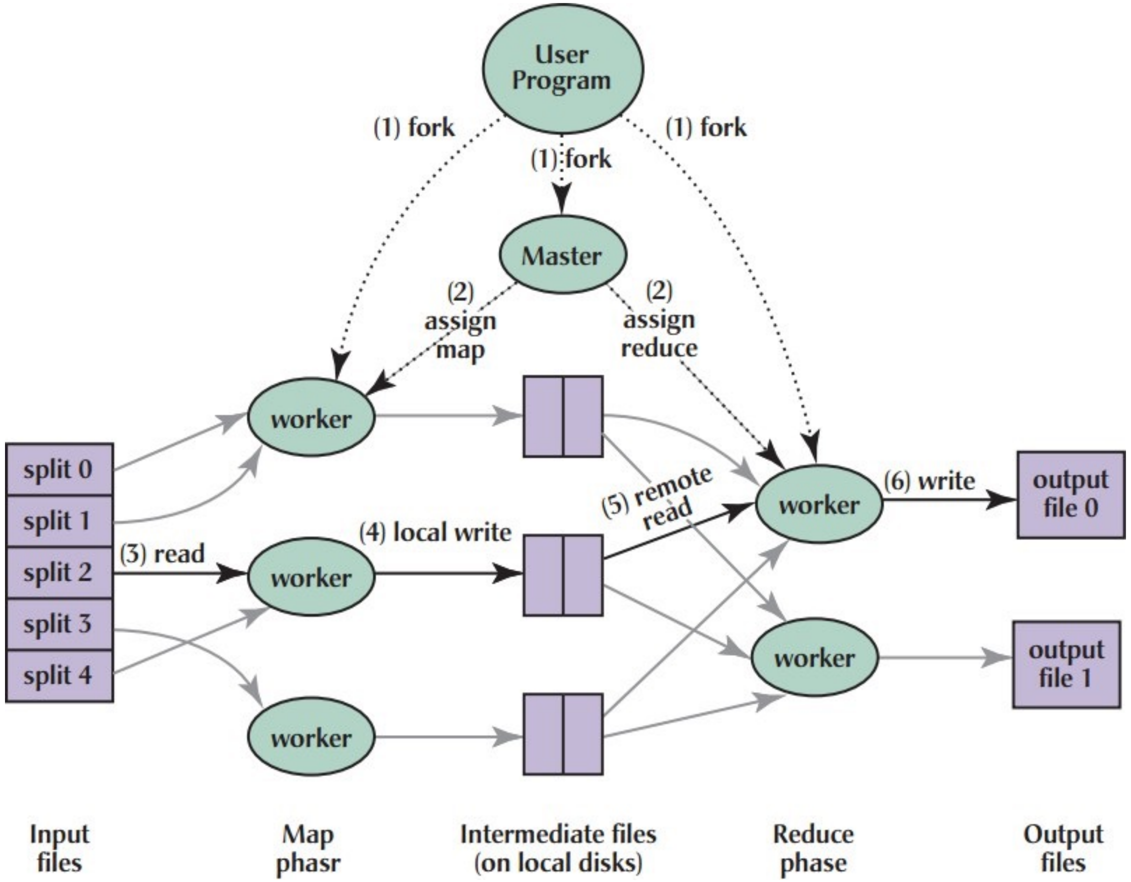
MapReduce Execution (1/7)

- The user program divides the input files into M splits.
  - A typical size of a split is the size of a HDFS block (64 -128MB).
  - Converts them to key/value pairs.
- It starts up many copies of the program on a cluster of machines.



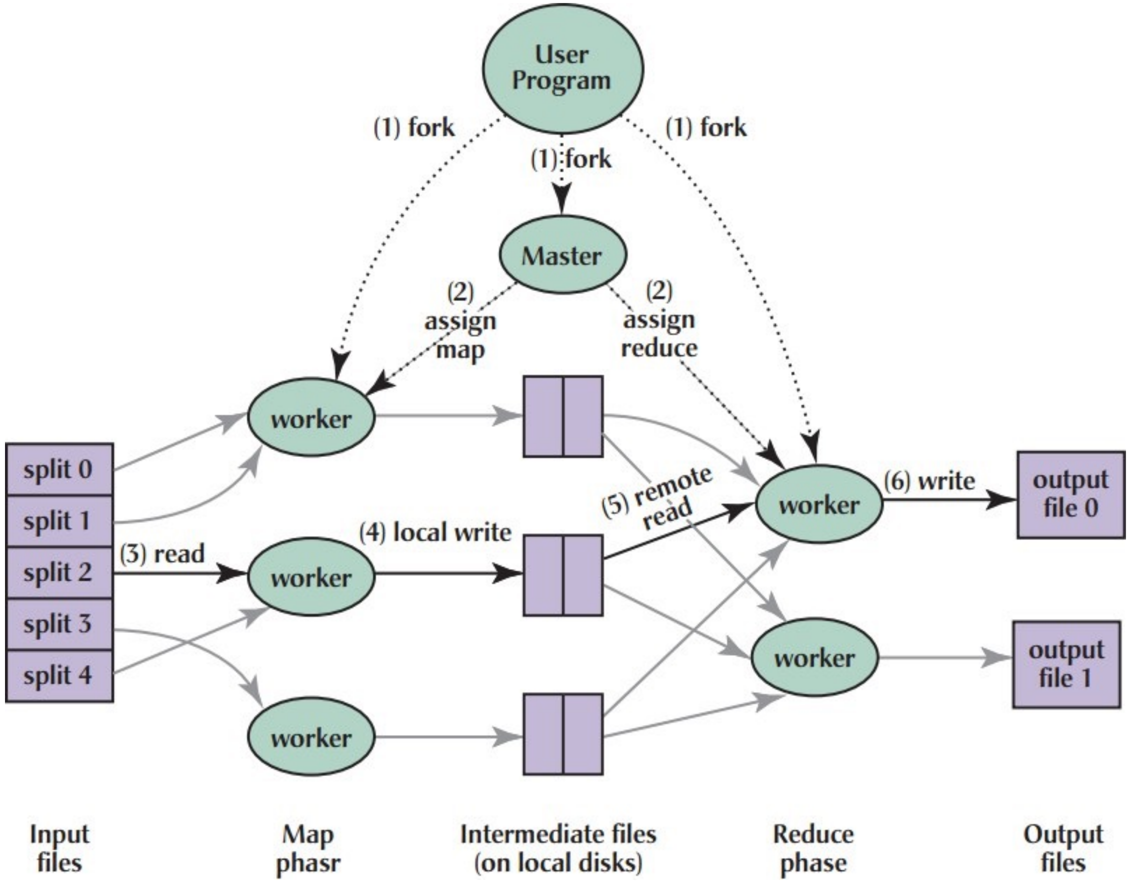
MapReduce Execution (2/7)

- One of the copies of the program is master, and the rest are workers.
- The master assigns works to the workers.
  - It picks idle workers and assigns each one a map task or a reduce task.



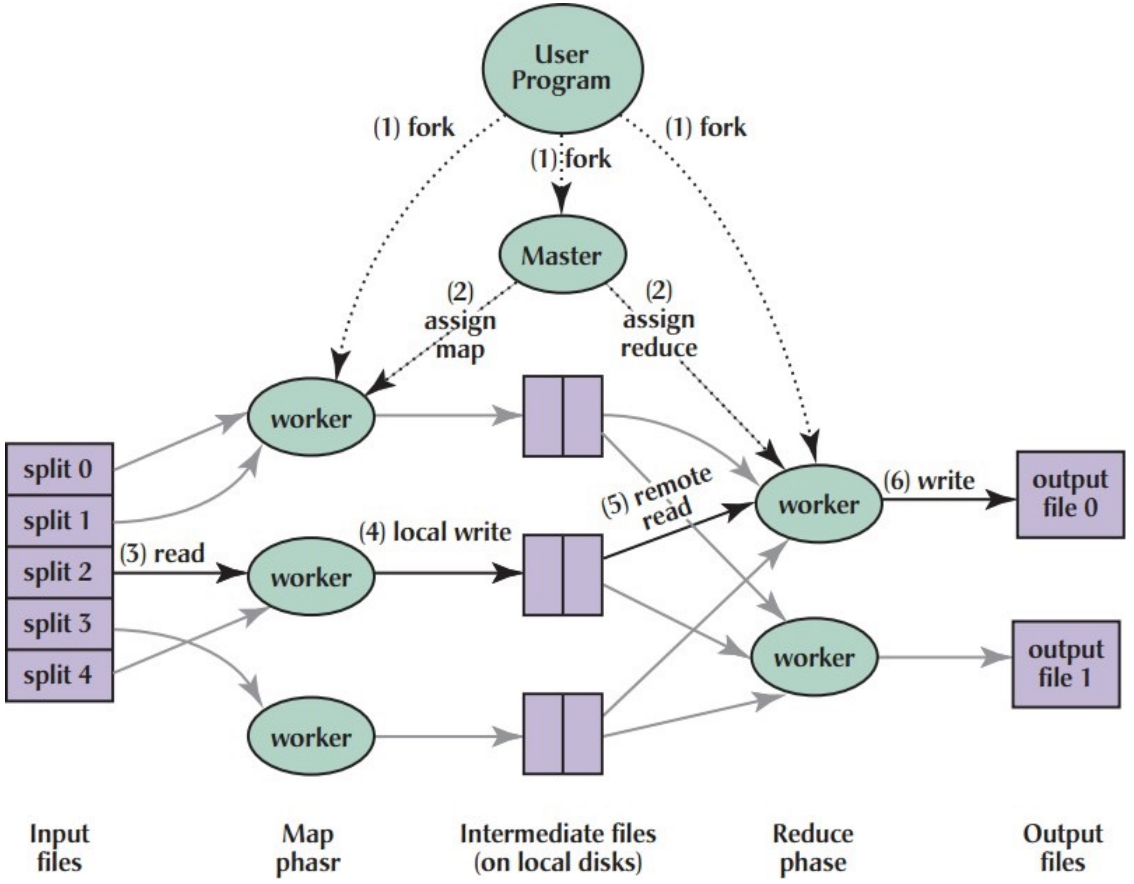
MapReduce Execution (3/7)

- A map worker reads the contents of the corresponding input splits.
- It parses key/value pairs out of the input data and passes each pair to the user defined map function.
- The intermediate key/value pairs produced by the map function are buffered in memory.



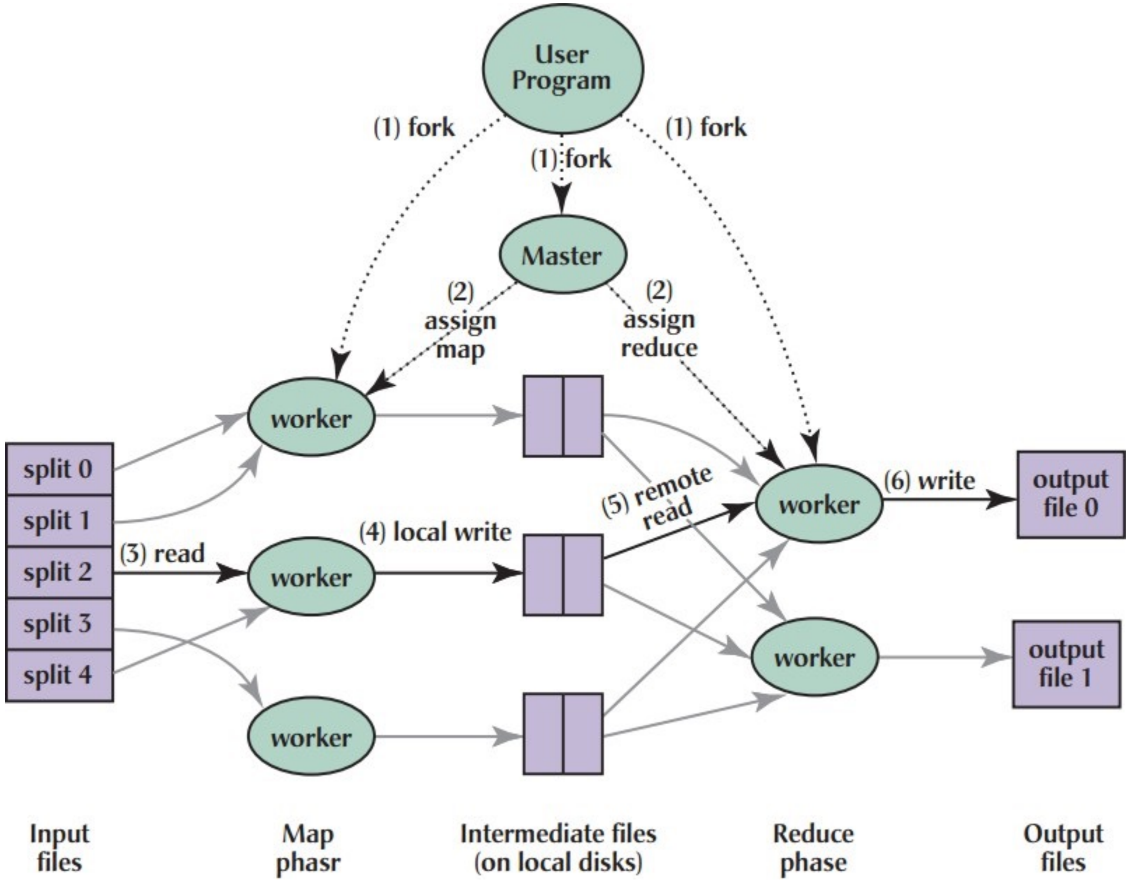
MapReduce Execution (4/7)

- The buffered pairs are periodically written to local disk.
- They are partitioned into R regions (hash(key) mod R).
- The locations of the buffered pairs on the local disk are passed back to the master.
- The master forwards these locations to the reduce workers.



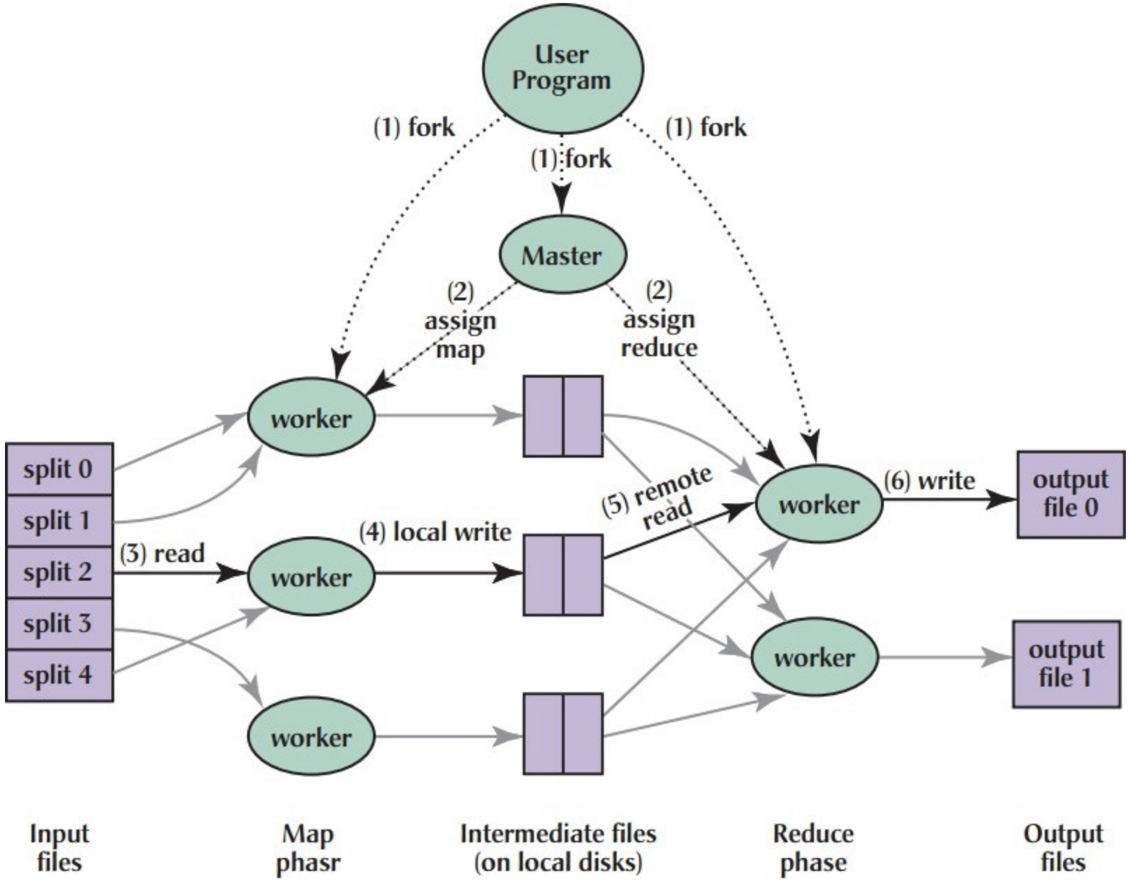
MapReduce Execution (5/7)

- A reduce worker reads the buffered data from the local disks of the map workers.
- When a reduce worker has read all intermediate data, it sorts it by the intermediate keys.



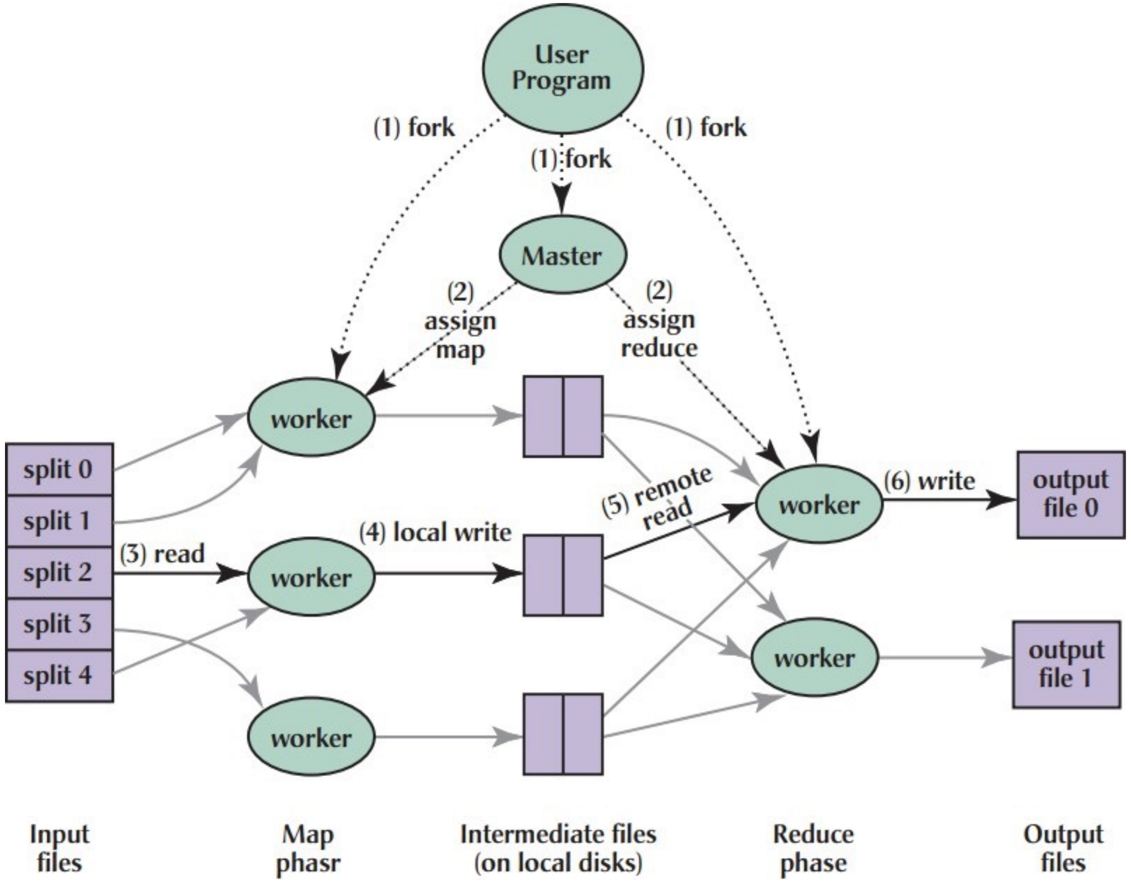
MapReduce Execution (6/7)

- The reduce worker iterates over the intermediate data.
- For each unique intermediate key, it passes the key and the corresponding set of intermediate values to the user defined reduce function.
- The output of the reduce function is appended to a final output file for this reduce partition.

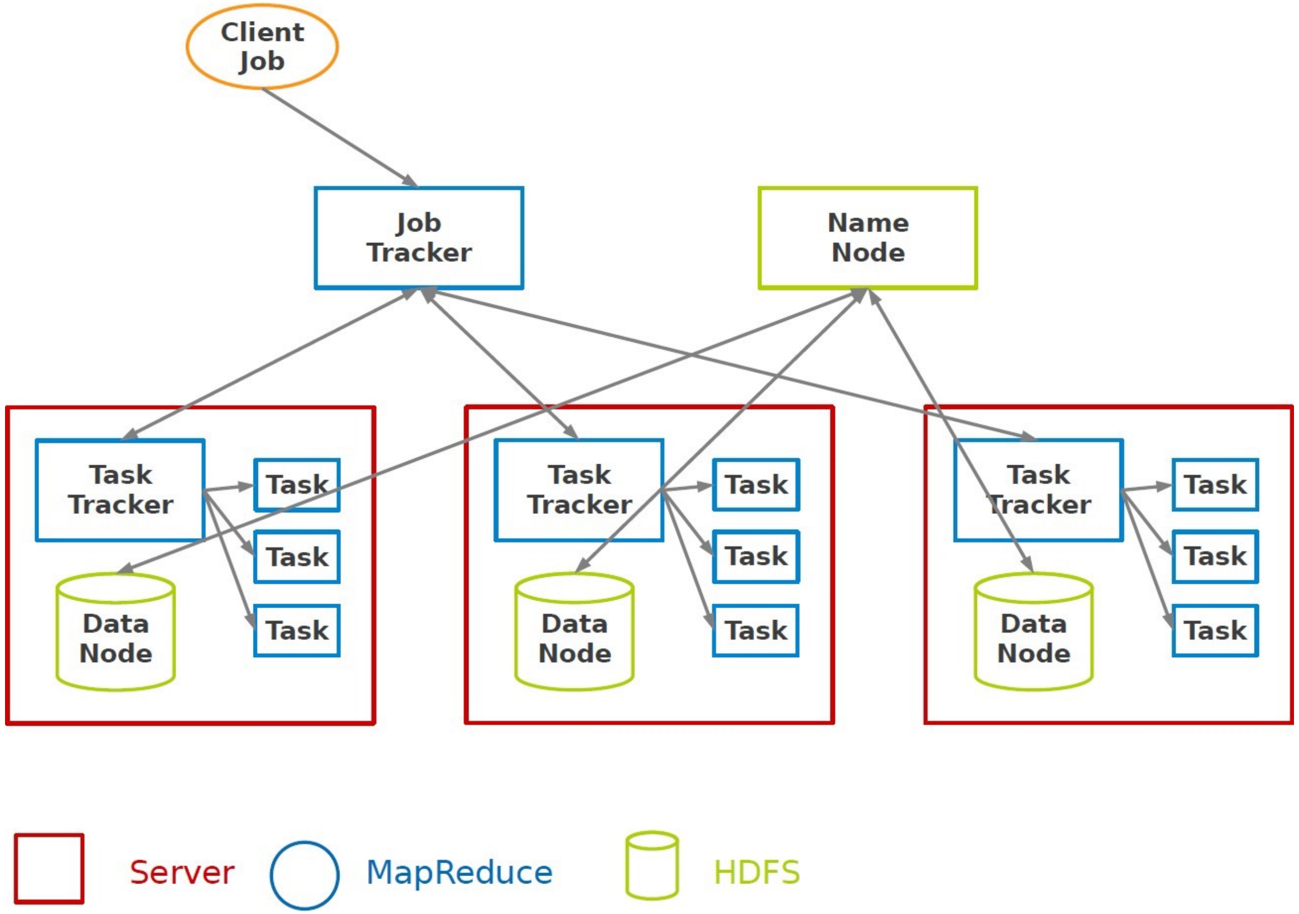


MapReduce Execution (7/7)

- When all map tasks and reduce tasks have been completed, the master wakes up the user program.



Hadoop MapReduce and HDFS



Fault tolerance - Worker

- Detect failure via periodic heartbeats.
- Re-execute in-progress map and reduce tasks.
- Re-execute completed map tasks: their output is stored on the local disk of the failed machine and is therefore inaccessible.
- Completed reduce tasks do not need to be re-executed since their output is stored in a global filesystem.

Fault tolerance - Master

- State is periodically checkpointed: a new copy of master starts from the last checkpoint state.

Is MapReduce Applicable on Every Function?

- It is easy in MapReduce:

  words(doc.txt) | sort | uniq -c

- What about this one?

  words(doc.txt) | grep | sed | sort | awk | perl

Next class:

Spark Execution Engine