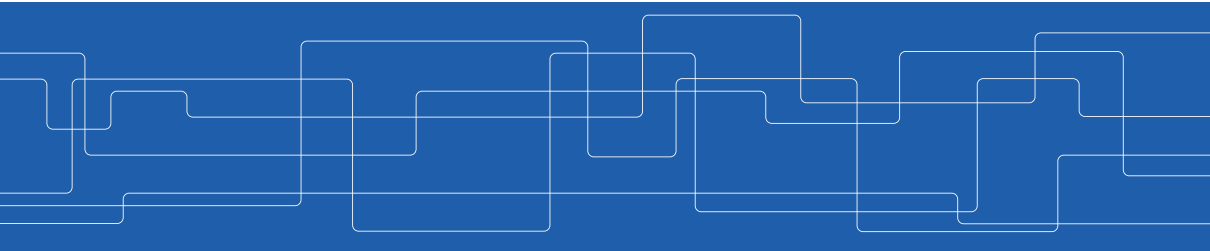
Parallel Processing - MapReduce and FlumeJava

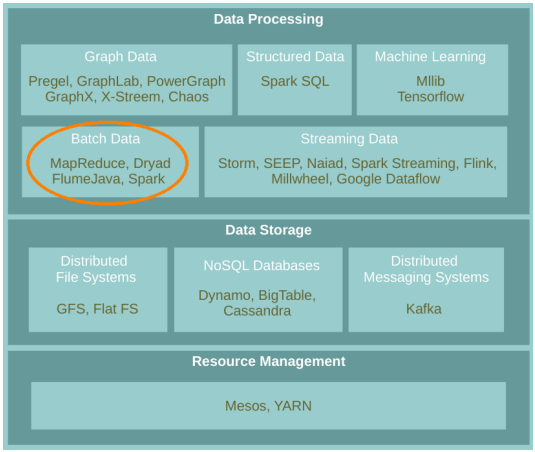
Amir H. Payberah
payberah@kth.se
10/09/2019



The Course Web Page

https://id2221kth.github.io

Where Are We?

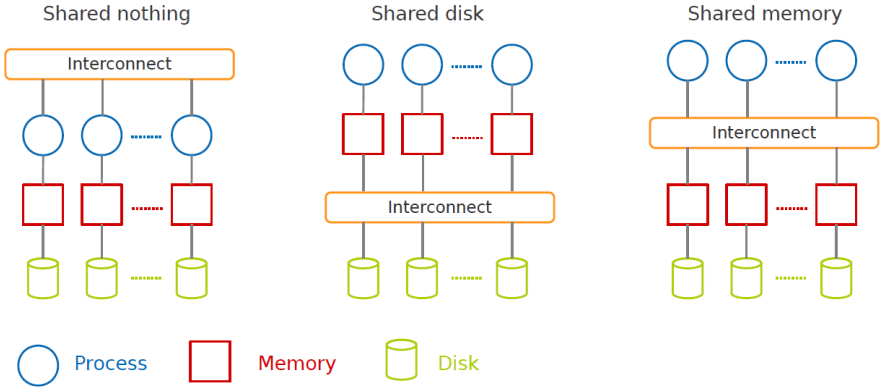

Data Processing
Graph Data
Pregel, GraphLab, PowerGraph
GraphX, X-Streem, Chaos
Structured Data
Spark SQL
Machine Learning
Mllib
Tensorflow
Batch Data
MapReduce, Dryad
FlumeJava, Spark
Streaming Data
Storm, SEEP, Naiad, Spark Streaming, Flink,
Millwheel, Google Dataflow
Data Storage
Distributed
File Systems
GFS, Flat FS
NoSQL Databases
Dynamo, BigTable,
Cassandra
Distributed
Messaging Systems
Kafka
Resource Management
Mesos, YARN

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.

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.

Challenges

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

Simplicity

- MapReduce takes care of parallelization, fault tolerance, and data distribution.
- Hide system-level details from programmers.

[http://www.johnlund.com/page/8358/elephant-on-a-scooter.asp]

MapReduce Definition

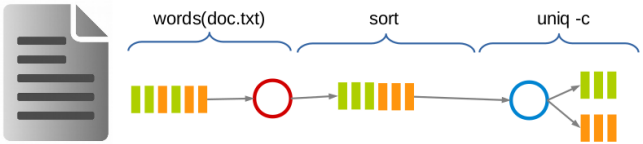
- A programming model: to batch process large data sets (inspired by functional programming).
- An execution framework: to run parallel algorithms on clusters of commodity hardware.

Programming Model


hello world
ICANHASCHEEZBURGER.COM

Word Count

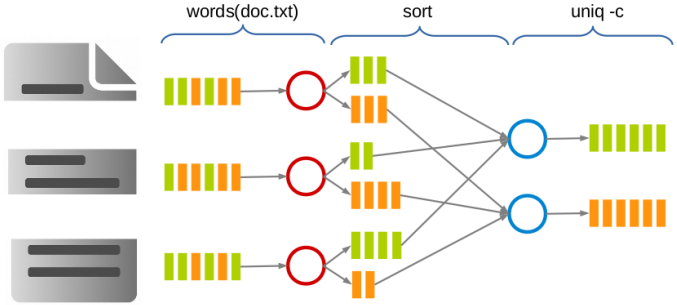
- Count the number of times each distinct word appears in the file
- If the file fits in memory: `words(doc.txt) | sort | uniq -c`



- If not?

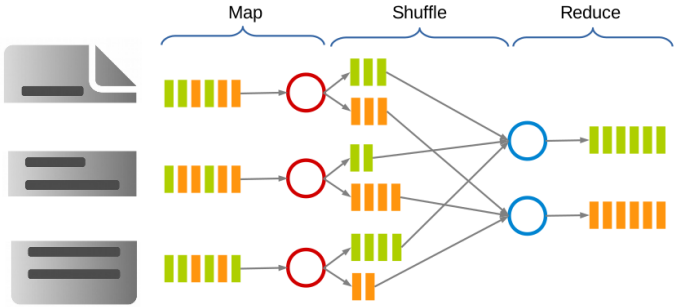
Data-Parallel Processing (1/2)

- Parallelize the data and process.



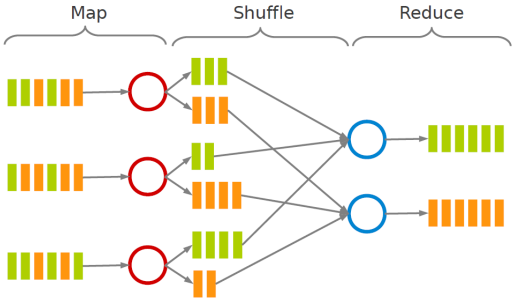
Data-Parallel Processing (2/2)

- MapReduce



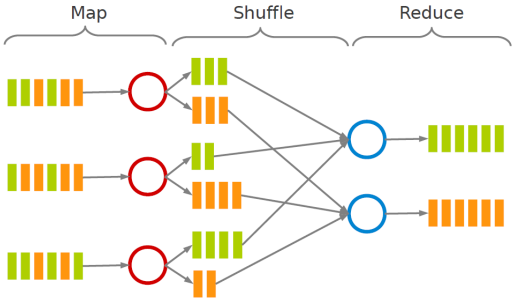
MapReduce Stages - Map

- 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.



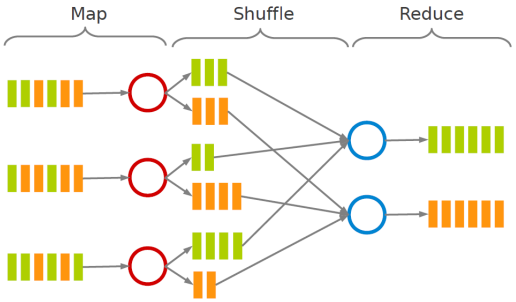
MapReduce Stages - Shuffle and Sort

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



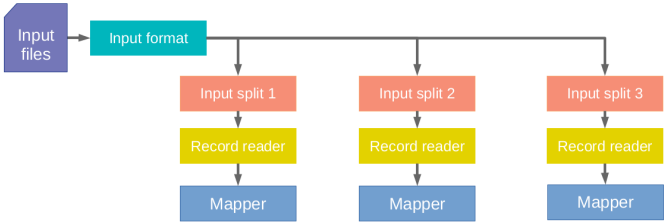
MapReduce Stages - Reduce

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

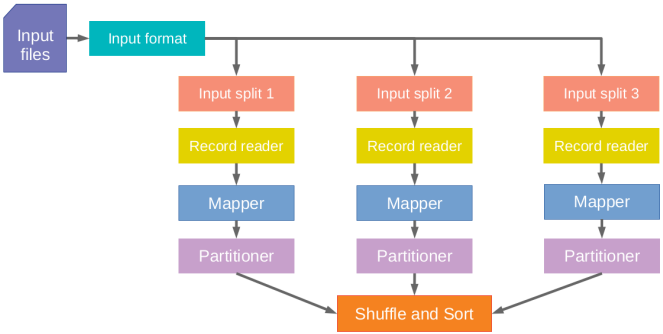


MapReduce Data Flow (1/5)

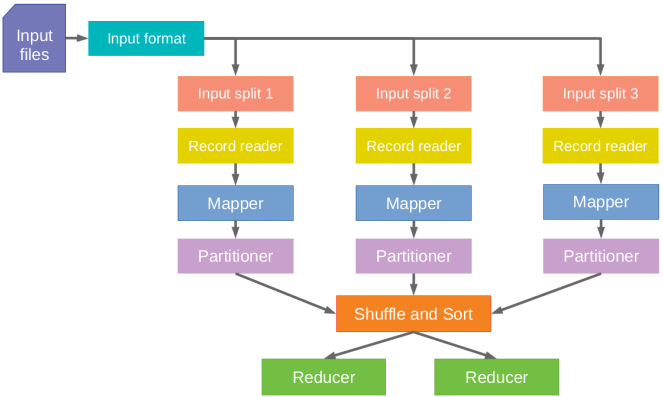
MapReduce Data Flow (2/5)



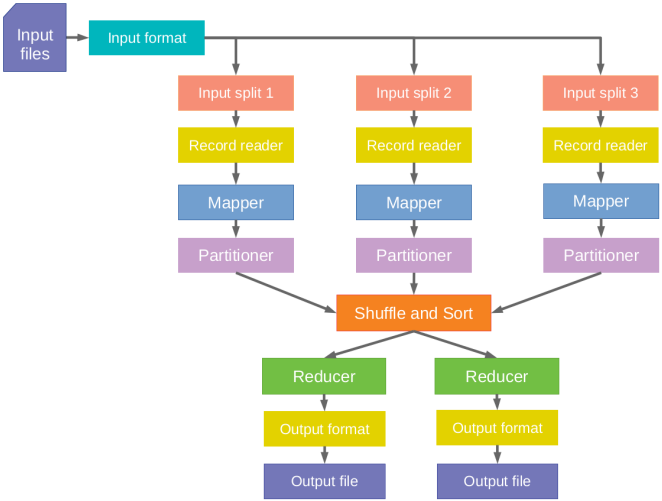
MapReduce Data Flow (3/5)



MapReduce Data Flow (4/5)

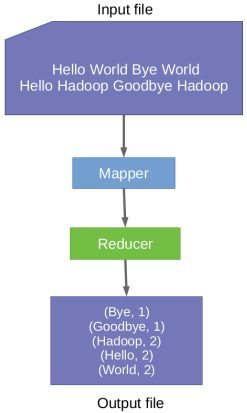


MapReduce Data Flow (5/5)

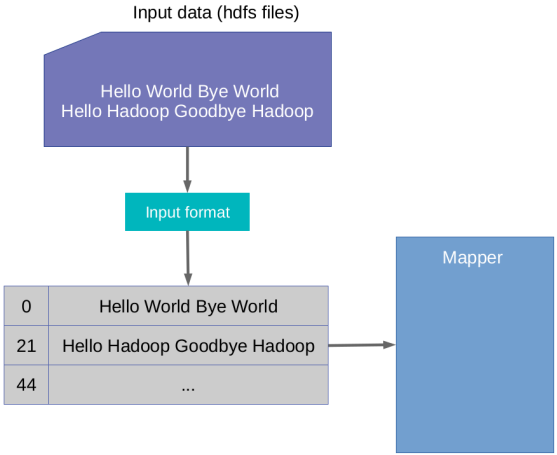


Word Count in MapReduce

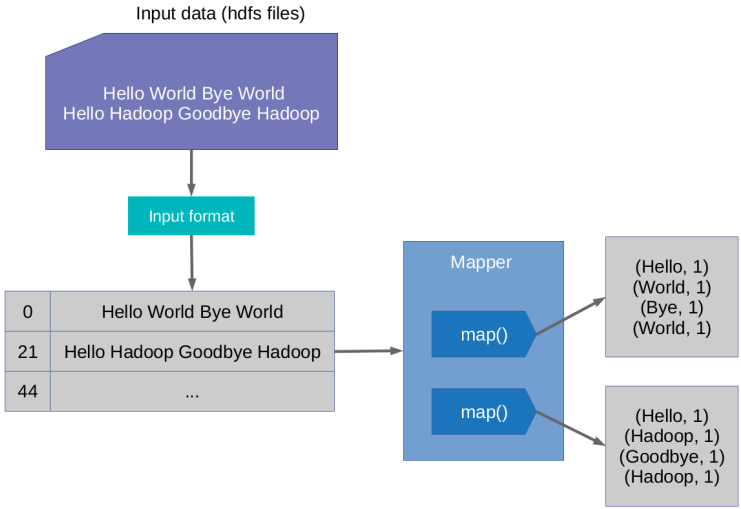
- Consider doing a word count of the following file using MapReduce



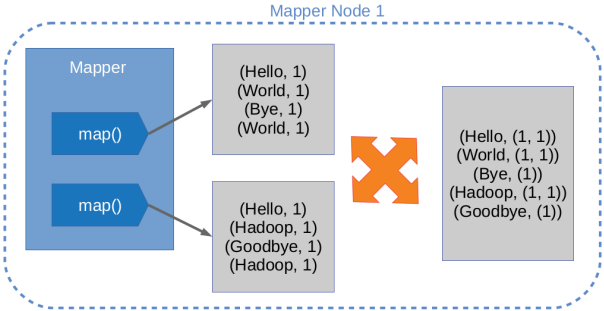
Word Count in MapReduce - Map (1/2)



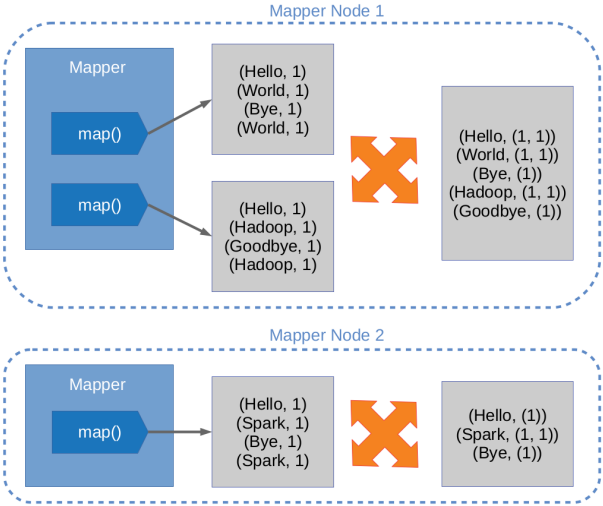
Word Count in MapReduce - Map (2/2)



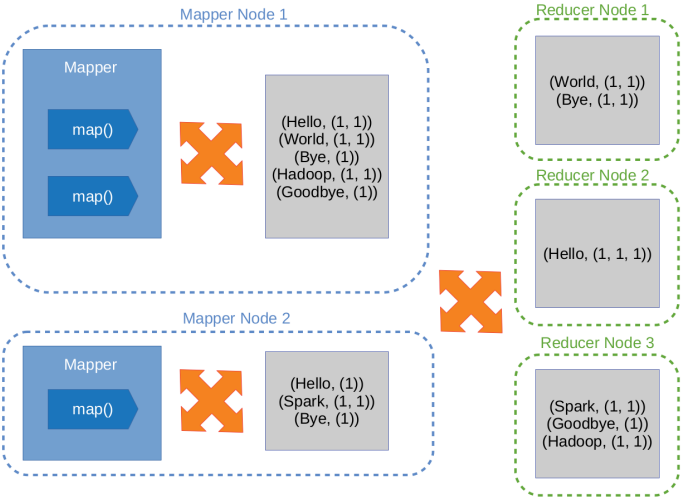
Word Count in MapReduce - Shuffle and Sort (1/3)



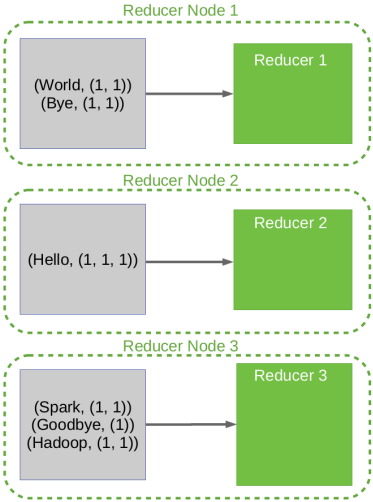
Word Count in MapReduce - Shuffle and Sort (2/3)



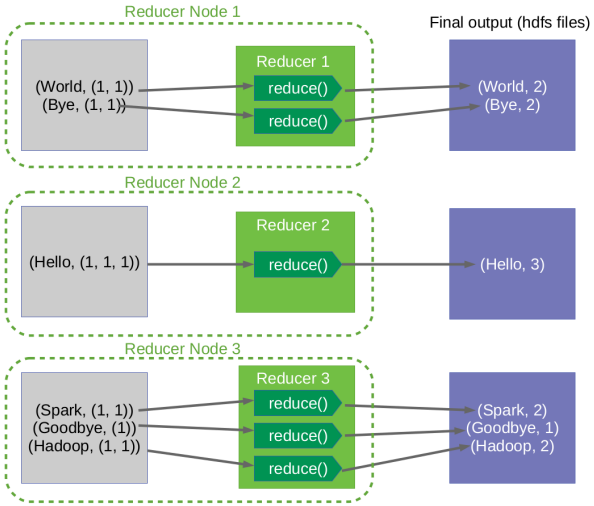
Word Count in MapReduce - Shuffle and Sort (3/3)



Word Count in MapReduce - Reduce (1/2)



Word Count in MapReduce - Reduce (2/2)



Mapper

The Mapper

- Input: (key, value) pairs
- Output: a list of (key, value) pairs
- The Mapper may use or completely ignore the input key.
- A standard pattern is to read one line of a file at a time.
  - Key: the byte offset
  - Value: the content of the line

```
map(in_key, in_value) -> list of (inter_key, inter_value)
```

```
(in_key, in_value) ⇒ map() ⇒ (inter_key1, inter_value1)
                             (inter_key2, inter_value2)
                             (inter_key3, inter_value3)
                             (inter_key4, inter_value4)
                              ...
```

The Mapper Example (1/3)

- Turn input into upper case

```
map(k, v) = emit (k.to_upper, v.to_upper)
```

(kth, this is the course id2221) ⇒ map() ⇒ (KTH, THIS IS THE COURSE ID2221)

The Mapper Example (2/3)

- Count the number of characters in the input

```
map(k, v) = emit (k, v.length)
```

(kth, this is the course id2221) ⇒ map() ⇒ (kth, 26)

The Mapper Example (3/3)

- Turn each word in the input into pair of (word, 1)

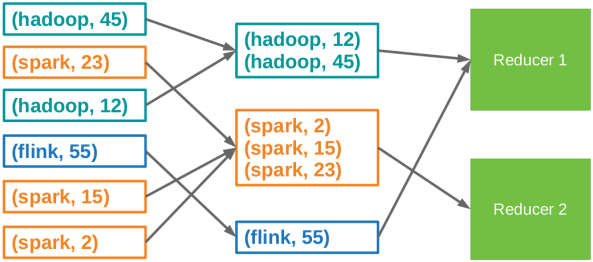
```
map(k, v) = foreach w in v emit (w, 1)
```

```
(21, Hello Hadoop Goodbye Hadoop) ⇒ map() ⇒ (Hello, 1)
                                             (Hadoop, 1)
                                             (Goodbye, 1)
                                             (Hadoop, 1)
```

Reducer

Shuffle and Sort

- After the Map phase, all intermediate (key, value) pairs are grouped by the intermediate keys.
- Each (key, list of values) is passed to a Reducer.



The Reducer

- Input: (key, list of values) pairs
- Output: a (key, value) pair or list of (key, value) pairs
- The Reducer outputs zero or more final (key, value) pairs

```
reduce(inter_key, [inter_value1, inter_value2, ...]) -> (out_key, out_value)
```

`(inter_k, [inter_v1, inter_v2, ···]) ⇒ reduce() ⇒ (out_k, out_v)`

or

```
(inter_k, [inter_v1, inter_v2, ···]) ⇒ reduce() ⇒ (out_k, out_v1)
                                                  (out_k, out_v2)
                                                   ...
```

The Reducer Example (1/3)

- Add up all the values associated with each intermediate key

```
reduce(k, vals) = {
  sum = 0
  foreach v in vals sum += v
  emit (k, sum)
}
```

(Hello, [1, 1, 1]) ⇒ reduce() ⇒ (Hello, 3)

(Bye, [1]) ⇒ reduce() ⇒ (Bye, 1)

The Reducer Example (2/3)

- Get the maximum value of each intermediate key

```
reduce(k, vals) = emit (k, max(vals))
```

(KTH, [5, 1, 12, 7]) ⇒ `reduce()` ⇒ (KTH, 12)

The Reducer Example (3/3)

- Identify reducer

```
reduce(k, vals) = foreach v in vals emit (k, v))
```

```
(KTH, [5, 1, 12, 7]) ⇒ reduce() ⇒ (KTH, 5)
                                   (KTH, 1)
                                   (KTH, 12)
                                   (KTH, 7)
```

Example: Word Count - map

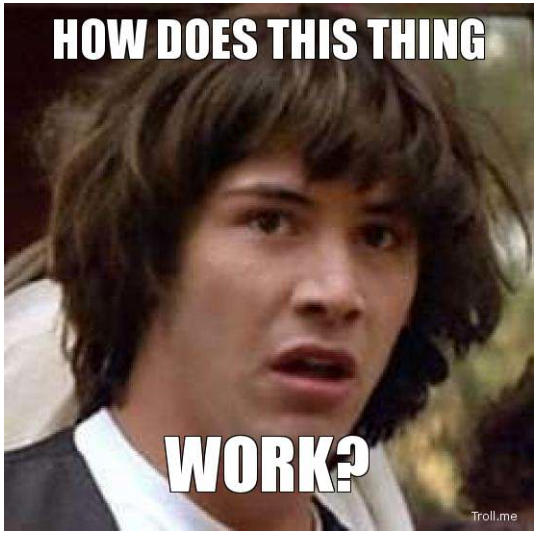
```
public static class MyMap extends Mapper<...> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  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));
  }
}
```

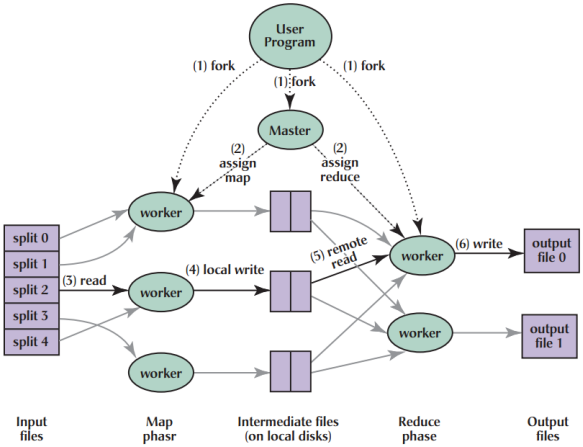
Example: Word Count - driver

```
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

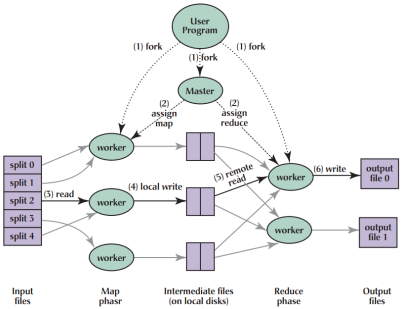
Implementation

Architecture



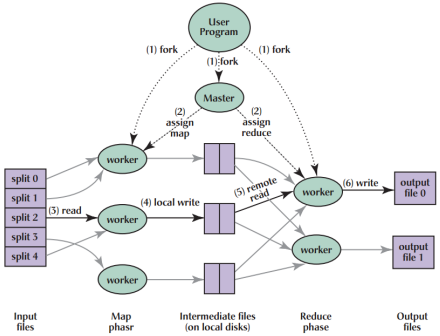
MapReduce Execution (1/7)

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



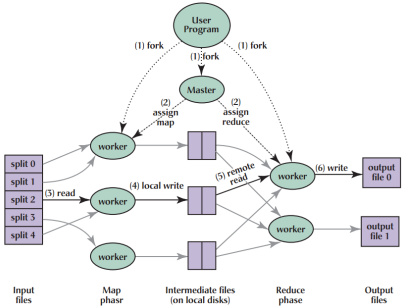
MapReduce Execution (2/7)

- 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.



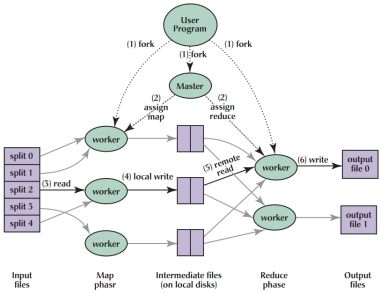
MapReduce Execution (3/7)

- 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 key/value pairs produced by the map function are buffered in memory.



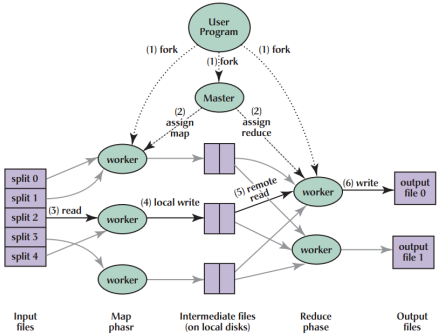
MapReduce Execution (4/7)

- 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.



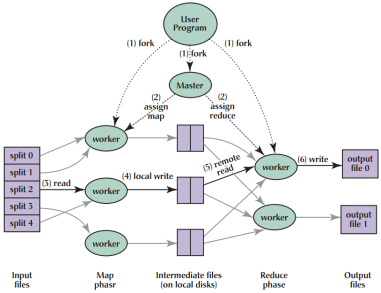
MapReduce Execution (5/7)

- 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.



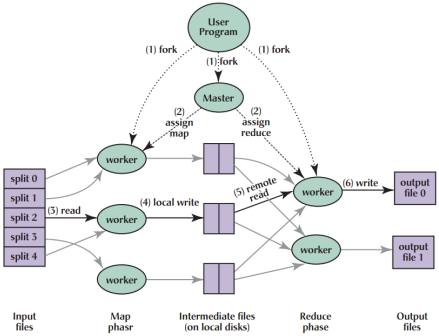
MapReduce Execution (6/7)

- 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.

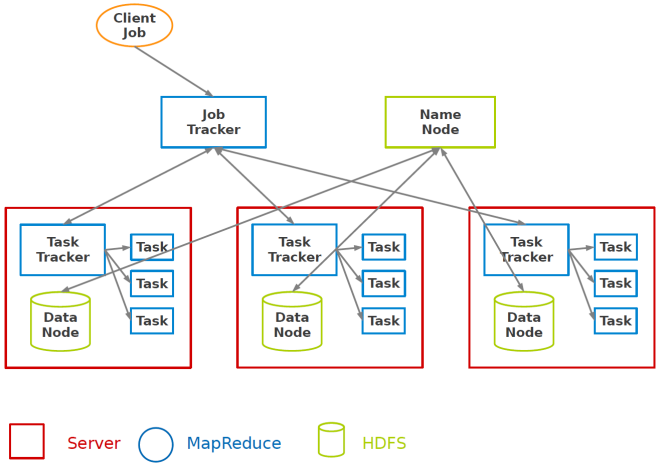


MapReduce Execution (7/7)

- 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.

MapReduce Algorithm Design

MapReduce Algorithm Design

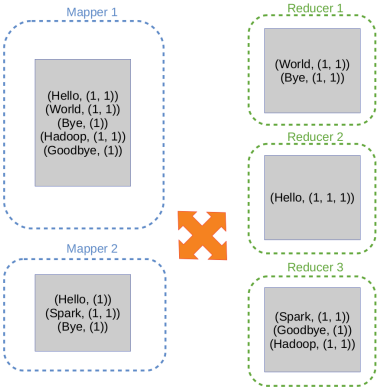
- Local aggregation
- Joining
- Sorting

MapReduce Algorithm Design

- Local aggregation
- Joining
- Sorting

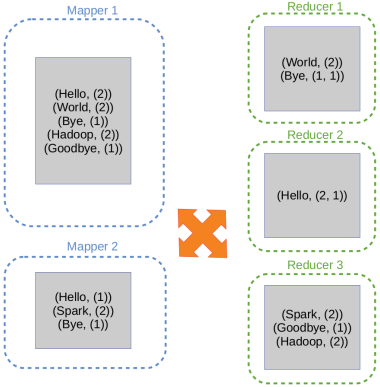
Local Aggregation - In-Map Combiner (1/2)

- In some cases, there is significant repetition in the intermediate keys produced by each map task, and the reduce function is commutative and associative.



Local Aggregation - In-Map Combiner (2/2)

- Merge partially data before it is sent over the network to the reducer.
- Typically the same code for the combiner and the reduce function.

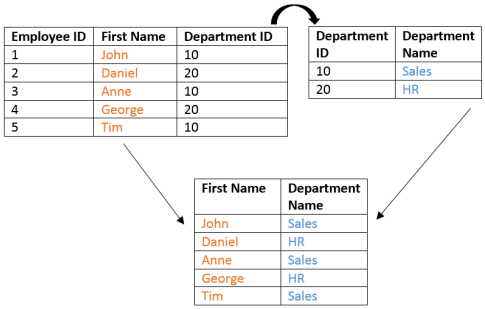


MapReduce Algorithm Design

- Local aggregation
- Joining
- Sorting

Joins

- Joins are relational constructs you use to combine relations together.
- In MapReduce joins are applicable in situations where you have two or more datasets you want to combine.



Employee ID	First Name	Department ID
1	John	10
2	Daniel	20
3	Anne	10
4	George	20
5	Tim	10

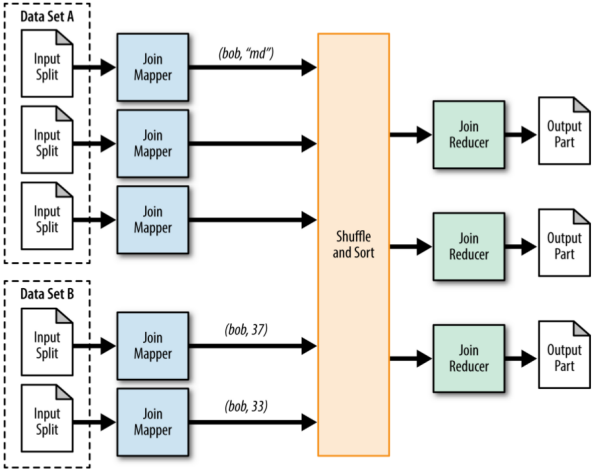
Department ID	Department Name
10	Sales
20	HR

First Name	Department Name
John	Sales
Daniel	HR
Anne	Sales
George	HR
Tim	Sales

Joins - Two Strategies

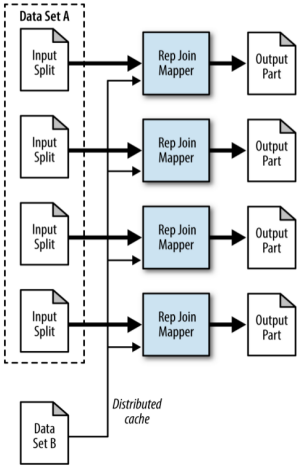
- Reduce-side join
  - Repartition join
  - When joining two or more large datasets together
- Map-side join
  - Replication join
  - When one of the datasets is small enough to cache

Joins - Reduce-Side Join



[M. Donald et al., MapReduce design patterns, O'Reilly, 2012.]

Joins - Map-Side Join



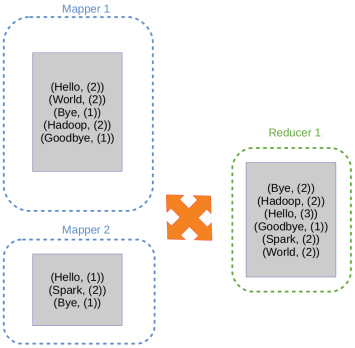
[M. Donald et al., MapReduce design patterns, O'Reilly, 2012.]

MapReduce Algorithm Design

- Local aggregation
- Joining
- Sorting

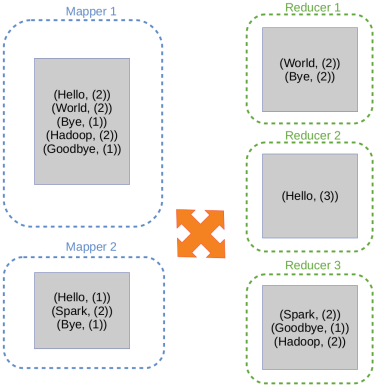
Sort (1/3)

- Assume you want to have your job output in total sort order.
- Trivial with a single Reducer.
  - Keys are passed to the Reducer in sorted order.



Sort (2/3)

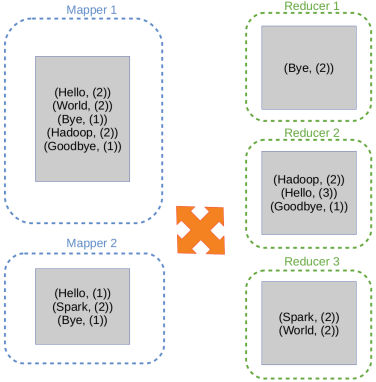
- What if we have multiple Reducer?



Sort (3/3)

- For multiple Reducers we need to choose a partitioning function

$$\texttt{key1} < \texttt{key2} \Rightarrow \texttt{partition(key1)} \leq \texttt{partition(key2)}$$



FlumeJava

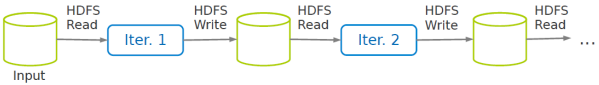
Motivation (1/2)

- It is easy in MapReduce:
  `words(doc.txt) | sort | uniq -c`

- What about this one?
  `words(doc.txt) | grep | sed | sort | awk | perl`

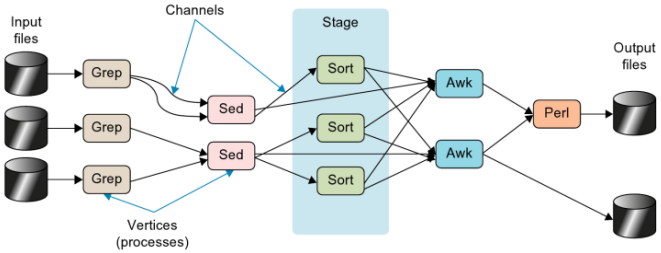
Motivation (2/2)

- Big jobs in MapReduce run in more than one Map-Reduce stages.
- Reducers of each stage write to replicated storage, e.g., HDFS.



FlumeJava

- FlumeJava is a library provided by Google to simply the creation of pipelined MapReduce tasks.



Core Idea

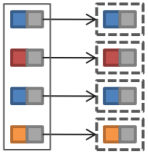
- Providing a couple of immutable parallel collections
  - `PCollection<T>` and `PTable<K, V>`
- Providing a number of parallel operations for processing the parallel collections
  - `parallelDo`, `groupByKey`, `combineValues` and `flatten`
- Building an execution plan dataflow graph
- Optimizing the execution plan, and then executing it

Parallel Collections

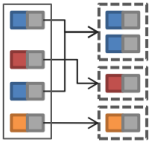
- A few classes that represent parallel collections and abstract away the details of how data is represented.
- `PCollection<T>`: an immutable bag of elements of type `T`.
- `PTable<K, V>`: an immutable multi-map with keys of type `K` and values of type `V`.
- The main way to manipulate these collections is to invoke a data-parallel operation on them.

Parallel Operations (1/2)

- `parallelDo()`: elementwise computation over an input `PCollection<T>` to produce a new output `PCollection<S>`.



- `groupByKey()`: converts a multi-map of type `PTable<K, V>` into a uni-map of type `PTable<K, Collection<V>>`.



Parallel Operations (2/2)

- `combineValues()`: takes an input `PTable<K, Collection<V>>` and an associative combining function on `V`s, and returns a `PTable<K, V>`, where each input collection of values has been combined into a single output value.

- `flatten()`: takes a list of `PCollection<T>`s and returns a single `PCollection<T>` that contains all the elements of the input `PCollections`.

Word Count in FlumeJava

```
public class WordCount {
  public static void main(String[] args) throws Exception {
    Pipeline pipeline = new MRPipeline(WordCount.class);

    PCollection<String> lines = pipeline.readTextFile(args[0]);

    PCollection<String> words = lines.parallelDo(new DoFn<String, String>() {
      public void process(String line, Emitter<String> emitter) {
        for (String word : line.split("\\s+")) {
          emitter.emit(word);
        }
      }
    }, Writables.strings());

    PTable<String, Long> counts = Aggregate.count(words);

    pipeline.writeTextFile(counts, args[1]);

    pipeline.done();
  }
}
```

Summary

Summary

- Scaling out: shared nothing architecture
- MapReduce
  - Programming model: Map and Reduce
  - Execution framework
- FlumeJava
  - Dataflow DAG
  - Parallel collection: `PCollection` and `PTable`
  - Transforms: `ParallelDo`, `GroupByKey`, `CombineValues`, `Flatten`

References

- J. Dean et al., "MapReduce: simplified data processing on large clusters", Communications of the ACM, 2008.
- C. Chambers et al., "FlumeJava: easy, efficient data-parallel pipelines", ACM Sigplan Notices, 2010.
- J. Lin et al., "Data-intensive text processing with MapReduce", Synthesis Lectures on Human Language Technologies, 2010.

Questions?