MapReduce

A programming model for processing large datasets

Part 2

Traditional HPC systems

- CPU-intensive computations
  - Relatively small amount of data
  - Tightly-coupled applications
  - Highly concurrent I/O requirements
  - Complex message passing paradigms such as MPI, PVM…
  - Developers might need to spend some time designing for failure

Challenges

- Data and storage
  - Locality, computation close to the data
- In large-scale systems, nodes fail
  - MTBF (Mean time between failures) for 1 node = 3 years
  - MTBF for 1000 nodes = 1 day
  - Solution: Built in fault-tolerance
- Commodity network = low bandwidth
- Distributed programming is hard
  - Solution: simple data-parallel programming model: users structure the application in “map” & “reduce” functions, system distributes data/work and handles faults
  - Not all applications can be parallelised: tightly-coupled computations

What requirements?

- A simple data-parallel programming model, designed for high scalability and resiliency
  - Scalability to large-scale data volumes
  - Automated fault-tolerance at application level rather than relying on high-availability hardware
  - Simplified I/O and tasks monitoring
  - All based on cost-efficient commodity machines (cheap, but unreliable), and commodity network

Core concepts

- Data spread in advance, persistent (in terms of locality), and replicated
- No inter-dependencies / shared nothing architecture
- Applications written in two pieces of code
  - And developers do not have to worry about the underlying issues in networking, jobs interdependencies, scheduling, etc…

The model

- A map function processes a key/value pair to generate a set of intermediate key/value pairs
  - Divides the problem into smaller 'intermediate key/value' pairs
- The reduce function merge all intermediate values associated with the same intermediate key
- Run-time system takes care of:
  - Partitioning the input data across nodes (blocks/chunks typically of 64Mb to 128Mb)
  - Scheduling the data and execution
  - Node failures, replication, re-submissions
  - Coordination among nodes

Map function

- A map function processes a key/value pair to generate a set of intermediate key/value pairs
  - Divides the problem into smaller 'intermediate key/value' pairs
- Map: (key1, val1) → (key2, val2)
- Ex:
  - (line-id, text) → (word, 1)
  - (2, the apple is an apple) → (the,1), (apple,1), (is,1), (an,1), (apple,1)

Reduce function

- The reduce function merge all intermediate values associated with the same intermediate key
- Reduce: (key2, [val2]) → [val3]
- Ex:
  - (word, [val_1,val_2,…]) → (word, Σval_i)
  - (the,1), (apple,1), (is,1), (an,1), (apple,1) → (an, 1), (apple,2), (is,1), (the,1)

Map/Reduce

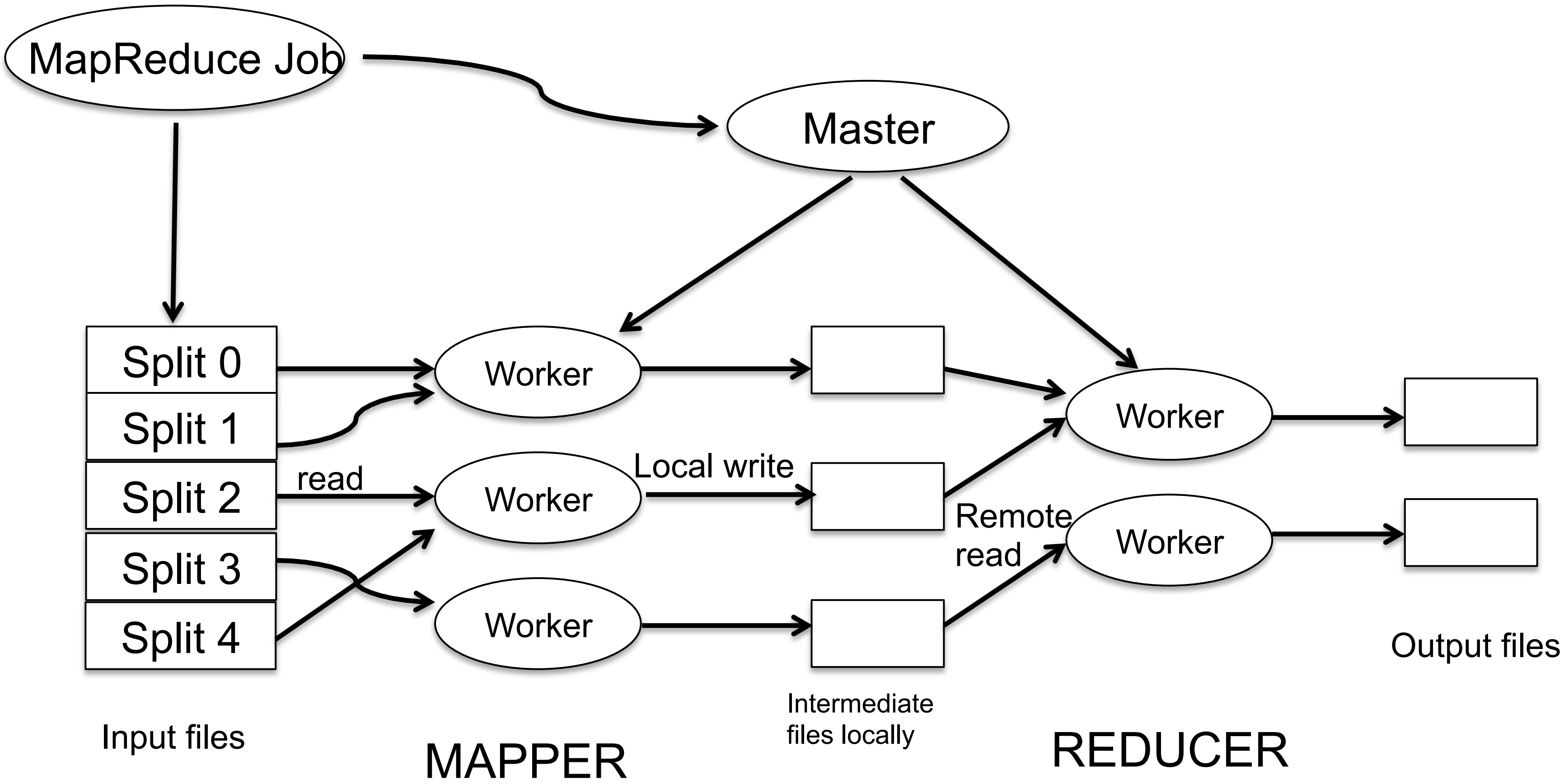
Map

- Extract something of interest from a large number of records

Reduce

- Sort intermediate results
- Aggregate intermediate results
- Generate final output

MapReduce execution



Simple Word Count

```
#key: offset, value: line
def mapper():
      for line in open("doc"):
          for word in line.split():
        output(word, 1)

#key: a word, value: iterator over counts
def reducer():
    output(key, sum(value))
```