CS-E4640 - Big Data Platforms
Lecture 3

Keijo Heljanko

Department of Computer Science
University of Helsinki & Aalto University
keijo.heljanko@helsinki.fi

26.9-2018

Spark Libraries

- Spark SQL and Dataframes - A parallel SQL database tailored for analytics (not realtime) workloads
- MLlib - A parallel machine learning library
- GraphX - A library for parallel graph processing
- Spark Streaming - A library for realtime data processing
- See: `http://spark.apache.org/`
- Commecial support from Hadoop vendors (Cloudera, HortonWorks, MapR) and from DataBricks (`http://www.databricks.com/`)

Spark RDD Transformations and Actions

- Spark uses RDD transformations and actions to process RDDs in parallel
- See: `https://spark.apache.org/docs/2.2.0/programming-guide.html` for the version 2.2.0 RDD documentation
- Please use this reference for getting familiar with Spark
- You will need RDD transformations for all home assignments

Spark SQL and Dataframes

- A parallel SQL database tailored for analytics (not realtime) workloads
- Dataframes is the underlying processing engine that can also be directly used without the SQL frontend
- Dataframe programs can be optimized by the SQL query optimizer
- See: `http://spark.apache.org/docs/2.2.0/sql-programming-guide.html` for the version 2.2.0 documentation
- You will need Spark SQL for home assignment 2

GraphX

- A parallel graph processing library on top of Spark
- See: `http://spark.apache.org/docs/2.2.0/graphx-programming-guide.html` for the version 2.2.0 documentation
- Commonly used for large social network analysis problems
- Can represent graphs as RDDs, do simple graph operations, and complex graph algorithms using the Pregel BSP (bulk synchronous parallel) computing model
- Also contains several graph algorithms: PageRank, ConnectedComponents, and TriangleCounting
- You will need GraphX for home assignment 1

MLlib

- A parallel machine learning library
- See: `http://spark.apache.org/docs/2.2.0/mllib-guide.html` for the version 2.2.0 documentation
- Widely used for large scale machine learning but not the widest collection of algorithms nor the fastest implementation of any single algorithm
- Tight integration with e.g., Spark SQL makes it a convenient choice for many machine learning use cases
- You will need MLlib for home assignment 3

Google MapReduce

- A scalable batch processing framework developed at Google for computing the Web index that predates Apache Spark
- Still in very widespread production use
- When dealing with Big Data (a substantial portion of the Internet in the case of Google!), the only viable option is to use hard disks in parallel to store and process it
- Some of the challenges for storage is coming from Web services to store and distribute pictures and videos
- We need a system that can effectively utilize hard disk parallelism and hide hard disk and other component failures from the programmer

Google MapReduce (cnt.)

- MapReduce is tailored for batch processing with hundreds to thousands of machines running in parallel, typical job runtimes are from minutes to hours
- As an added bonus we would like to get increased programmer productivity compared to each programmer developing their own tools for utilizing hard disk parallelism

Google MapReduce (cnt.)

- The MapReduce framework takes care of all issues related to parallelization, synchronization, load balancing, and fault tolerance. All these details are hidden from the programmer
- The system needs to be linearly scalable to thousands of nodes working in parallel. The only way to do this is to have a very restricted programming model where the communication between nodes happens in a carefully controlled fashion
- Apache Hadoop is an open source MapReduce implementation used by Yahoo!, Facebook, and Twitter

MapReduce and Functional Programming

- Based on the functional programming in the large:
  - User is only allowed to write side-effect free functions "**Map**" and "**Reduce**"
  - Re-execution is used for fault tolerance. If a node executing a Map or a Reduce task fails to produce a result due to hardware failure, the task will be re-executed on another node
  - Side effects in functions would make this impossible, as one could not re-create the environment in which the original task executed
  - One just needs a fault tolerant storage of task inputs
  - The functions themselves are usually written in a standard imperative programming language, usually Java

Why No Side-Effects?

- Side-effect free programs will produce the same output irregardless of the number of computing nodes used by MapReduce
- Running the code on one machine for debugging purposes produces the same results as running the same code in parallel
- It is easy to introduce side-effect to MapReduce programs as the framework does not enforce a strict programming methodology. However, the behavior of such programs is undefined by the framework, and should therefore be avoided.

Yahoo! MapReduce Tutorial

- We use a Figures from the excellent MapReduce tutorial of Yahoo! [YDN-MR], available from: `http://developer.yahoo.com/hadoop/tutorial/module4.html`
- In functional programming, two list processing concepts are used
  - Mapping a list with a function
  - Reducing a list with a function

Mapping a List

Mapping a list applies the mapping function to each list element (in parallel) and outputs the list of mapped list elements:

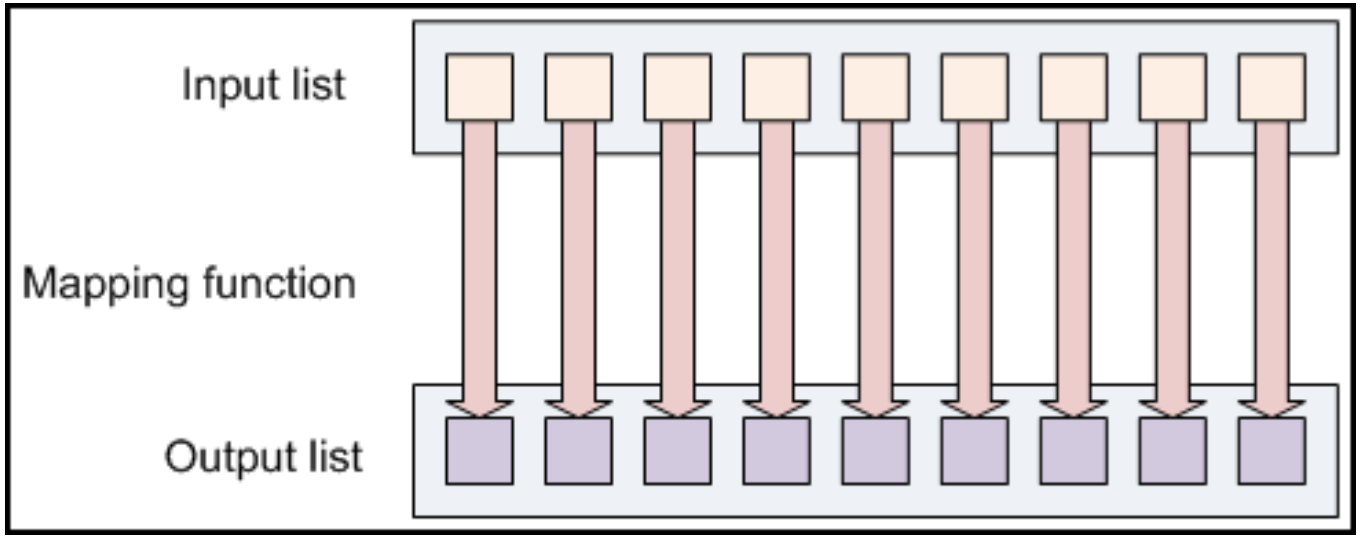


Figure: Mapping a List with a Map Function, Figure 4.1 from [YDN-MR]

Reducing a List

Reducing a list iterates over a list sequentially and produces an output created by the reduce function:

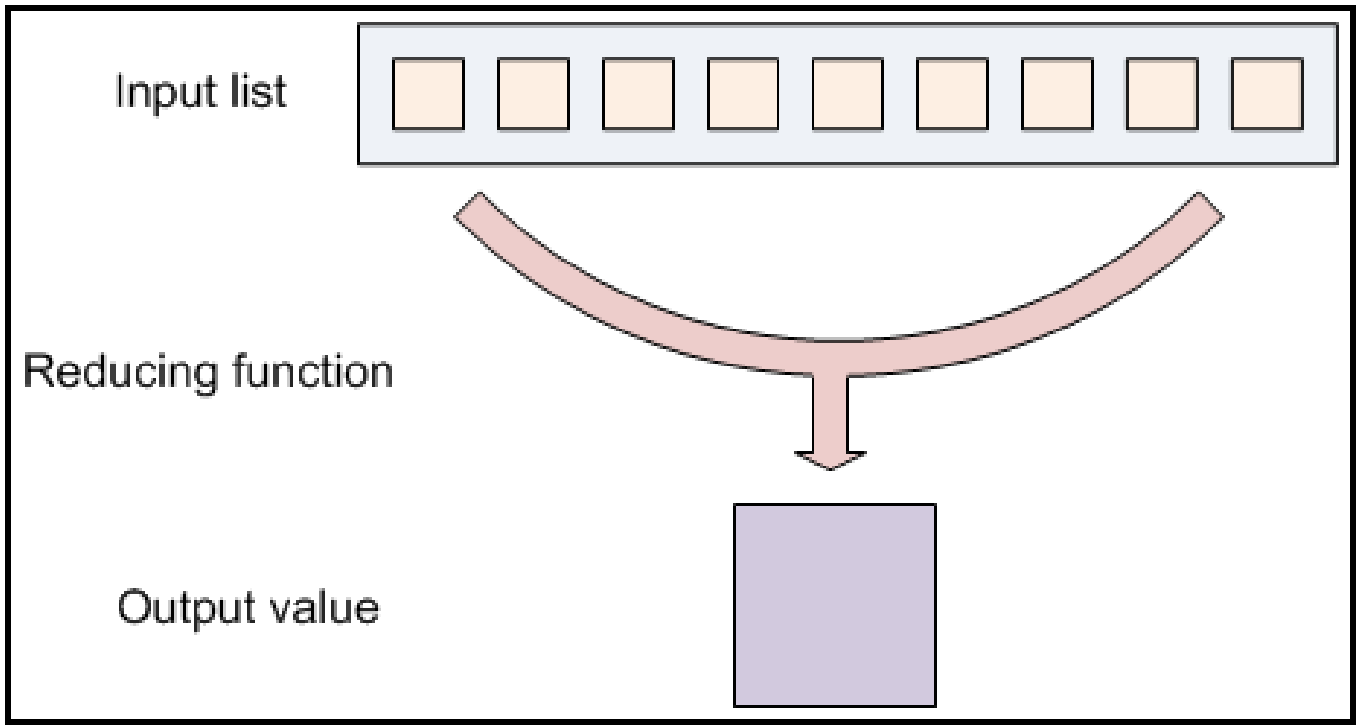


Figure: Reducing a List with a Reduce Function, Figure 4.2 from [YDN-MR]

Grouping Map Output by Key to Reducer

In MapReduce the map function outputs `(key, value)`-pairs. The MapReduce framework groups map outputs by key, and gives each reduce function instance `(key, (..., list of values, ...))` pair as input. Note: Each list of values having the same key will be independently processed:

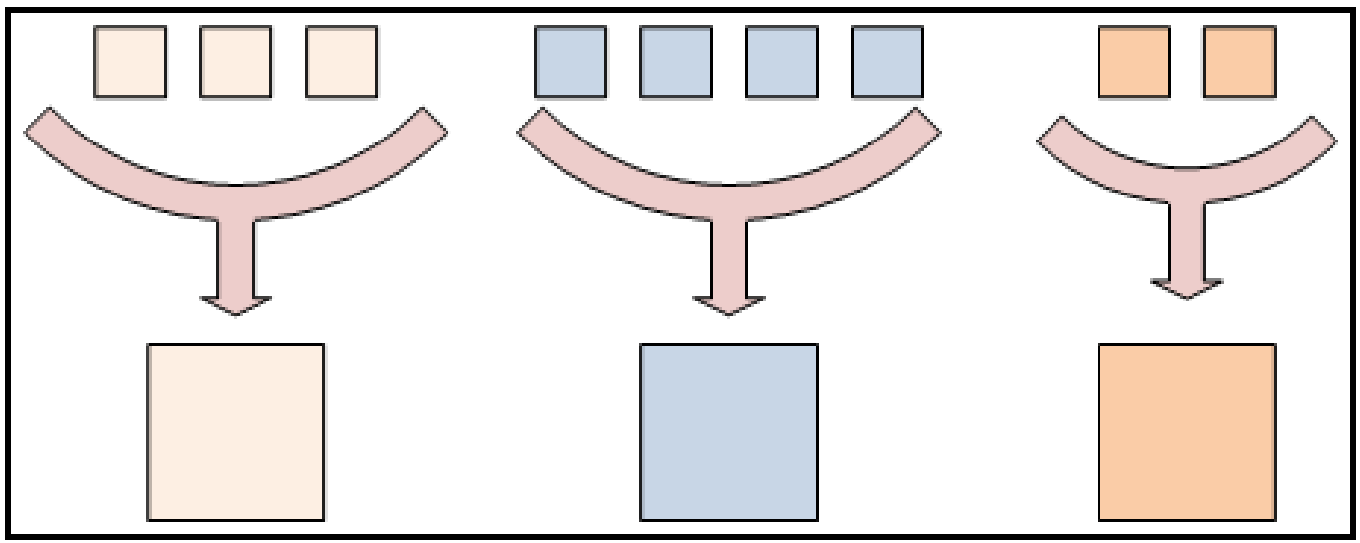


Figure: Keys Divide Map Output to Reducers, Figure 4.3 from [YDN-MR]

MapReduce Data Flow

Practical MapReduce systems split input data into large (64MB+) blocks fed to user defined map functions:

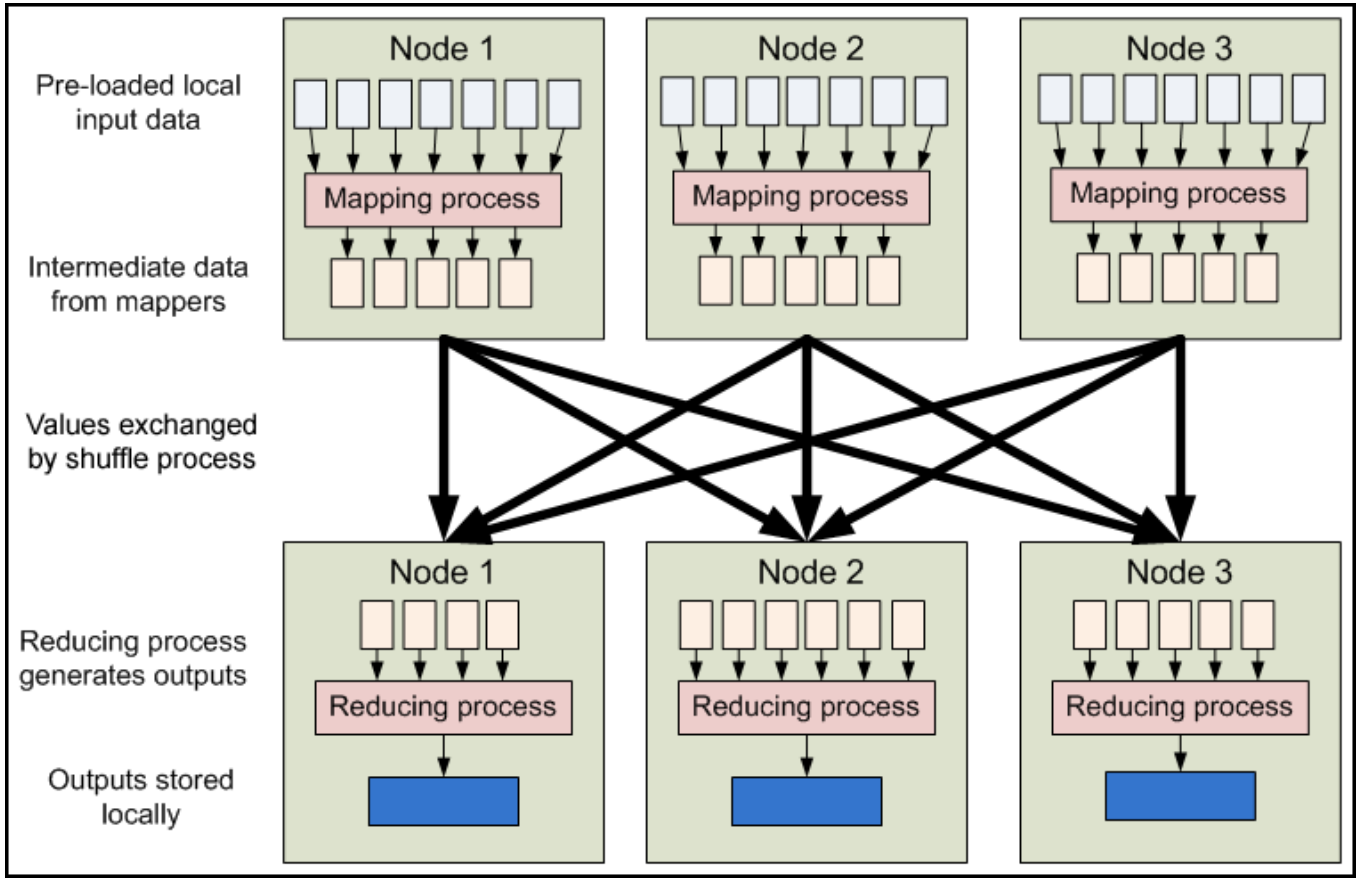


Figure: High Level MapReduce Dataflow, Figure 4.4 from [YDN-MR]

Recap: Map and Reduce Functions

- The framework only allows a user to write two functions: a "**Map**" function and a "**Reduce**" function
- The **Map**-function is fed blocks of data (block size 64-128MB), and it produces `(key, value)` -pairs
- The framework groups all values with the same key to a `(key, (..., list of values, ...))` format, and these are then fed to the **Reduce** function
- A special Master node takes care of the scheduling and fault tolerance by re-executing Mappers or Reducers

MapReduce Diagram

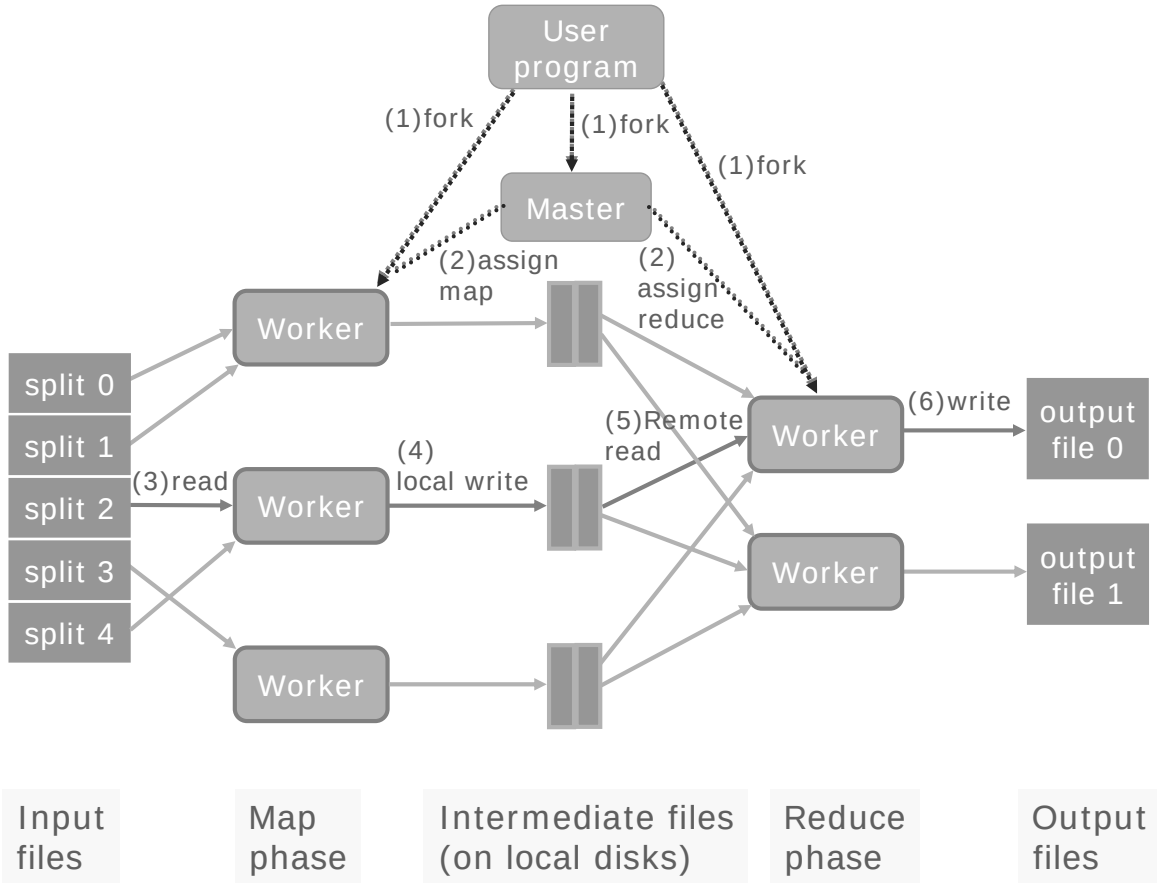


Figure: J. Dean and S. Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*, OSDI 2004

Example: Word Count

- Classic word count example from the Hadoop MapReduce tutorial:
  `http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`
- Consider doing a word count of the following file using MapReduce:

```
Hello World Bye World
Hello Hadoop Goodbye Hadoop
```

Example: Word Count (cnt.)

- Consider a Map function that reads in words one a time, and outputs `(word, 1)` for each parsed input word
- The Map function output is:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

Example: Word Count (cnt.)

- The Shuffle phase between Map and Reduce phase creates a list of values associated with each key
- The Reduce function input is:

```
(Bye, (1))
(Goodbye, (1))
(Hadoop, (1, 1))
(Hello, (1, 1))
(World, (1, 1))
```

Example: Word Count (cnt.)

- Consider a reduce function that sums the numbers in the list for each key and outputs `(word, count)` pairs. The output of the Reduce function is the output of the MapReduce job:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

Phases of MapReduce

1. A Master (In Hadoop terminology: Job Tracker) is started that coordinates the execution of a MapReduce job. Note: Master is a single point of failure
2. The master creates a predefined number of *M* Map workers, and assigns each one an input split to work on. It also later starts a predefined number of *R* reduce workers
3. Input is assigned to a free Map worker 64-128MB split at a time, and each user defined Map function is fed `(key, value)` pairs as input and also produces `(key, value)` pairs

Phases of MapReduce(cnt.)

4. Periodically the Map workers flush their `(key, value)` pairs to the local hard disks, partitioning by their `key` to R partitions (default: use hashing), one per reduce worker
5. When all the input splits have been processed, a Shuffle phase starts where $M \times R$ file transfers are used to send all of the mapper outputs to the reducer handling each key partition. After reducer receives the input files, reducer sorts (and groups) the `(key, value)` pairs by the key
6. User defined Reduce functions iterate over the `(key, (..., list of values, ...))` lists, generating output `(key, value)` pairs files, one per reducer

Google MapReduce (cnt.)

- The user just supplies the Map and Reduce functions, nothing more
- The only means of communication between nodes is through the shuffle from a mapper to a reducer
- The framework can be used to implement a distributed sorting algorithm by using a custom partitioning function
- The framework does automatic parallelization and fault tolerance by using a centralized Job tracker (Master) and a distributed filesystem that stores all data redundantly on compute nodes
- Uses functional programming paradigm to guarantee correctness of parallelization and to implement fault-tolerance by re-execution

Detailed Hadoop Data Flow

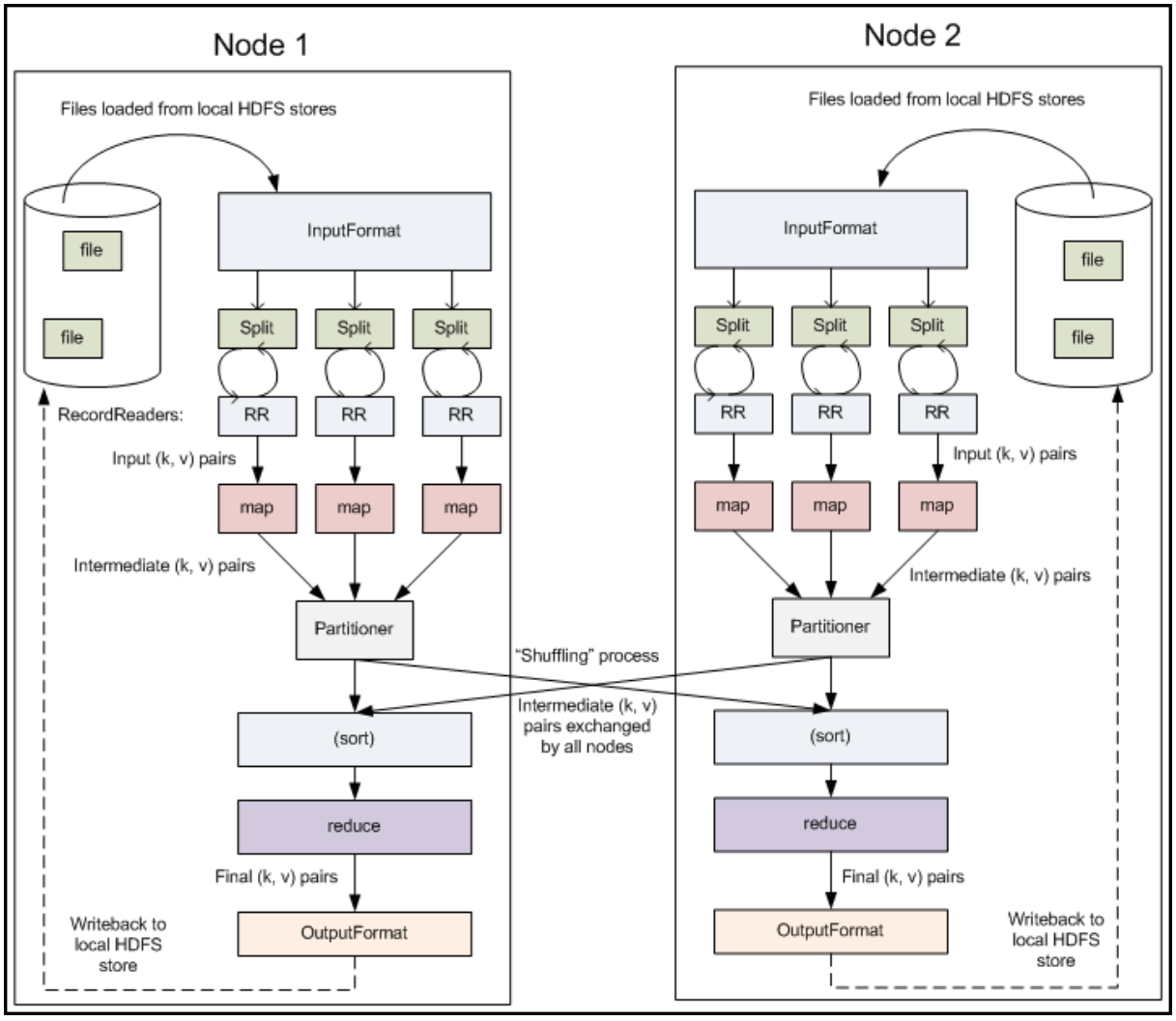


Figure: Detailed DataFlow in Hadoop, Figure 4.5 from [YDN-MR]

Adding Combiners

Combiners are Reduce functions run on the Map side. They can be used if the Reduce function is associative and commutative.

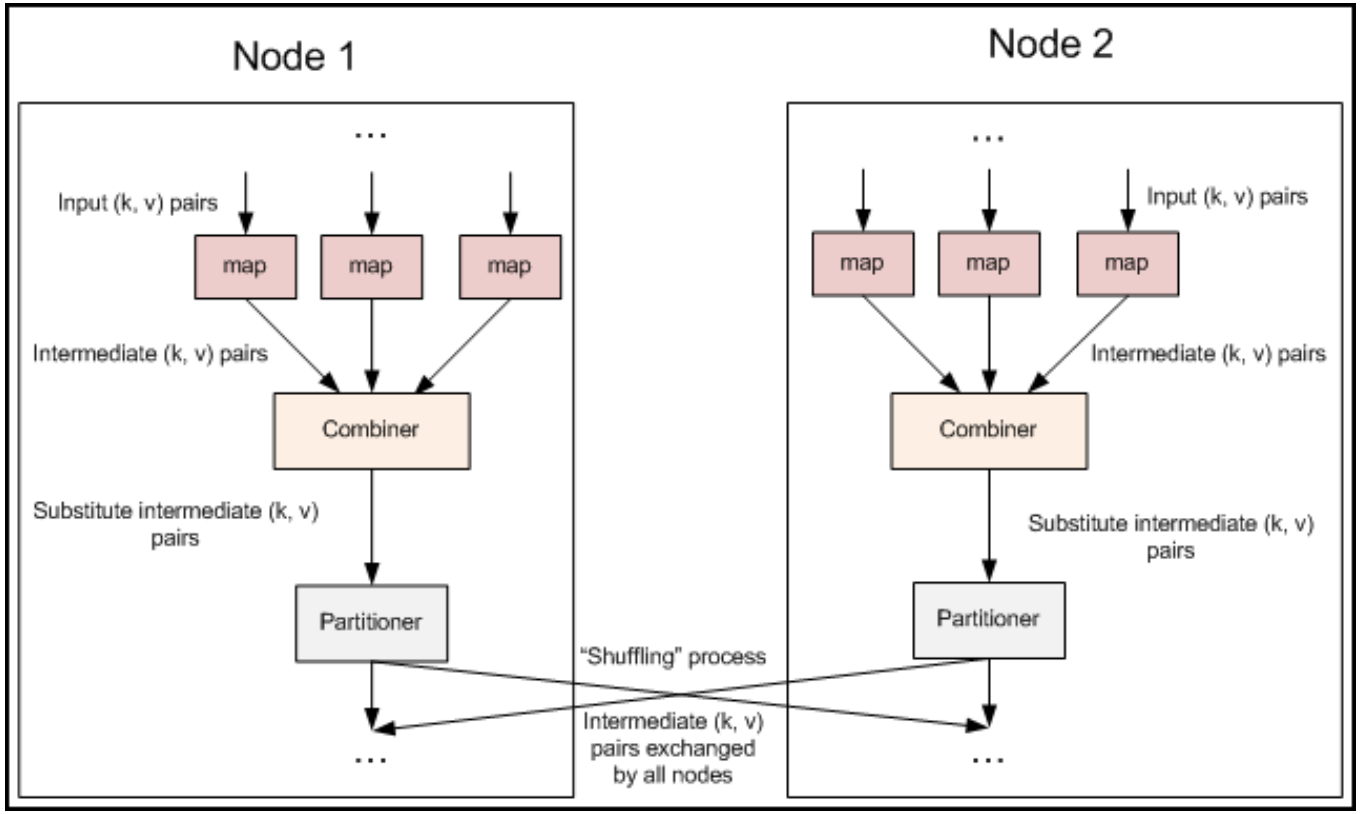


Figure: Adding a Combiner Function, Figure 4.6 from [YDN-MR]

Apache Hadoop

- An Open Source implementation of the MapReduce framework, originally developed by Doug Cutting and heavily used by e.g., Yahoo! and Facebook
- "Moving Computation is Cheaper than Moving Data" - Ship code to data, not data to code.
- Map and Reduce workers are also storage nodes for the underlying distributed filesystem: Job allocation is first tried to a node having a copy of the data, and if that fails, then to a node in the same rack (to maximize network bandwidth)
- Project Web page: `http://hadoop.apache.org/`

Apache Hadoop (cnt.)

- When deciding whether MapReduce is the correct fit for an algorithm, one has to remember the fixed data-flow pattern of MapReduce. The algorithm has to be efficiently mapped to this data-flow pattern in order to efficiently use the underlying computing hardware
- Builds reliable systems out of unreliable commodity hardware by replicating most components (exceptions: Master/Job Tracker and NameNode in Hadoop Distributed File System)

Apache Hadoop (cnt.)

- Tuned for large (gigabytes of data) files
- Designed for very large 1 PB+ data sets
- Designed for streaming data accesses in batch processing, designed for high bandwidth instead of low latency
- For scalability: NOT a POSIX filesystem
- Written in Java, runs as a set of user-space daemons

Hadoop Distributed Filesystem (HDFS)

- A distributed replicated filesystem: All data is replicated by default on three different Data Nodes
- Inspired by the Google Filesystem
- Each node is usually a Linux compute node with a small number of hard disks (4-16)
- A NameNode that maintains the file locations, many DataNodes (1000+)

Hadoop Distributed Filesystem (cnt.)

- Any piece of data is available if at least one datanode replica is up and running
- Rack optimized: by default one replica written locally, second in the same rack, and a third replica in another rack (to combat against rack failures, e.g., rack switch or rack power feed)
- Uses large block size, 128 MB is a common default - designed for batch processing
- For scalability: Write once, read many filesystem

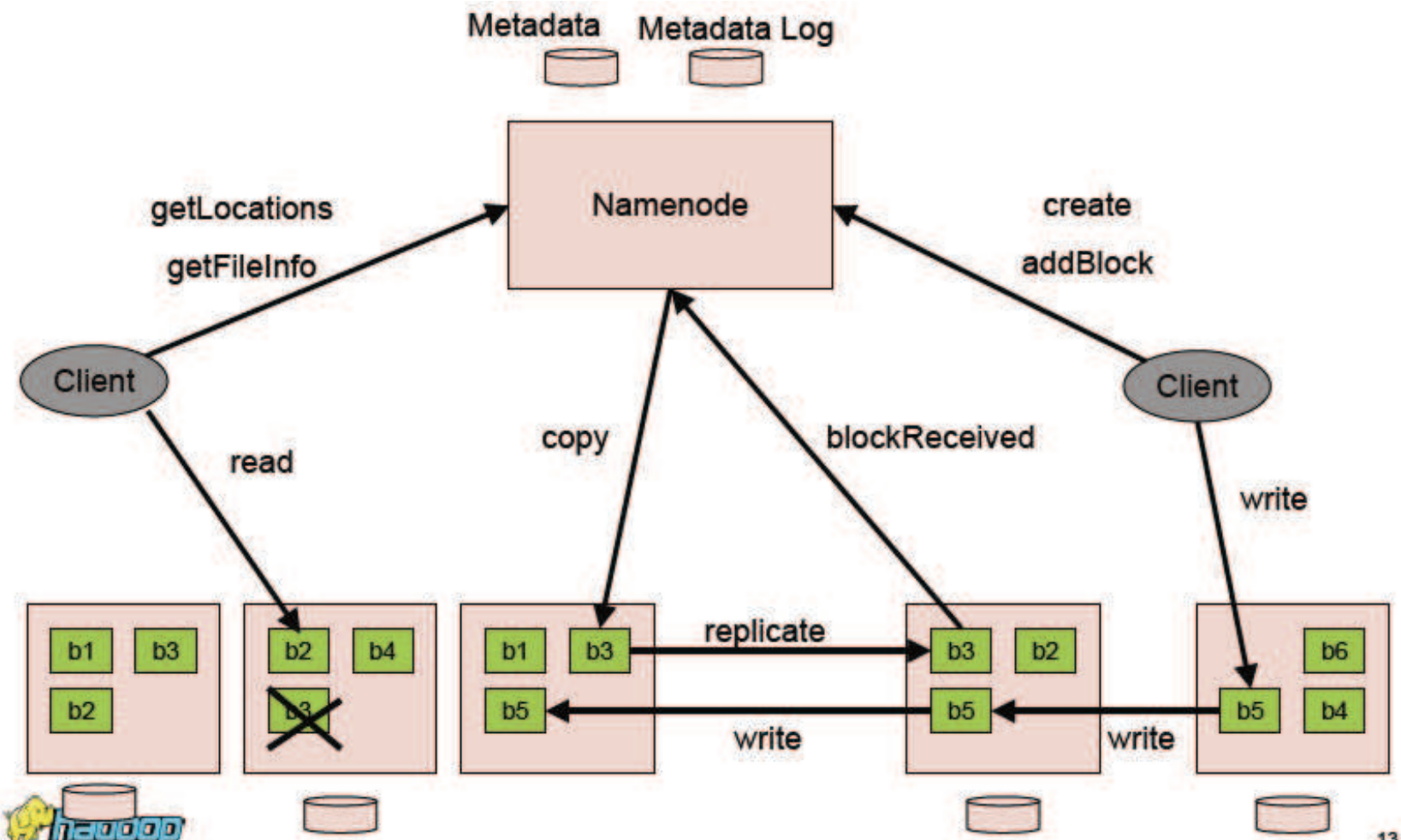
Implications of Write Once

- All applications need to be re-engineered to only do sequential writes. Example systems working on top of HDFS:
  - Apache Spark with all its libraries (MLlib, Spark SQL, GraphX, Spark Streaming)
  - HBase (Hadoop Database), a database system with only sequential writes, Google Bigtable clone
  - MapReduce batch processing system
  - Apache Pig and Hive data mining tools
  - Mahout machine learning libraries
  - Lucene and Solr full text search
  - Nutch web crawling

HDFS Architecture

- From: HDFS Under The Hood by Sanjay Radia of Yahoo

http://assets.en.oreilly.com/1/event/12/HDFS%20Under%20the%20Hood%20Presentation%201.pdf



HDFS Architecture

- NameNode is a single computer that maintains the namespace (meta-data) of the filesystem. Implementation detail: Keeps all meta-data in memory, writes logs, and does periodic snapshots to the disk
- Recent new feature: Namespace can be split between multiple NameNodes in HDFS Federation: `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html`
- All data accesses are done directly to the DataNodes
- Replica writes are done in a daisy chained (pipelined) fashion to maximize network utilization

HDFS Scalability Limits

- 20PB+ deployed HDFS installations (10 000+ hard disks)
- 4000+ DataNodes
- Single NameNode scalability limits: The HDFS is NameNode scalability limited for write only workloads to around HDFS 10 000 clients, K. V. Shvachko: HDFS scalability: the limits to growth: `http://www.usenix.org/publications/login/2010-04/openpdfs/shvachko.pdf`
- The HDFS Federation feature was implemented to address scalability (not fault tolerance) issues

HDFS Design Goals

Some of the original HDFS targets have been reached by 2009, some need additional work:

- Capacity: 10 PB (Deployed: 14 PB in 2009, 20+ PB by 2010)
- Nodes: 10000 (Deployed: 4000)
- Clients: 100000 (Deployed: 15000)
- Files: 100000000 (Deployed: 60000000)

Hadoop Hardware

- Reasonable CPU speed, reasonable RAM amounts for each node
- 4-12 hard disks per node seem to be the current suggestion
- CPU speeds are growing faster than hard disk speeds, so newer installations are moving to more hard disks / node
- 10 Gigabit Ethernet networking seems to be dominant

Hadoop Network Bandwidth Consideration

- Hadoop is fairly network latency insensitive
- Mapper reads can often be read from the local disk or the same rack (intra-rack network bandwidth is cheaper than inter-rack bandwidth)
- For jobs with only small Map output, very little network bandwidth is used
- For jobs with large Map output, Hadoop likes large inter-node network bandwidth (Shuffle phase),
- To save network bandwidth, Mappers should produce a minimum amount of output