Big Data Platforms (5 ECTS)
DATA14003/AYDATA14003en
Lecture 3

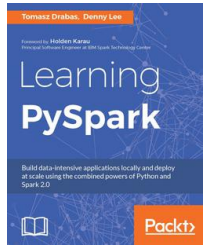
Keijo Heljanko

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

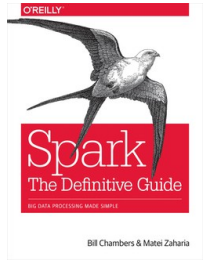
14.9-2021

Spark Books

- A good beginner book for learning PySpark is:
  - Tomasz Drabasand and Denny Lee: “Learning PySpark”, released in 2017. Publisher: Packt Publishing ISBN: 9781786463708

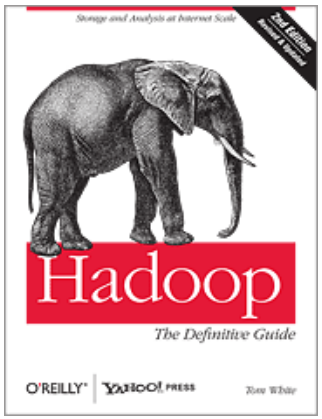


- A more comprehensive Spark reference is:
  - Bill Chambers and Matei Zaharia: “Spark: The Definitive Guide”, released in 2018. Publisher: O’Reilly Media, Inc. ISBN: 9781491912218



Hadoop Books

- I can warmly recommend the Book:
  - Tom White: “Hadoop: The Definitive Guide, Fourth Edition”, O’Reilly Media, 2015. ISBN: 9781491901687.
    `http://www.hadoopbook.com/`



- An alternative book is:
  - Chuck Lam: “Hadoop in Action”, Manning, 2010. Print ISBN: 9781935182191.

Spark Libraries

- Spark SQL and Dataframes - A parallel SQL database tailored for analytics (not realtime) workloads
- MLlib - A parallel machine learning library
- GraphFrames - A library for parallel graph processing
- Structured Streaming - Stream data processing
- Spark Documentation:

  `https://spark.apache.org/docs/latest/`
- Commercial support from vendors such as DataBricks (`http://www.databricks.com/`) and Cloudera

Spark RDD Transformations and Actions

- Spark uses RDD transformations and actions to process RDDs in parallel
- See: `https://spark.apache.org/docs/latest/rdd-programming-guide.html` for the latest version RDD documentation
- Please use this reference for getting familiar with Spark, you can toggle the language to Python to see PySpark example codes

GraphFrames

- A parallel graph processing library on top of Spark, not part of Spark core
- See: `https://docs.databricks.com/spark/latest/graph-analysis/graphframes/index.html` for the latest version documentation
- Commonly used for large social network analysis problems
- Can represent graphs as GraphFrame, do simple graph operations, and complex graph algorithms using the Pregel BSP (bulk synchronous parallel) computing model
- Also contains several useful graph algorithms such as TringleCount and PageRank
- GraphX is an earlier graph library for Scala based on RDDs

MLlib

- A parallel machine learning library
- See: `https://spark.apache.org/docs/latest/ml-guide.html` for the latest version 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

Spark SQL and Dataframes

- A parallel SQL database implementation tailored for analytics (not realtime) workloads
- Dataframes is the underlying data structure and 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/latest/sql-programming-guide.html` for the latest version documentation
- "SQL is the most widely used parallel functional programming language"

Data Frames

Data Frames

- The Spark project has introduced a new data storage and processing framework - Data Frames - to eventually replace RDDs for many applications
- Problem with RDDs: As RDDs are operated by arbitrary user functions in Scala/Java/Python, optimizing them is very hard
- DataFrames allow only a limited number of DataFrame native operations, and allows the Spark SQL optimizer to rewrite user DataFrame code to equivalent but faster codes
- Instead of executing what the user wrote (as with RDDs), execute an optimized sequence of operations
- Allows DataFrame operations to be also implemented in a compiled fashion

Broadcast Variables

Broadcast Variables

- Broadcast variables are a way to send some read-only data to all PySpark workers in a coordinated fashion:

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
<pyspark.broadcast.Broadcast object at 0x102789f10>

>>> broadcastVar.value
[1, 2, 3]
```

Accumulators

Accumulators

- Accumulators allow a way to compute statistics done during operations:

```
>>> accum = sc.accumulator(0)
>>> accum
Accumulator<id=0, value=0>

>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s

>>> accum.value
10
```

- Note: If a transformation is rescheduled during operation execution, the accumulators are increased several times
- Thus the accumulators can also count processing that is “wasted” due to fault tolerance / speculation
- Use accumulators only to count statistics on processed items, not to do real computations!

Buggy Code misusing Closures

Buggy Code misusing Closures

- When Spark is run in truly distributed mode, any changes made to worker local variables will be lost when the worker exists
- The only way to communicate to other threads from workers is through data produced into RDDs and DataFrames

RDD Persistence levels

RDDs can be configured with different persistence levels for caching RDDs:

These levels can be used to `persist()` RDD or DataFrame for further use. `unpersist()` frees the memory for other uses.

- `MEMORY_ONLY`
- `MEMORY_AND_DISK`
- `MEMORY_ONLY_SER` (Java and Scala)
- `MEMORY_AND_DISK_SER` (Java and Scala)
- `DISK_ONLY`
- `MEMORY_ONLY_2, MEMORY_AND_DISK_ONLY_2`
- `OFF_HEAP` (experimental)

Advanced MapReduce: Recap of Phases

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 centralised 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

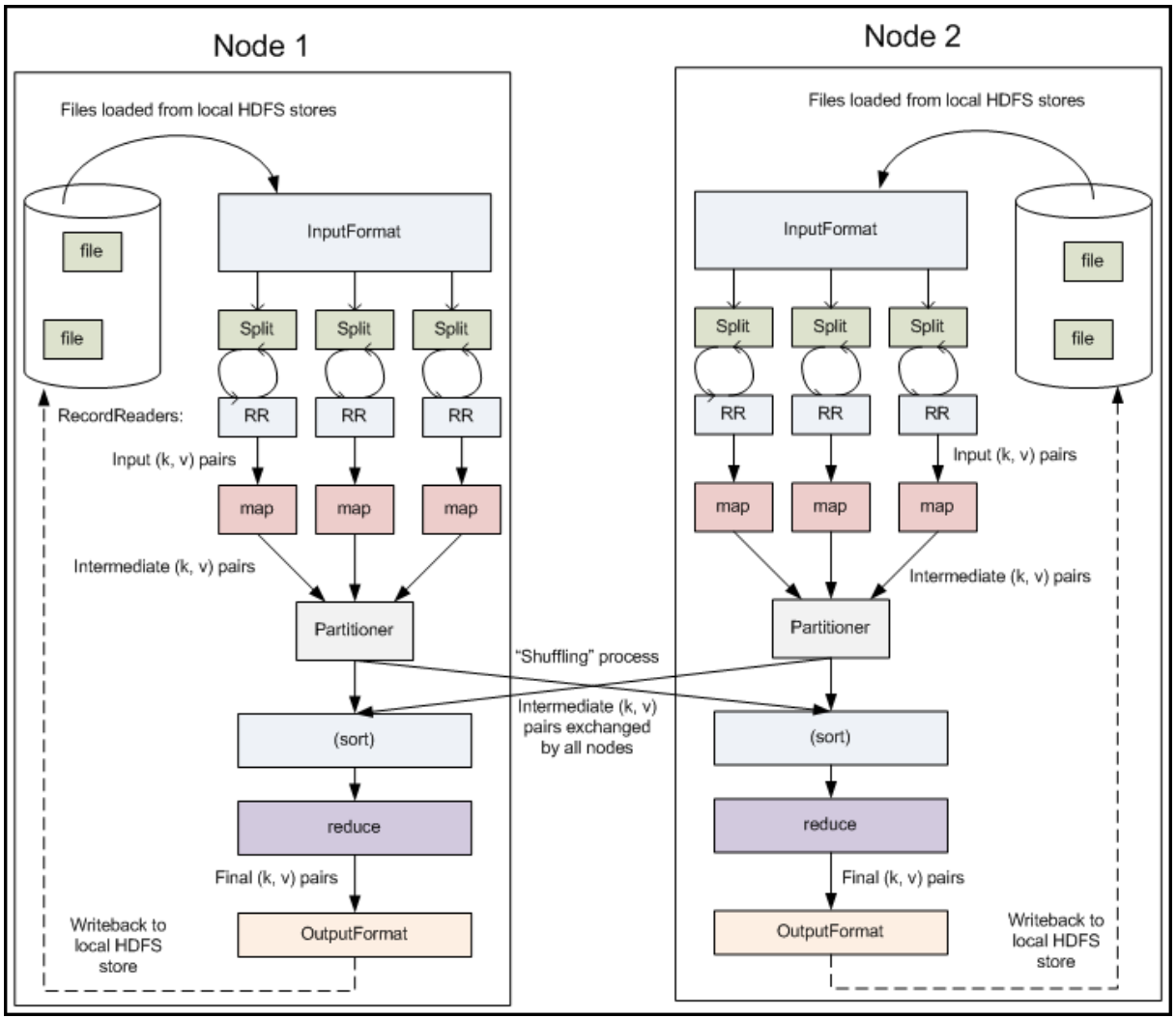


Figure: Detailed DataFlow in Hadoop, Figure 4.5 from [YDN-MR]

Adding Combiners

Combiners are Reduce functions run on the Map side. They can be used if the Reduce function is associative and commutative. Note: Spark requires this, and thus always runs "combiners".

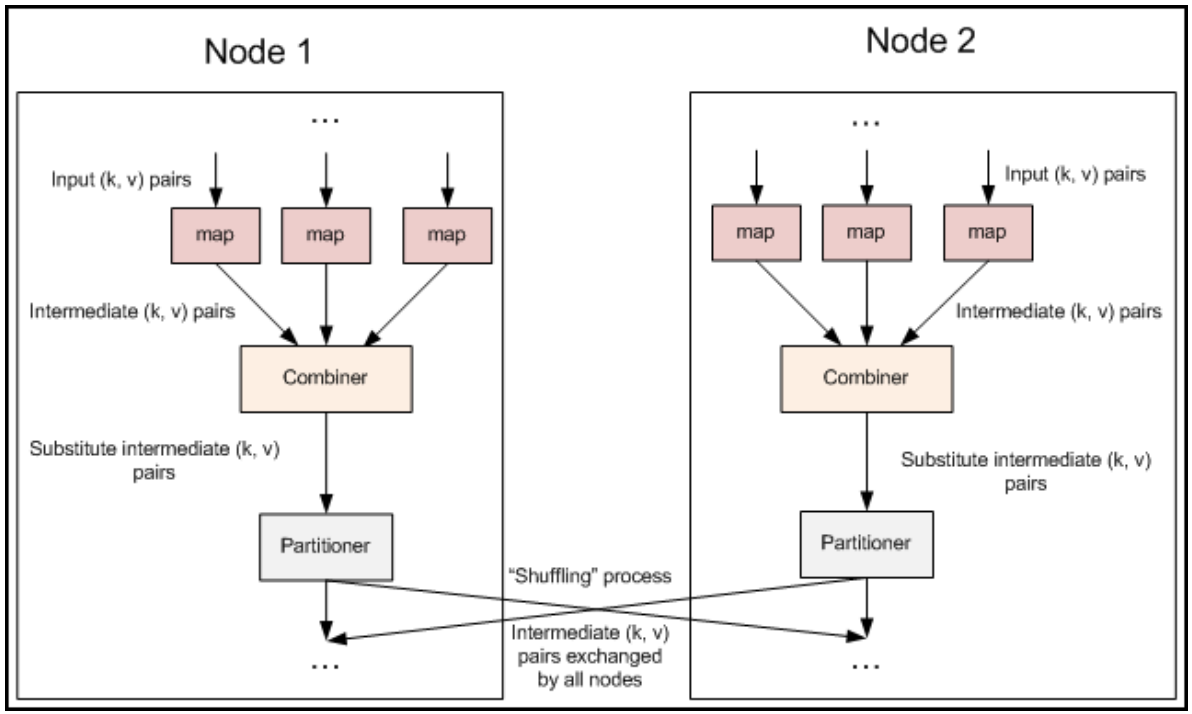


Figure: Adding a Combiner Function, Figure 4.6 from [YDN-MR]

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/`
- The Hadoop Distributed Filesystem (HDFS) is still in very wide use

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 (8-24)
- A NameNode that maintains the file locations, many DataNodes (1000+)
- HDFS is still often the filesystem of choice for Spark users

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

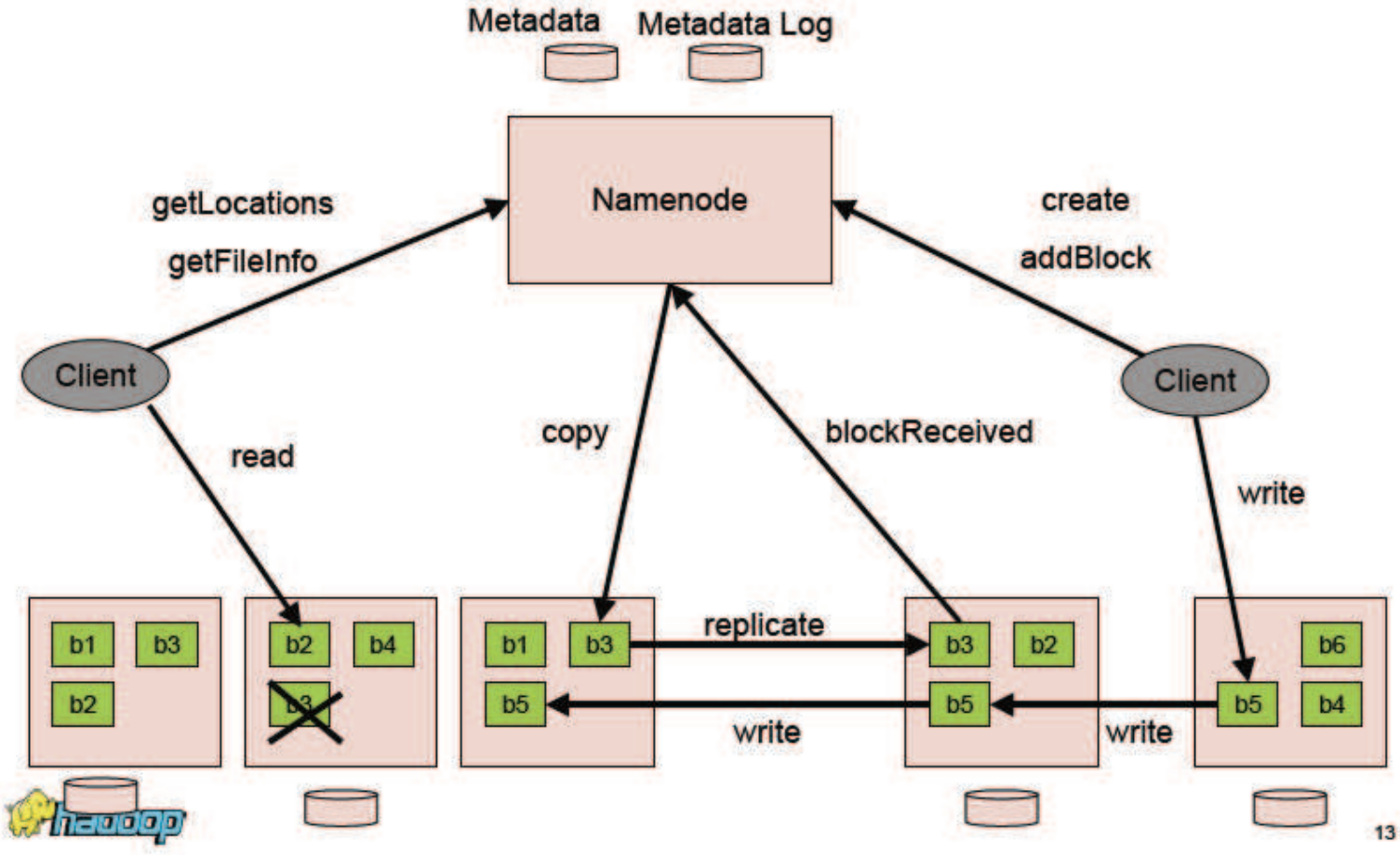
Implications of Write Once

- 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, GraphFrames, Structured 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%



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`

  and ViewFS:

  `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ViewFs.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) in 2009, 100+ PB in 2018
- 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 and ViewFs features have been implemented to address scalability (not fault tolerance) issues
- Other newer performance features include reading from replica namenodes (HDFS-12975)

HDFS Design Goals

Some of the original HDFS targets have been reached already by 2009:

- Capacity: 10 PB (Deployed: 14 PB in 2009, 100+ PB at Uber in 2018)
- Nodes: 10000 (Deployed: 4000)
- Clients: 100000 (Deployed: 15000)
- Files: 100000000 (Deployed: 60000000)

Spark and Hadoop Hardware

- Reasonable CPU speed, reasonable RAM amounts for each node
- 8-24 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
- Spark is able to use RAM for caching, typically 32GB memory per node minimum but also upto 256 GB RAM configurations are common

Network Bandwidth Consideration

- Spark and Hadoop are fairly network latency insensitive
- Mapper reads can often be read from the local disk or the same rack with HDFS (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, Spark and Hadoop need large inter-node network bandwidth (Shuffle phase),
- To save network bandwidth, a minimum amount of shuffle data should be generated