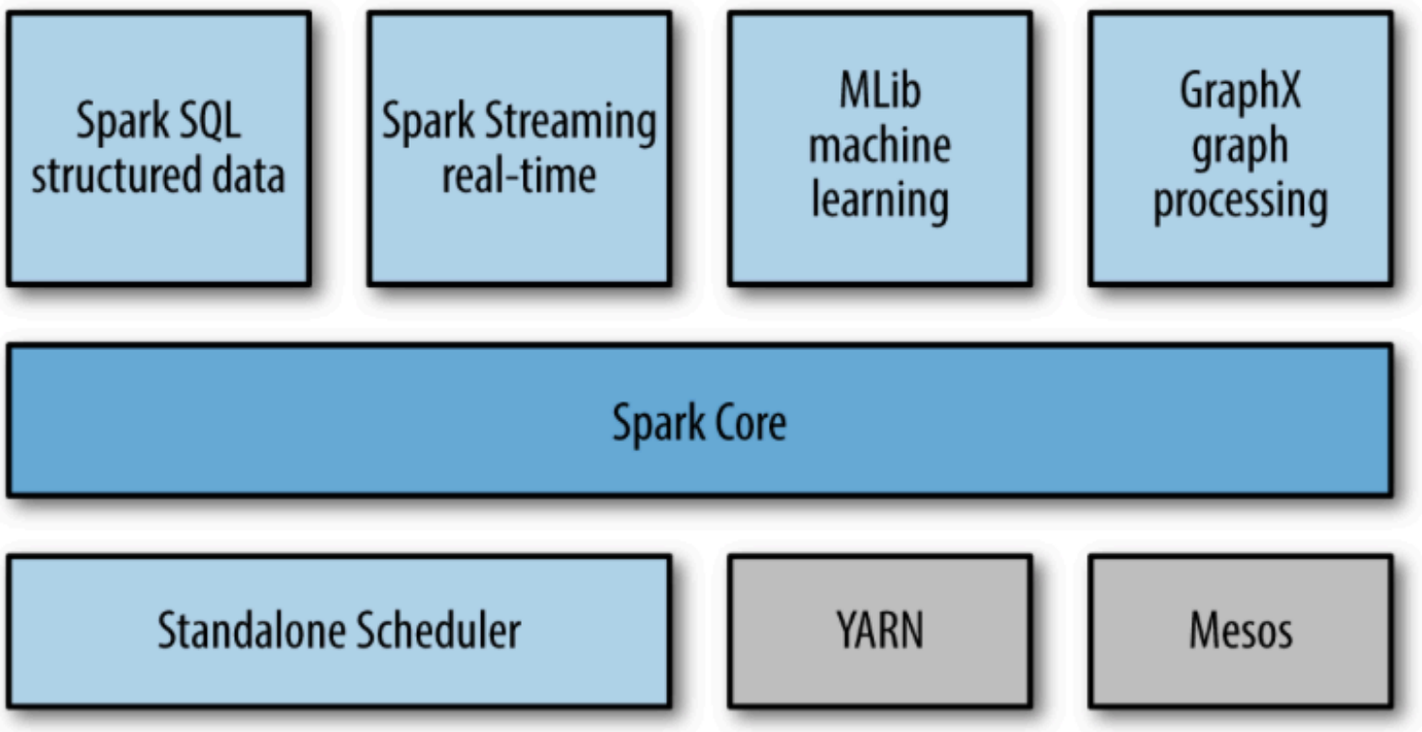
Performance issues with Spark

Yves Denneulin

Some experiments…

- Run of a ML application on Spark (Alternated Least Square)
  - data size
  - number of cores
  - with and without persistance of RDD

Spark components

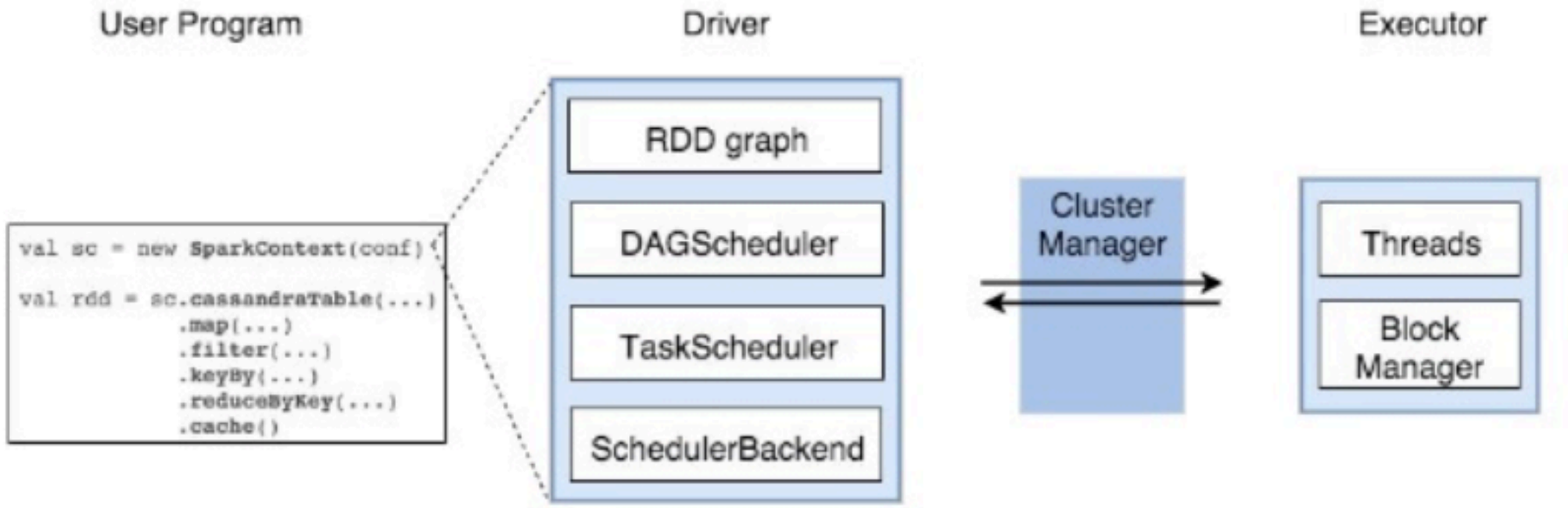


Spark core

- task planning
- memory and error management
- RDD API and management

In-memory data management

Spark execution workflow

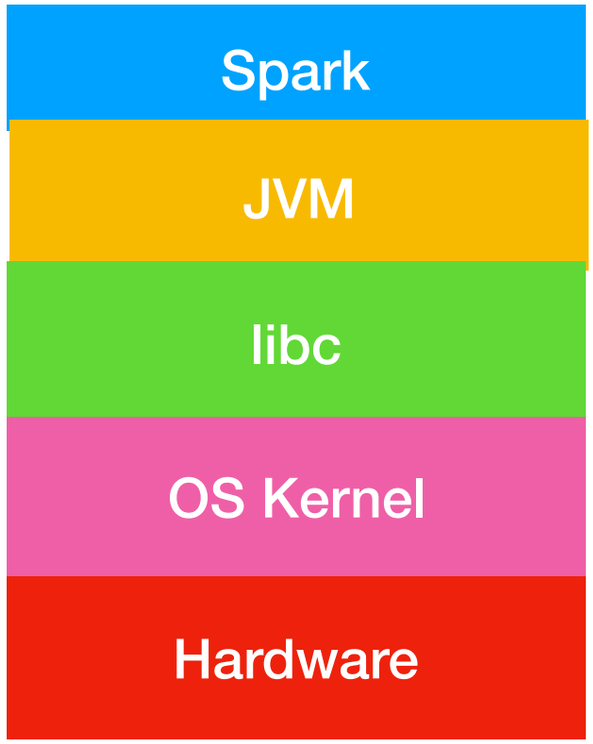


Spark core

- task planning
- memory and error management
- RDD API and management

From https://www.slideshare.net/akirillov/spark-workshop-internals-architecture-and-coding

Below Spark Core



- Written in Java
  - dedicated memory management

Key elements for performances
  quality of code
  memory management (Spark, JVM, OS Kernel)
  data movements (code and Spark dependent)

The Storage Latency Hierarchy

Technology	Latency	Size (e.g.)
L1 CPU Cache	4 cycles (~1 nsec)	32K
L2 CPU Cache	10 cycles (3 nsec)	256K
LLC CPU Cache	40 cycles (13 nsec)	1 MB
DRAM	240 cycles (80 nsec)	16 GB
NVRAM	1200 cycles (400 nsec)	128 GB
RDMA Read	6K cycles (2 usec)	16 GB
FLASH Read	150K cycles (50 usec)	128 GB
FLASH Write	1500K cycles (500 usec)	128 GB
HDD Write min	1500K cycles (500 usec)*	4 TB
HDD Read min	15000K cycles (5 msec)	4 TB
HDD Read max	75000K cycles (25 msec)	4 TB
Tape File Access	150000000K cycles (50 sec)	6 TB

2017-8

* Write to track cache

Operations on RDDs

- 2 categories in Spark
  - narrow : no data movement necessary (`filter, select,...`)~`map`
    - can be combined
  - Wide : data movement necessary between nodes (`groupBy, orderBy`, ...)~`reduce`

Writing efficient code

- Conception time
  - think in terms of narrow and wide operations
    - narrow is good, wide is bad
    - `cache` RDDs that will be re-used
- Running time: two aspects to take into account
  - the movement of data between the nodes
  - the local execution and its memory
- Spark is built for in-memory data management

Understanding how Spark works

- Operation(s) on a RDD done in 3 phases

1. create the logical plan of execution: a DAG of transformations between parent data and the resulting RDD

2. Actions on the RDD: creation of a physical plan from the logical one done by the Spark Catalyst

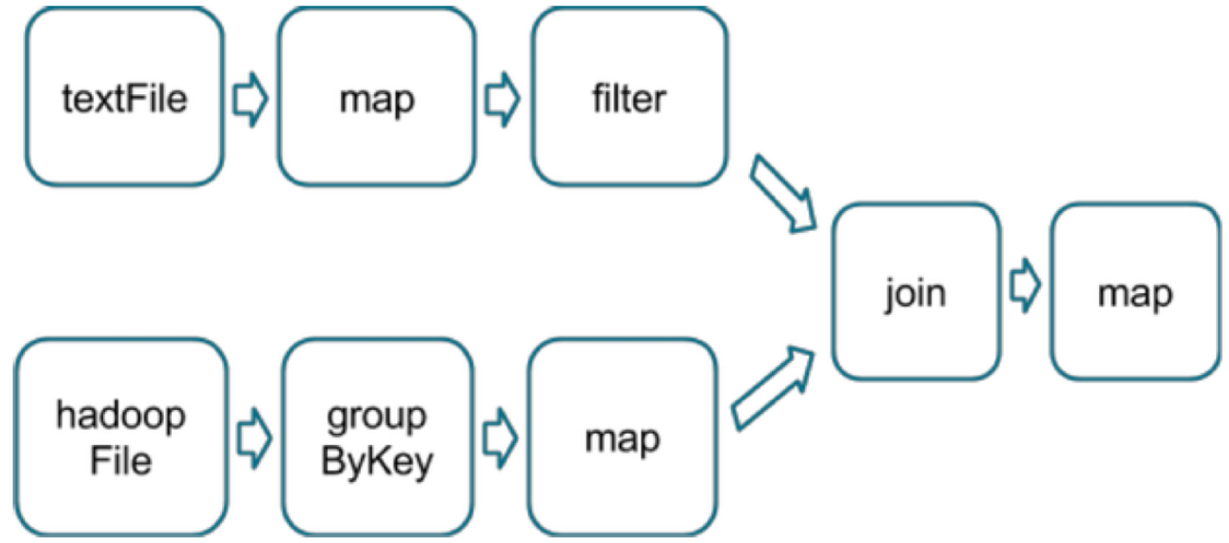
3. Execute the tasks on the {node, cluster}

Understanding how Spark works (2)

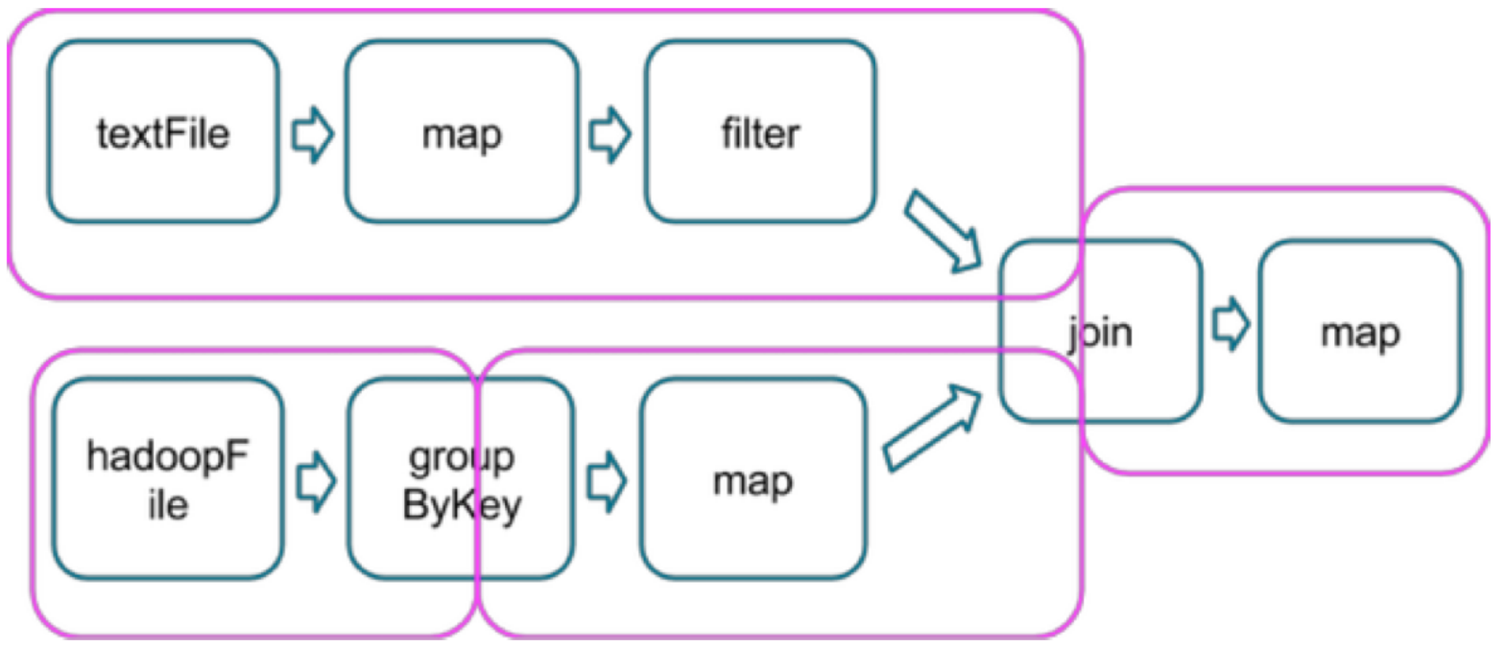
- spark action done by a job
  - job = {transformation}
- a transformation is split into stages
  - stage = {task}
- a task = same code on a subset of data
  - no shuffling

Understanding how Spark works (3)

- exemple of a graph



- divided in stages



Understanding how Spark works (4)

- between each stage
  - data written to disk by parents and fetch from disks by children (or sent through the network)
    - must be serialised (expensive!)
- If the number of data partitions is different between parents and children then reshuffle of all the data
  - `numPartitions` parameter can be used to avoid that

General tips

- limit the number of wide operations
  - avoid `groupbykey` prefer `reducebykey`

- Tips at `https://github.com/AllenFang/spark-overflow`

Looking at how a code is executed

- use the `explain(true)` method to view it
  - or the `toDebugString` method (for RDD)
- Alternatively
  - the web frontend `localhost:4040`

Tuning the execution

- Using various parameters
  - you have to understand how an execution is done
- Two main criteria use to schedule tasks: CPU and I/O
  - `–executor-cores` and `–executor-memory` flags (`–driver-memory`)
  - memory impacts the amount of data that can be cached and the maximum size of shuffle data
- With dynamic allocation Spark requests new nodes when too many tasks are waiting
- too much memory use can lead to excessive garbage collection time

Number of tasks

- The number of tasks sets the level of parallelism
  - essential for efficient execution on an architecture
- number of tasks in a stage=number of partitions in the last RDD in the stage
  - changed by `coalesce`, `union` and `cartesian` operations
- on a cluster: the number of HDFS nodes storing the data
- can be found in the `RDD.partitions.size` variable
- can be fixed for every wide operations using `numPartitions`

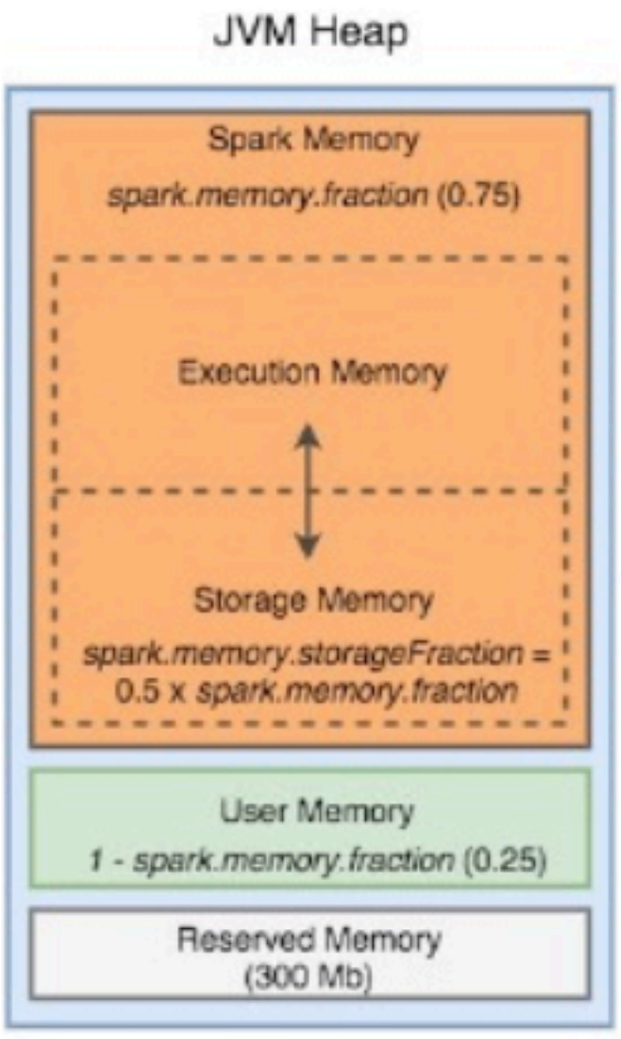
Key efficiency factors

- number of
  - workers
  - executors
  - => right level of parallelism
- buffer size
  - `shuffle.file.buffers`

Data storage

- Done by Spark in 2 formats
  - deserialized data objects (memory)
  - serialized binary for writing to disks and transferring on network
    - the Kryo library is more size efficient (but more expensive)

Memory management



- Execution Memory
  - storage for data needed during tasks execution
  - shuffle-related data
- Storage Memory
  - storage of cached RDDs and broadcast variables
  - possible to borrow from execution memory (spill otherwise)
  - safeguard value is 0.5 of Spark Memory when cached blocks are immune to eviction
- User Memory
  - user data structures and internal metadata in Spark
  - safeguarding against OOM
- Reserved memory
  - memory needed for running executor itself and not strictly related to Spark

From https://www.slideshare.net/akirillov/spark-workshop-internals-architecture-and-coding

Memory management

- Spark divides memory in 2 categories
  - execution memory: used to do computations
  - storage memory: used for caching
- size of each depends on the use
  - threshold for maximum size of storage (default: 60%)
- can be read on the localhost:4040 website

Local memory management

- Garbage collection can be expensive
  - large amount of physical memory, lots of objects
- Java objects are memory consuming (up to 4-5x)
  - their management takes place on the heap
- Solution: Tungsten

From http://spark.apache.org/docs/latest/tuning.html

Tips for memory optimisation

- use array instead of sets or hashmap
- avoid nested structures
- use numeric IDs or enum instead of String

From http://spark.apache.org/docs/latest/tuning.html

Tips for efficient storage

- Use an efficient backend
  - avro
  - parquet
  - etc.
- Handling and treatment will be much more efficient

From http://spark.apache.org/docs/latest/tuning.html

Levers for locality

- Move data or move code?
- Spark policy is to favour data locality (i.e. move code)
  - it waits for a time for a CPU to be free then move data
    - this time can be parametrised (`spark.locality`)

Conclusion

- Code is important for performances
  - narrow=good, wide=bad
- but so are finding the best parameters value!
  - partitioning, memory allocation, persistence
  - can be tedious

Paths for improvements

- Use DataSet instead of RDD
  - structuration helps performance
- don’t be afraid to use broadcast variables when necessary