Apache Spark under the hood

Big Data and Cloud Computing (CC4093)

Eduardo R. B. Marques, DCC/FCUP

Introduction

Apache Spark

Apache Spark has the stated goal of providing a "unified platform" for big data applications (see "Apache Spark:A Unified Engine for Big Data Processing" by M. Zaharia et al.).

- General graph execution model that is able to perform in-memory-processing and optimize data-flow, based on Resilient Distributed Datasets (RDDs), and higher-level DataFrame/SQL APIs.
- Supports batch processing but also stream processing. Specialised Spark libraries exist for machine learning or graph analytics.
- Language bindings for Scala, Java, Python and R.
- Flexible deployment (standalone, YARN/HDFS, ...) and interoperable with heterogeneous data-sources (e.g., Google Cloud Storage, SQL databases, ...) and formats (e.g., CSV, JSON, multiple binary formats).

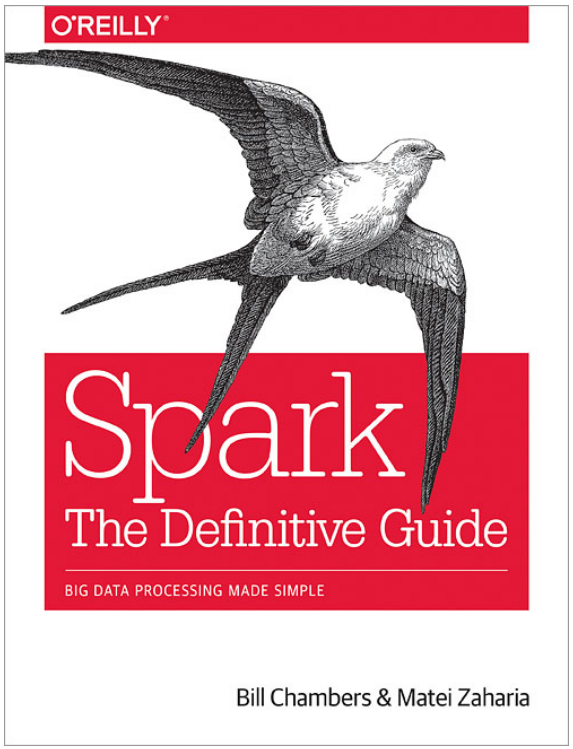
Spark applications

A benefit of using Spark (or MapReduce) is that programmers do not need to deal with aspects such as:

- how parallel execution and network communication operate during the execution of an application
- the allocation of computing resources necessary to run applications

Spark handles these aspects automatically, as you have experienced in practice. Let us now uncover the details of Spark in terms of application architecture and execution.

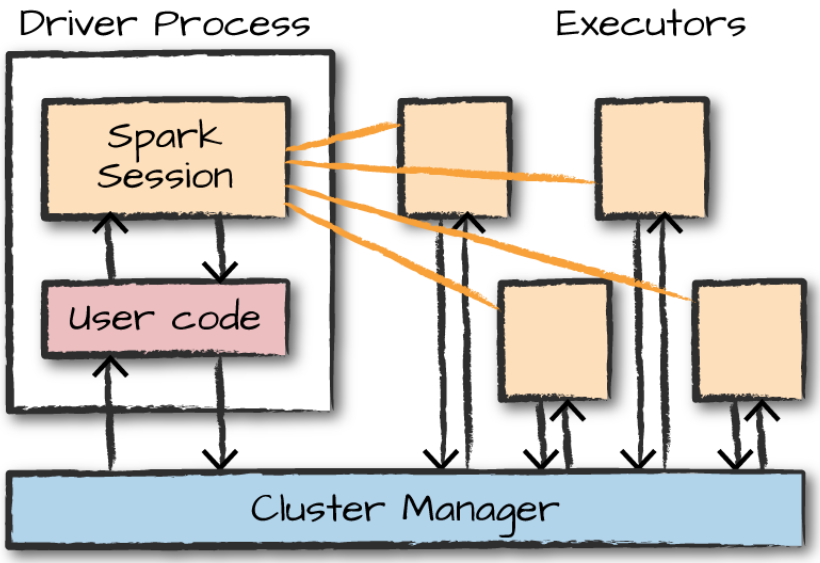
Image credits



Some images in these slides are taken from Bill Chambers and Matei Zaharia's book in compliance with O'Reilly's Safari learning platform membership agreement.

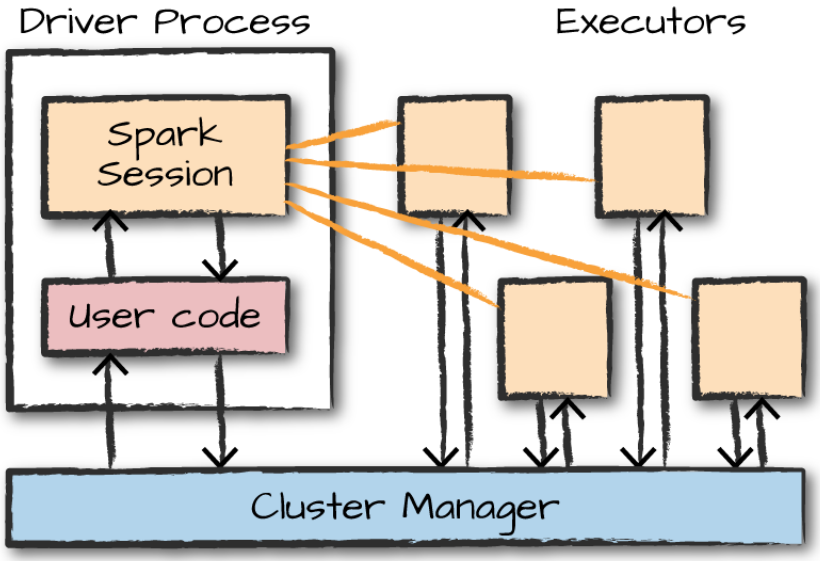
Architecture

Spark architecture



A Spark **application** is composed of a **driver process** (also called driver program) and a set of **executor processes**. The **driver** is responsible for maintaining application state, running user code and handling user input scheduling and distributing work to executors. **Executors** execute code assigned by the driver and report the state and final results of computation back to the driver.

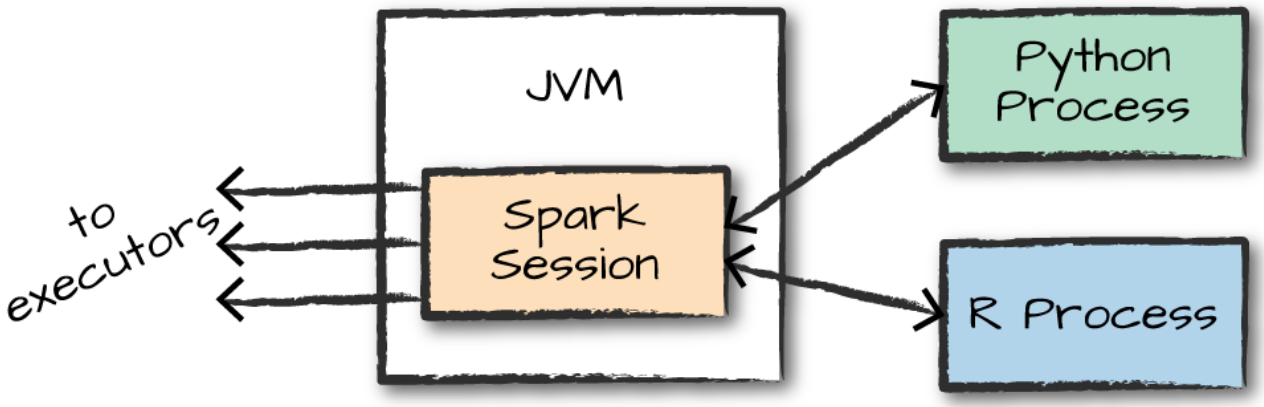
The Spark session



As part of a driver process, an application must first create a **Spark session**. The spark session provides user code with the primary interface for Spark functionality.

When you run your program using an interactive Spark shell (spark-shell, pyspark, sparkR) or custom notebook environments, the Spark session is conveniently created at startup and made accessible through the `spark` (and the `sc` "Spark context" variable). In Google Colab notebooks we must initialise these explicitly as we have seen.

PySpark and SparkR



Spark is primarily written in **Scala**. Scala is interoperable with **Java**, in any case you can write Spark applications using separate APIs for each language. The driver program in this case will be hosted by a **Java Virtual Machine (JVM)**, i.e., the execution engine for Java compiled bytecode.

Spark also has APIs for **Python (PySpark)** and **R (SparkR)** In this case the Python or R Spark program will spawn a JVM that hosts the actual Spark session. This happens transparently to the Python/R program(mer).

Spark session creation

For non-interactive program execution (outside "Spark shells") the Spark session must be created explicitly by user code.

Here's an example in Python for Spark in "stand-alone" mode (using the local machine):

```
if __name__ == "__main__" :
    from pyspark import SparkContext
    from pyspark.sql import SparkSession
    spark = SparkSession\
        .builder\
        .appName("My beautiful app")\
        .master("local[*]")\
        .getOrCreate()
    sc = spark.sparkContext
    sc.setLogLevel("WARN")
```

Spark execution

Execution of a Spark application

Let us present the core concepts for the execution of a Spark application.

- During execution, the **driver process** launches **jobs. A job is defined whenever the application triggers an action (e.g. `RDD.collect()`).**
- **Job = sequence of stages** $\text{Stage}_1 \rightarrow ... \rightarrow \text{Stage}_n$. Stages for the same job do not run in parallel.
- **Stage = groups of tasks that may execute together to compute the same operation on multiple RDD partitions / machines.** Stages are separated by **shuffle operations** that repartition data.
- **Task = computation that runs in a single executor operating transformations over blocks of data. Several instances of the same tasks may run in parallel (using distinct executors).**

Logical plans and physical plans

For each job (application action), Spark assembles:

- **Logical execution plans** structured in terms of (RDD, data frame, ...) transformations, that are independent of the cluster's characteristics.
- **Physical execution plans**, compiled from logical execution plans define the actual job stages and their component tasks. and may account for the cluster characteristics.

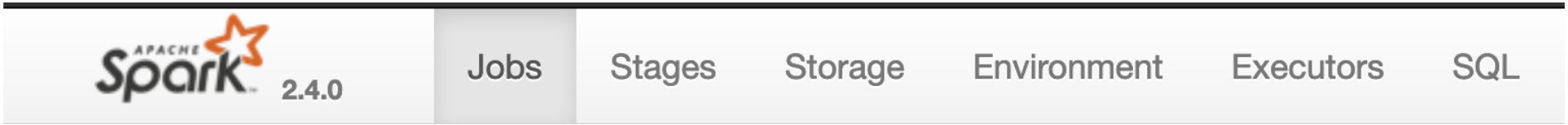
Logical and physical execution plans take form as directed acyclic graphs (DAGs), where nodes define tasks or reshuffle operations and edges reflect execution precedence.

Example 1

Let us consider a variant of the word count computation:

```
## "Word count" computation.
rdd = sc.textFile(input_file)\
        .flatMap(lambda line: [(word,1) \
                 for word in line.split()])\
        .reduceByKey(lambda x,y: x + y)
## Action that triggers the execution of job
results = rdd.collect()
```

Spark UI for an application



For every running application we can check its Spark UI, where we can get detailed information regarding:

- **Jobs**: jobs executed by the application;
- **Stages**: we can inspect stage details including RDD-level DAGs for component tasks;
- **Storage**: storage associated to the application;
- **Environment**: information on environment configuration;
- **Executors**: executors associated to the application;
- **SQL**: high-level execution plans/DAGs that are defined for Spark SQL (data frame operations).

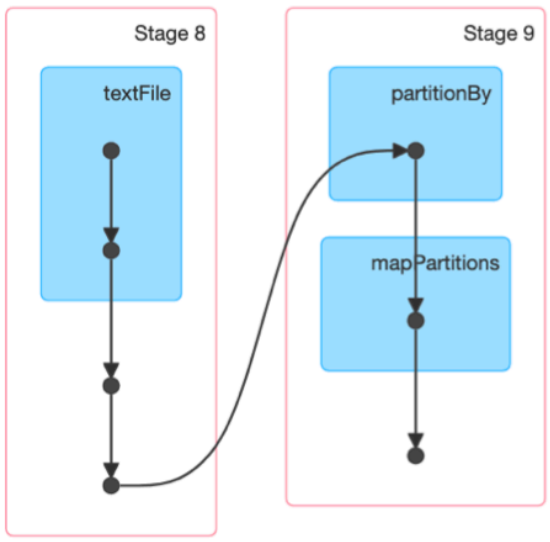
Example 1 in the Spark UI

The call to `collect()` triggers the execution of a new job, involving two stages and 8 tasks, as shown in the Spark UI:

Details for Job 6

Status: SUCCEEDED
Completed Stages: 2

▸ Event Timeline
▾ DAG Visualization

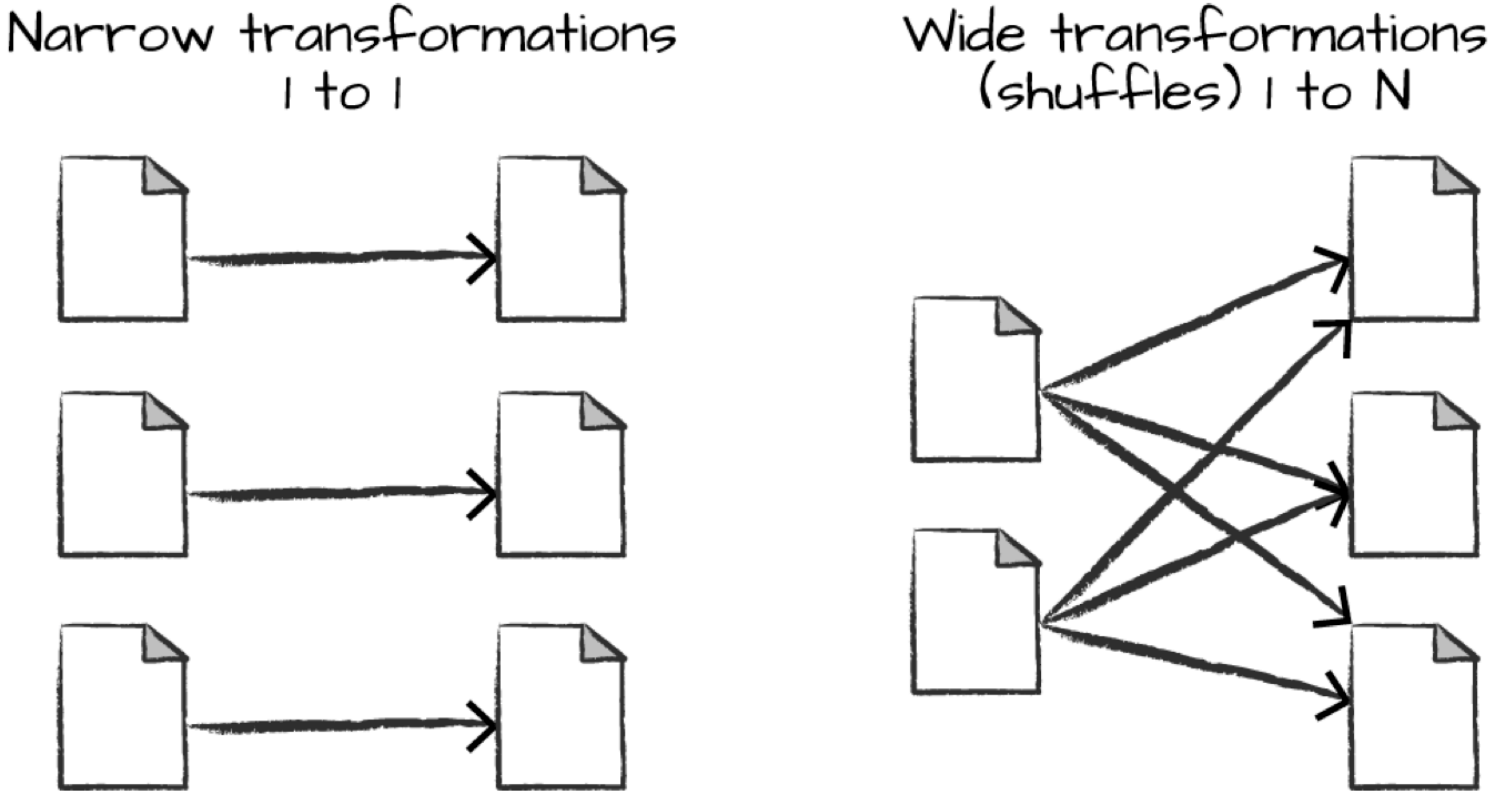


▾ **Completed Stages (2)**

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
9	_run_module_as_main at /Library/Frameworks /Python.framework /Versions/3.5/lib/python3.5 /runpy.py:170 +details	2019/03/05 22:15:59	0.2 s	4/4			2.1 MB	
8	_run_module_as_main at /Library/Frameworks /Python.framework /Versions/3.5/lib/python3.5 /runpy.py:170 +details	2019/03/05 22:15:57	2 s	4/4	6.5 MB			2.1 MB

In the DAG, we see that the there is a data reshuffle at the end of stage 1, before stage 2 can start. In this case, each stage has 4 tasks corresponding to 4 RDD partitions.

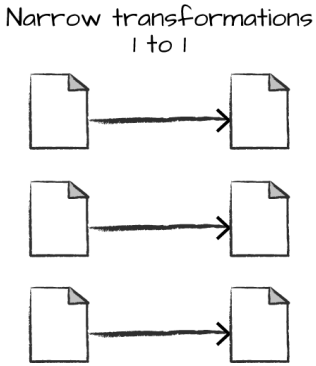
Narrow vs. wide transformations



Narrow transformations like `flatMap` (or `map`, `filter`) map one input partition to one output partition.

Wide transformations, like `reduceByKey`, also called **shuffles**, may read from several input partitions and contribute to many output partitions. They required data to be **reshuffled** across executors.

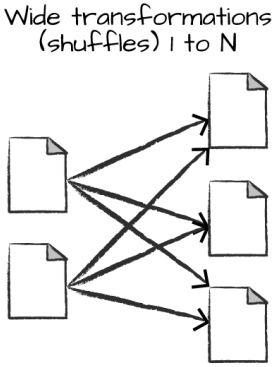
Narrow transformations and stage pipelining



We could add other narrow transformations in sequence to `flatMap` that would **fit in the same stage**.

In the same stage, Spark will pipeline the transformations, allowing for **in-memory data processing with no intermediate disk writes** (up to the amount of available memory per executor).

Wide transformations and reshuffle persistence



Wide transformations require reshuffling in the network, hence two wide transformations cannot be part of the same stage and be pipelined. Spark will however try to optimize performance through **shuffle persistence**:

- The **source** stage of a shuffle operation writes **shuffle files** to local disks (at the level of each executor).
- In the **sink** stage, that performs the grouping and reduction, data is fetched from these shuffle files in groups of keys.

The point is to avoid recomputation of the the source stage if there is a failure during the sink stage, and to be able to schedule the sink stage flexibly.

Example 2

We now consider a data frame (Spark SQL execution) The following is a simple (self-explanatory) example over a MovieLens data set:

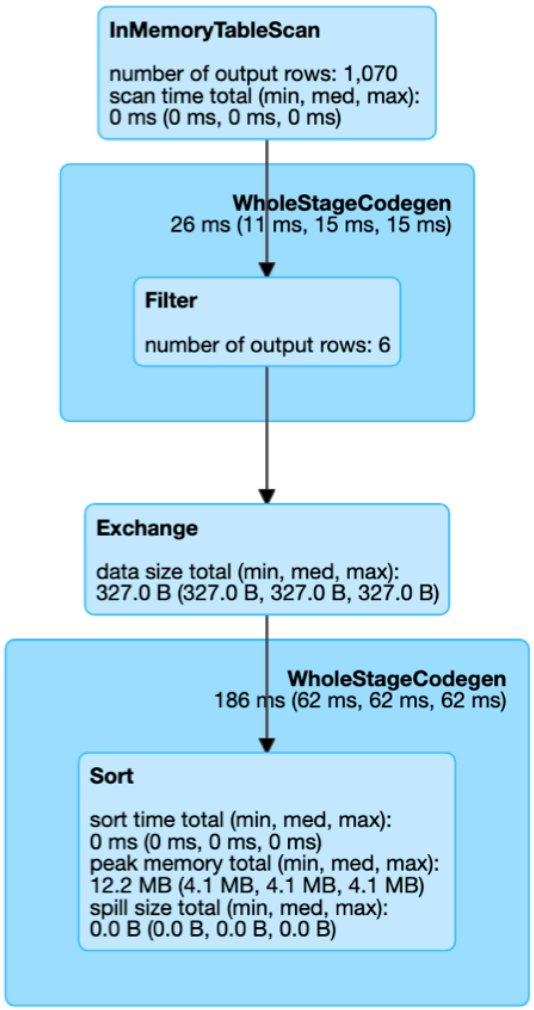
```
movies = ... ## read file
## Transformations form a Spark SQL query
StarWarMovies = movies\
  .filter(movies.title.contains('Star Wars'))\
  .orderBy(movies.title)
## Action
results = StarWarMovies.collect()
```

Spark SQL queries are executed as set of jobs that form a graph.

Example 2 - Spark UI

Details for Query 3

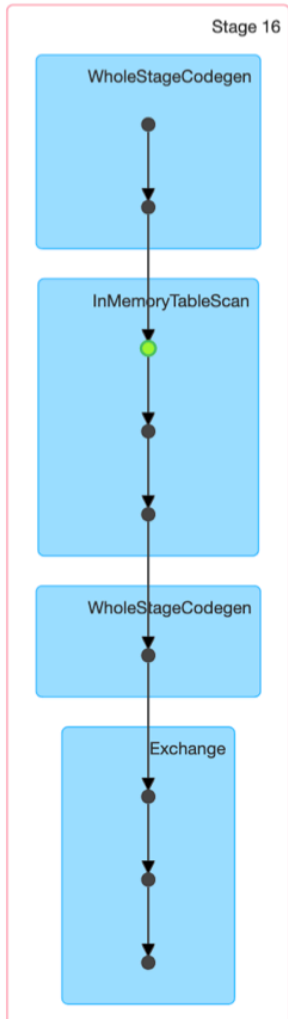
Submitted Time: 2019/03/05 23:30:22
Duration: 0.7 s
Succeeded Jobs: 13 14



Details for Job 13

Status: SUCCEEDED
Completed Stages: 1

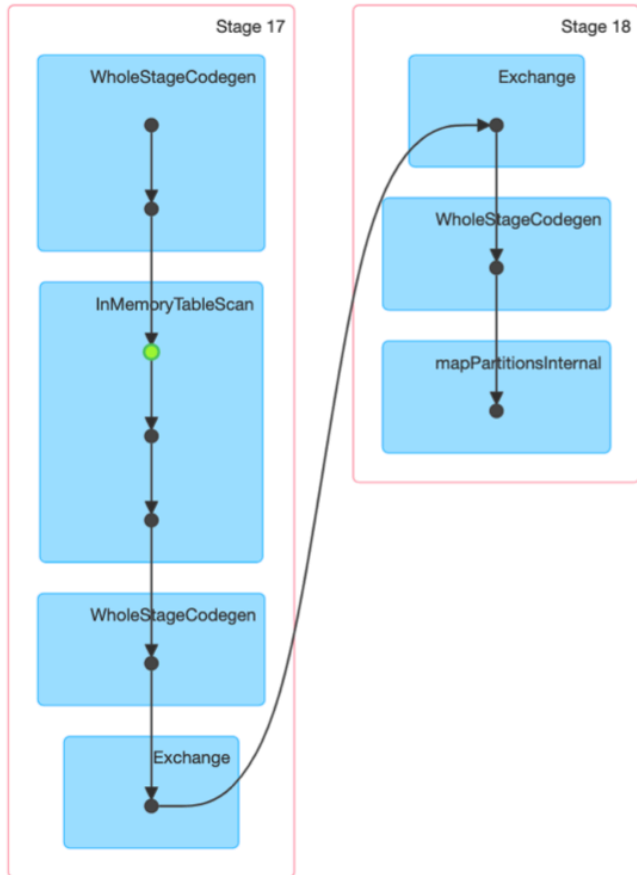
▸ Event Timeline
▾ DAG Visualization



Details for Job 14

Status: SUCCEEDED
Completed Stages: 2

▸ Event Timeline
▾ DAG Visualization



Spark deployment modes

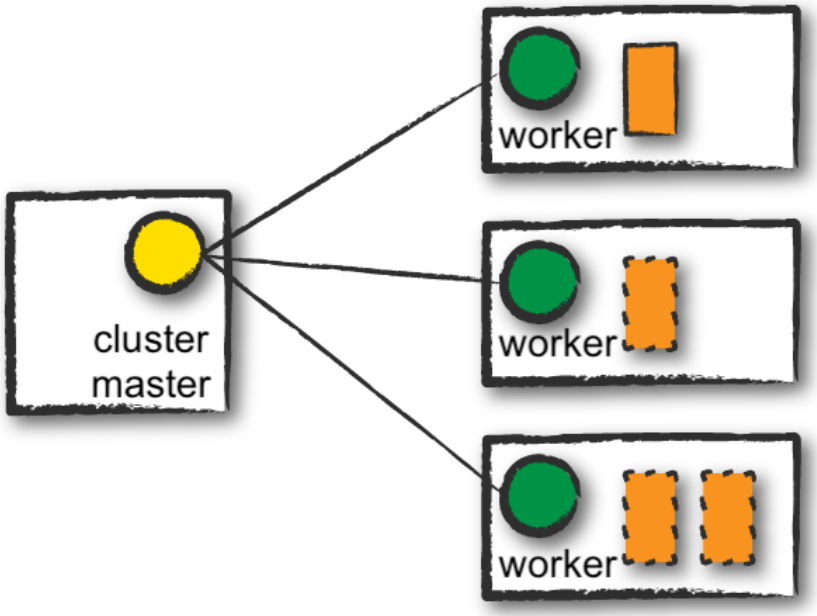
Application execution in local mode

In **local mode** (also called standalone mode) the entire application runs on a single machine.

Parallelism is only made possible by multithreading over the machine's CPU cores (and for some operations also GPUs).

Local mode is typically employed during application development for small/"not so big" datasets but not in production or for "really big" datasets.

Spark in cluster mode



A Spark cluster is composed of **master** and **worker nodes**. A **master** mediates access to workers in the cluster. **Workers** host driver or executor processes of Spark applications.

Each master/worker is normally tied to a distinct machine in a computer cluster, though other settings are also possible: e.g., a single machine hosting a master and worker simultaneously or multiple workers.

Cluster master - Spark UI

Spark Master at spark://caetano.local:7077

URL: spark://caetano.local:7077
Alive Workers: 3
Cores in use: 12 Total, 12 Used
Memory in use: 21.0 GB Total, 3.0 GB Used
Applications: 1 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (3)

Worker Id	Address	State	Cores	Memory
worker-20190306003850-192.168.1.4-63086	192.168.1.4:63086	ALIVE	4 (4 Used)	7.0 GB (1024.0 MB Used)
worker-20190306003929-192.168.1.4-63107	192.168.1.4:63107	ALIVE	4 (4 Used)	7.0 GB (1024.0 MB Used)
worker-20190306003929-192.168.1.4-63108	192.168.1.4:63108	ALIVE	4 (4 Used)	7.0 GB (1024.0 MB Used)

Cluster worker - Spark UI

Spark Worker at 192.168.1.4:63086

ID: worker-20190306003850-192.168.1.4-63086
Master URL: spark://caetano.local:7077
Cores: 4 (4 Used)
Memory: 7.0 GB (1024.0 MB Used)

Back to Master

Running Executors (1)

ExecutorID	Cores	State	Memory	Job Details	Logs
1	4	RUNNING	1024.0 MB	**ID:** app-20190306004548-0001 **Name:** PySparkShell **User:** edrdo	stdout stderr

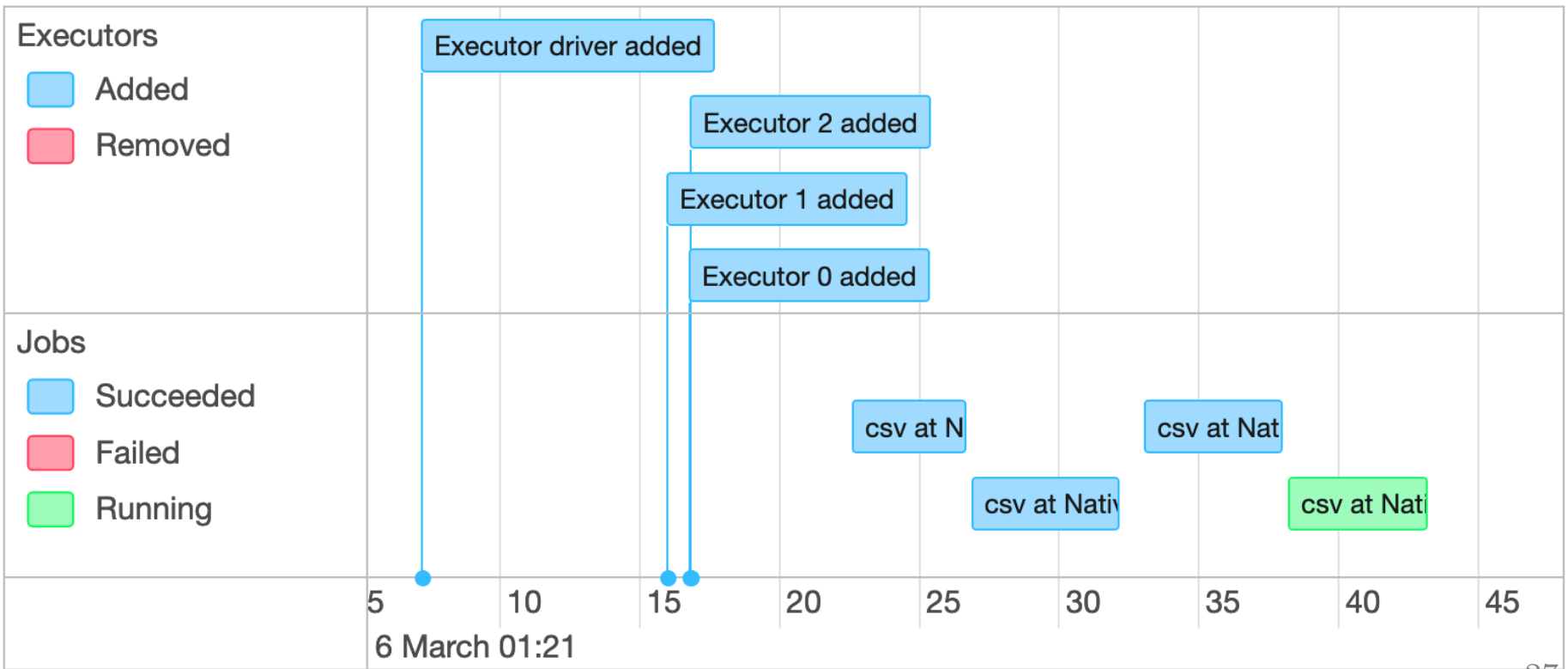
Application - Spark UI

Spark Jobs (?)

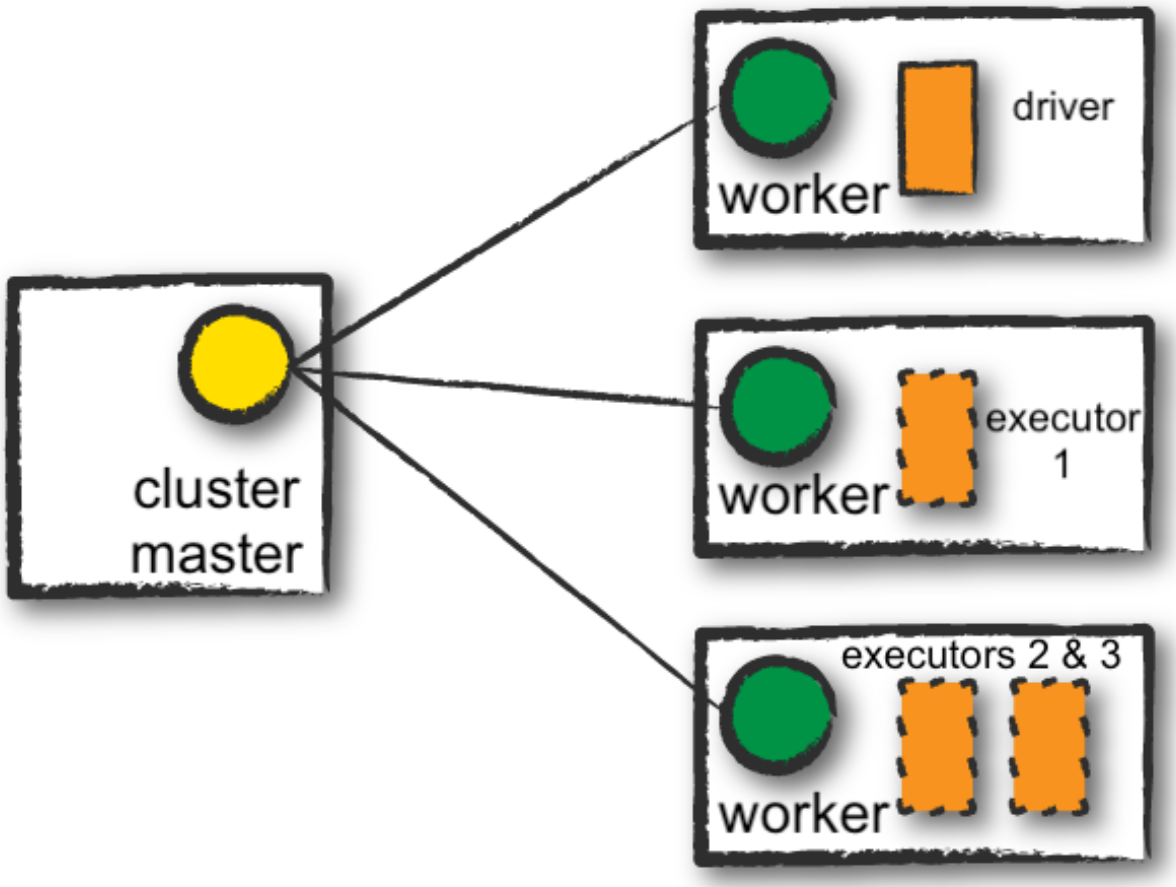
User: edrdo
Total Uptime: 38 s
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 3

▼ Event Timeline

☐ Enable zooming

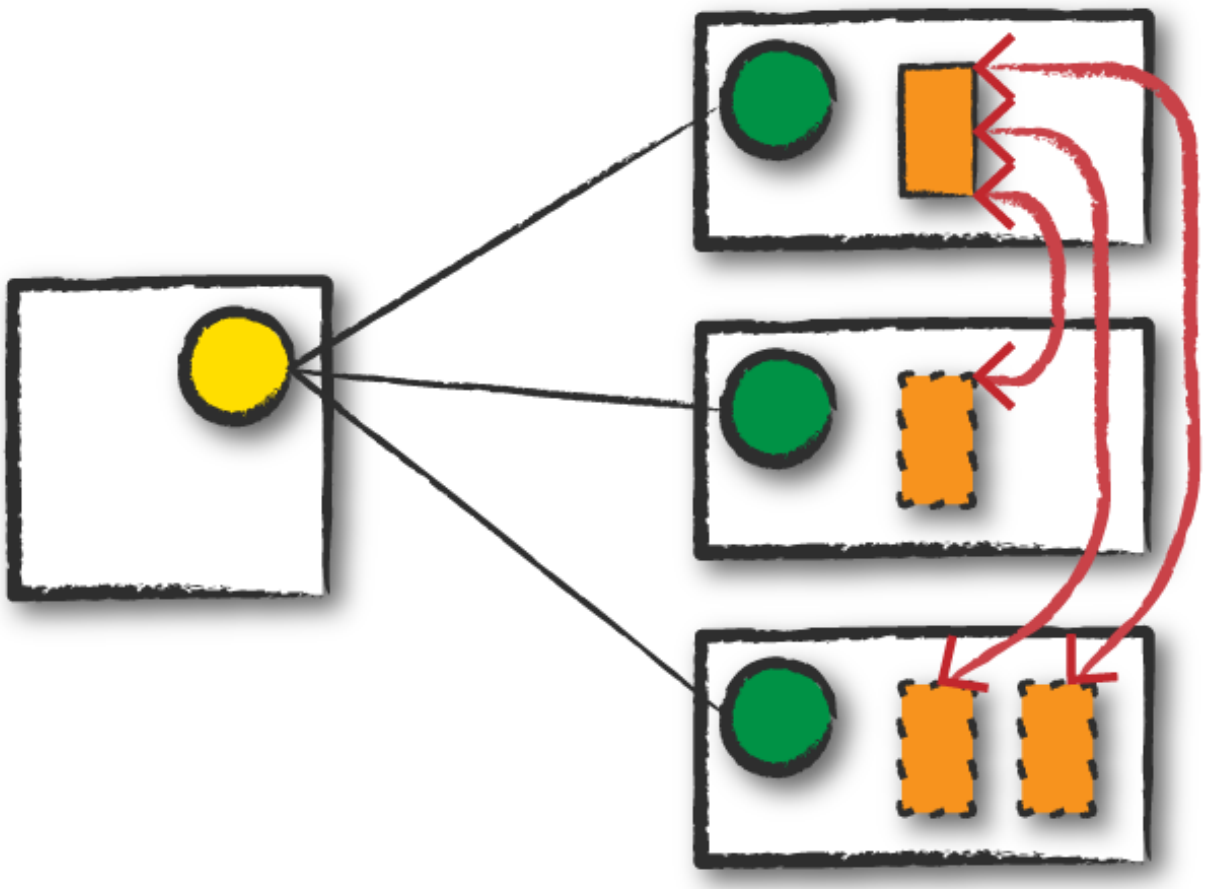


Application execution in cluster mode



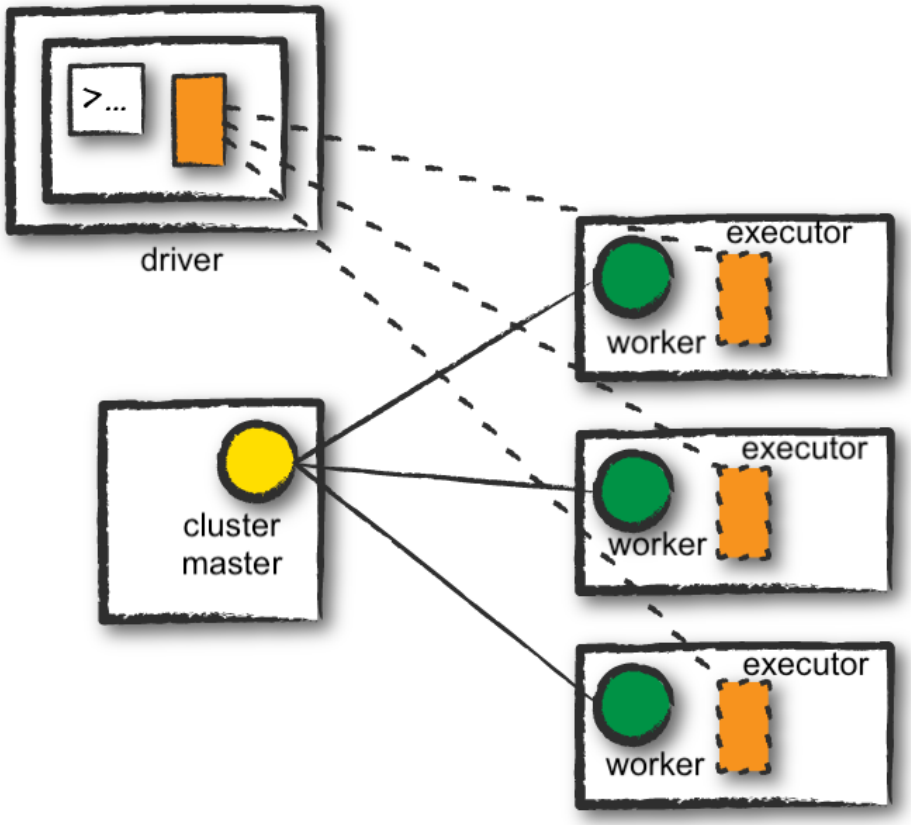
In **cluster mode**, a Spark application driver and its executors **all run inside** the cluster in association to workers.

Application execution in cluster mode (cont.)



Thus, in cluster mode, all driver/executor interaction for an application takes place within the cluster.

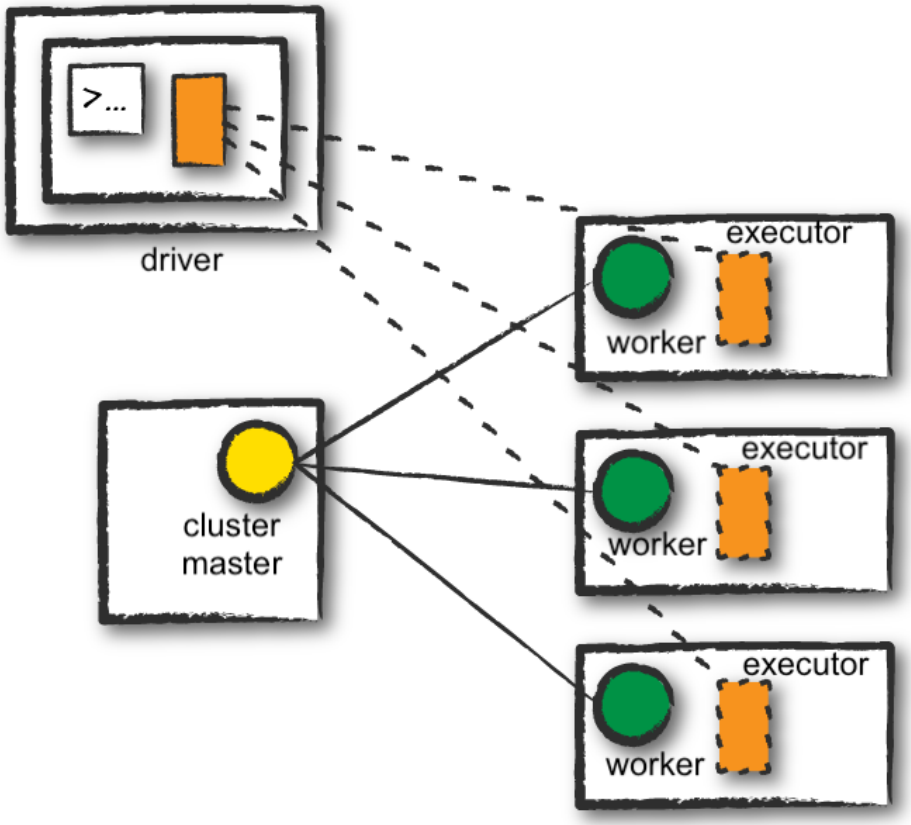
Application execution in client mode



In **client mode**, the application driver runs in a machine **outside** the cluster.

The driver's machine is often called a "gateway machine" or "edge node".

Application execution in client mode (cont.)



Client mode may be more convenient/flexible in a number of situations: if the driver program (but not executors) requires resources not accessible within the cluster; for security reasons (user code may not be trustworthy), ...

HDFS (Hadoop Distributed File System)

HDFS design goals

HDFS is commonly used in computer clusters as storage for Apache Spark, Hadoop MapReduce, and other frameworks in the Hadoop ecosystem.

HDFS, originally inspired by the Google File System (GFS), is a file system designed to store **very large files** across **distributed machines** in a **large cluster** with **streaming data access patterns**:

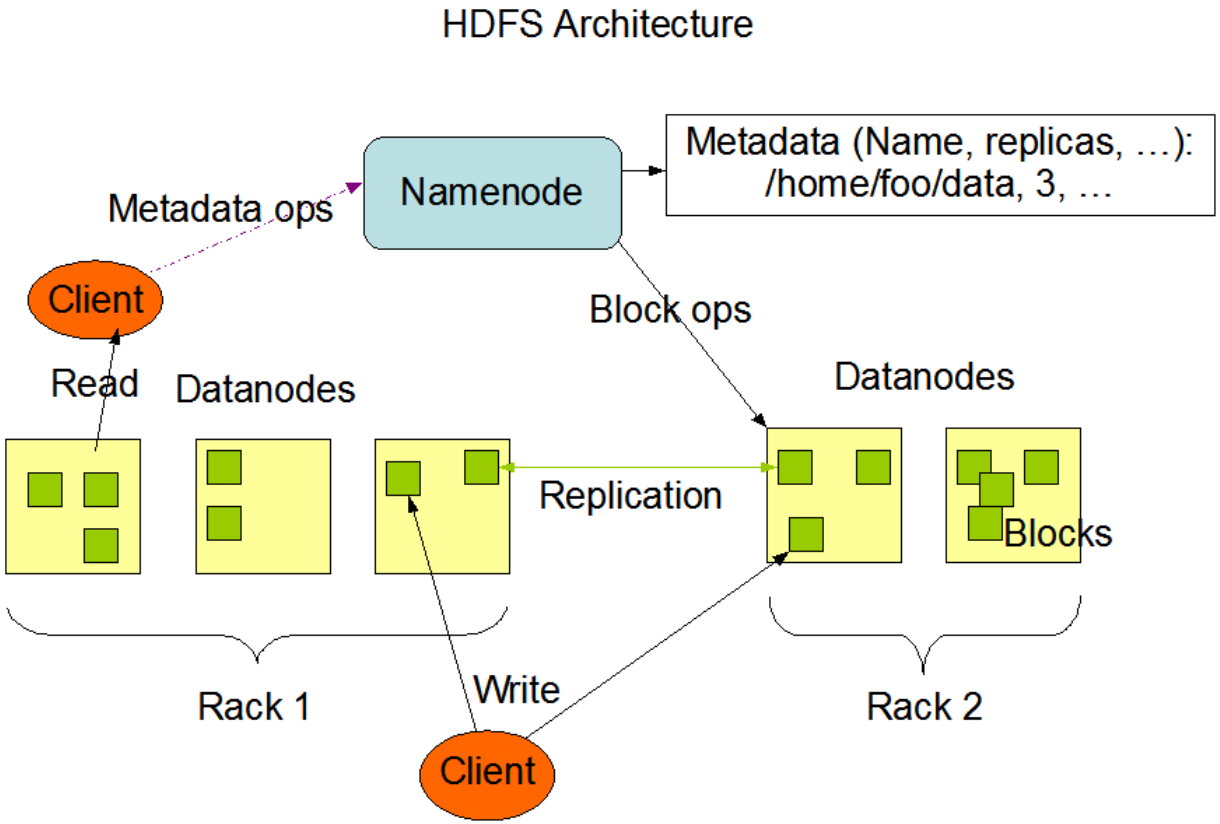
- **very large files** → hundreds of megabytes, gigabytes, or terabytes in size;
- **distributed** → data is distributed across several machines to allow for fault tolerance and parallel processing;
- **large clusters** → clusters can be formed by thousands of machines using commodity hardware (with non-negligible failure rate);
- **streaming data access pattern** → files are typically **written once** or in **append mode**, and **read many times** subsequently.

HDFS is not good for ...

HDFS is **not a good fit** for applications that require:

- **Multiple writers and/or random file access operations** → HDFS does not support these features.
- **Low-latency operation in the millisecond scale** → HDFS is oriented towards high throughput
- **Lots of small files** → HDFS is not designed with small files in mind, and a lot of small files in HDFS clusters in fact hurt performance.

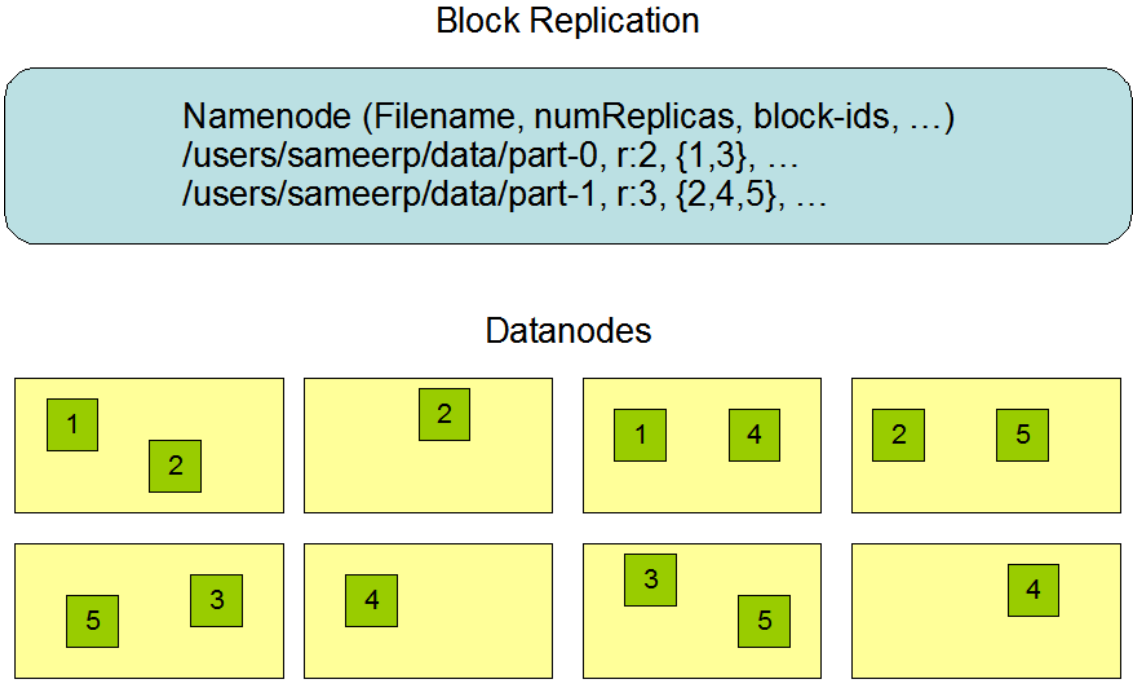
HDFS architecture



HDFS clusters are composed of **namenodes** and **datanodes**. **Namenodes** manage the file system and its meta-data, and **datanodes** provide actual storage. **Clients** access namenodes to get information about HDFS files, and datanodes to read and write data.

(Image from: HDFS Architecture, Apache Hadoop documentation)

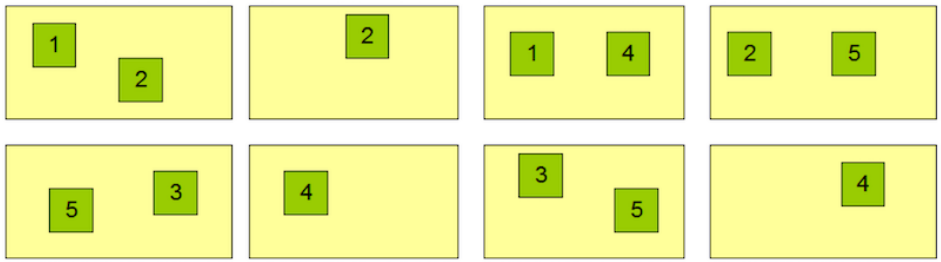
HDFS files



A HDFS file provides the abstraction of a single file, but is in fact divided into **blocks**, each with an equal size of typically 64 or 128 MB. A **block** is the elementary unit for read / write operations by **client** applications, and each is **stored and replicated independently** in different machines. The host file system in datanodes stores blocks as regular files.

(Image from: HDFS Architecture, Apache Hadoop documentation)

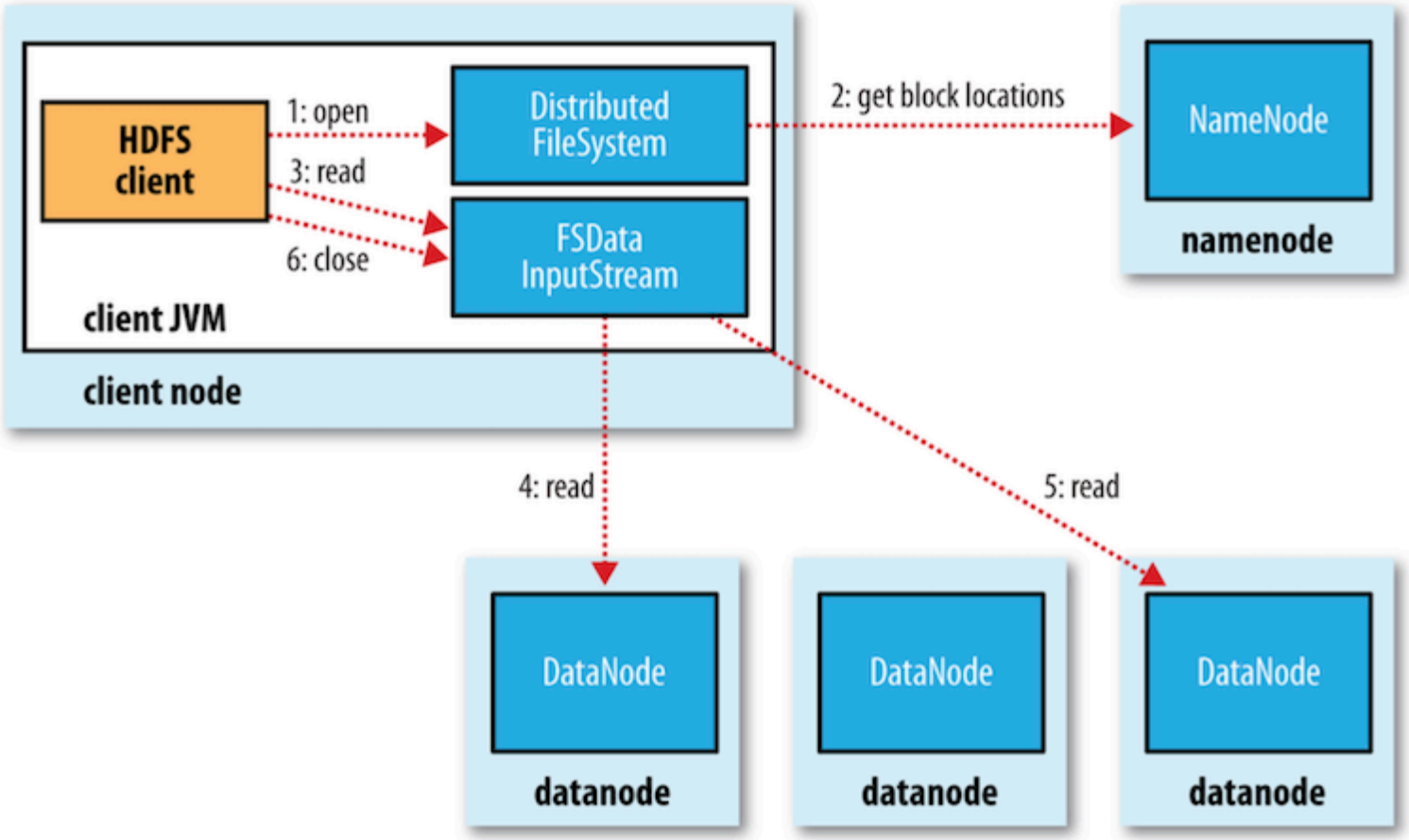
HDFS files (cont.)



Splitting a file into several replicated blocks has several advantages:

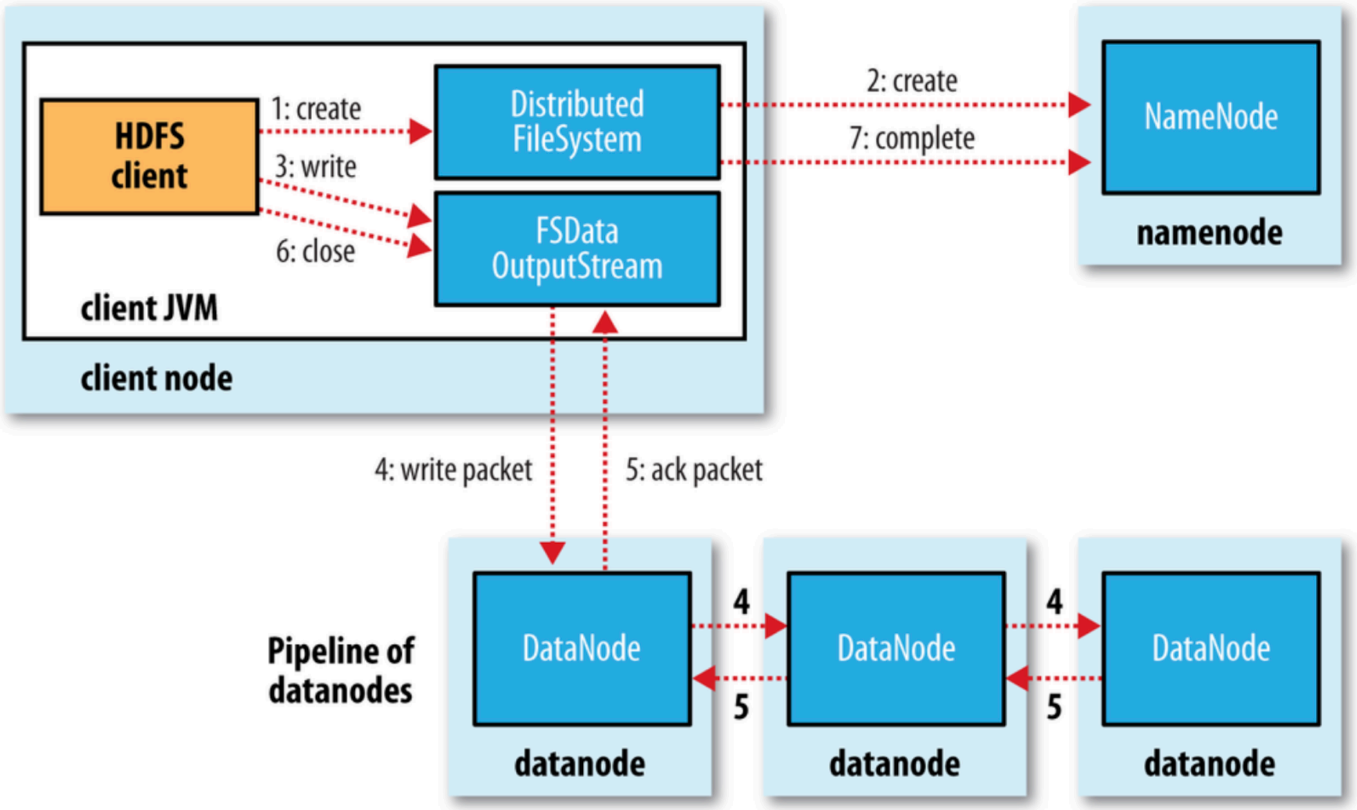
- **Support for "really big files"**: a HDFS file can be larger than any single disk in the network;
- **Fault tolerance / high availability**: if a block becomes unavailable from a datanode, a replica can be read from another datanode in a way that is transparent to the client.
- **Data locality in integration with MapReduce**: Hadoop MapReduce takes advantage of the block separation to **move computation near to where data is located**, rather than the other way round that would be quite more costly in terms of network communication/operation time. **Moving computation is cheaper than moving data.**

HDFS file reads



(Image from: Hadoop, The Definitive Guide, 4th ed.)

HDFS file writes



A pipeline is formed between datanodes to replicate blocks in response to block write operation by the client, according to the desired replication level (3 in the example above).

(Image from: Hadoop, The Definitive Guide, 4th ed.)