Programming Big Data 2016

Pierre Nugues, Peter Exner, **Marcus Klang**
Dennis Medved, Håkan Jonsson

Overview

- Week 1: Cloud architectures, Spark concepts, and Spark programming
- Week 2: Intermediate and advanced Spark
- Week 3: Supervised machine learning with Spark: MLlib and MLlib programming
- Week 4: Unsupervised machine learning

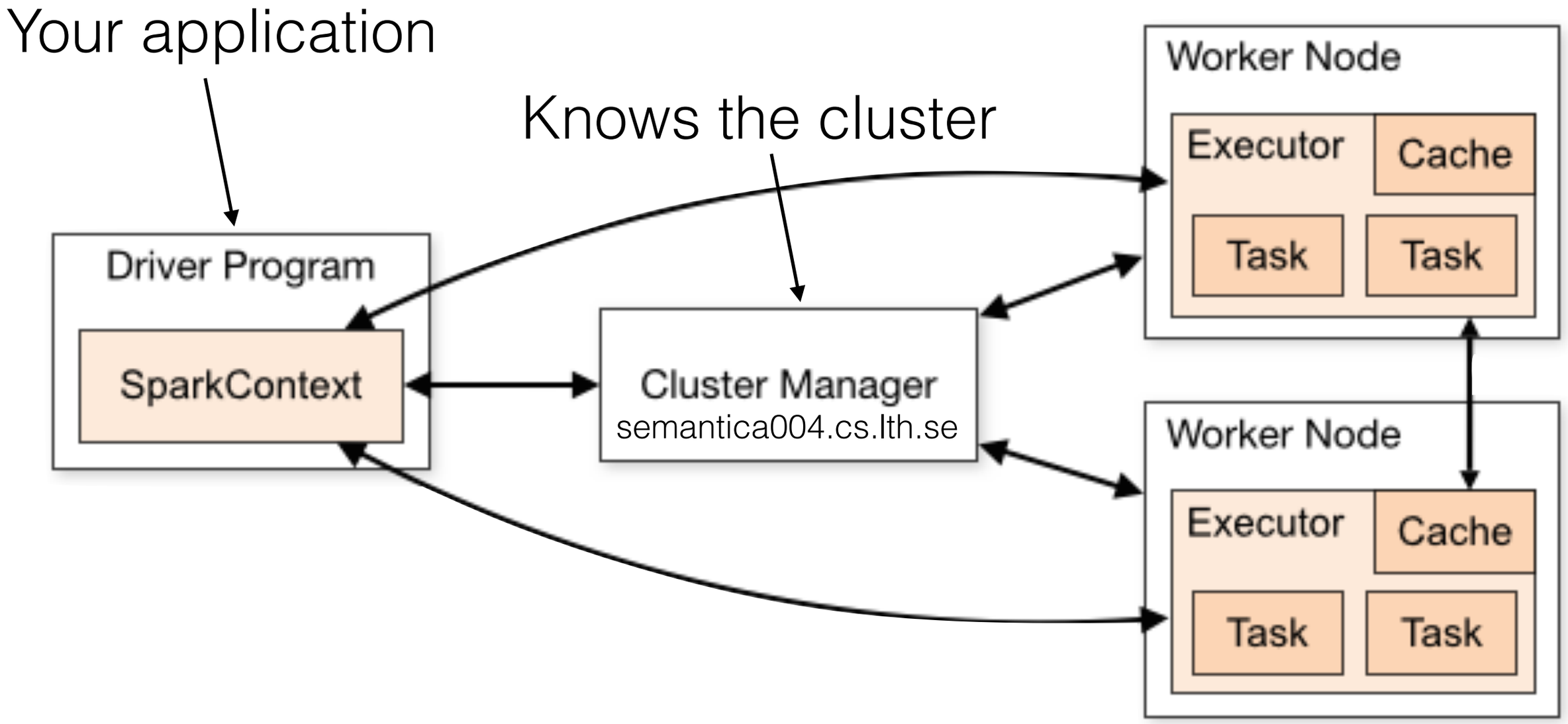
Week 2 overview

- Common tools
  - Spark UI, Pyspark job-submission, HDFS, Spark Context configuration
- Partitions, Performance, Spark SQL, Files/Dependencies,
- Exercises: Running on a real cluster, Spark SQL, Apache log analysis
- Assignment: Building a paragraph resolver using TF-IDF and Cosine similarity

Today's schedule

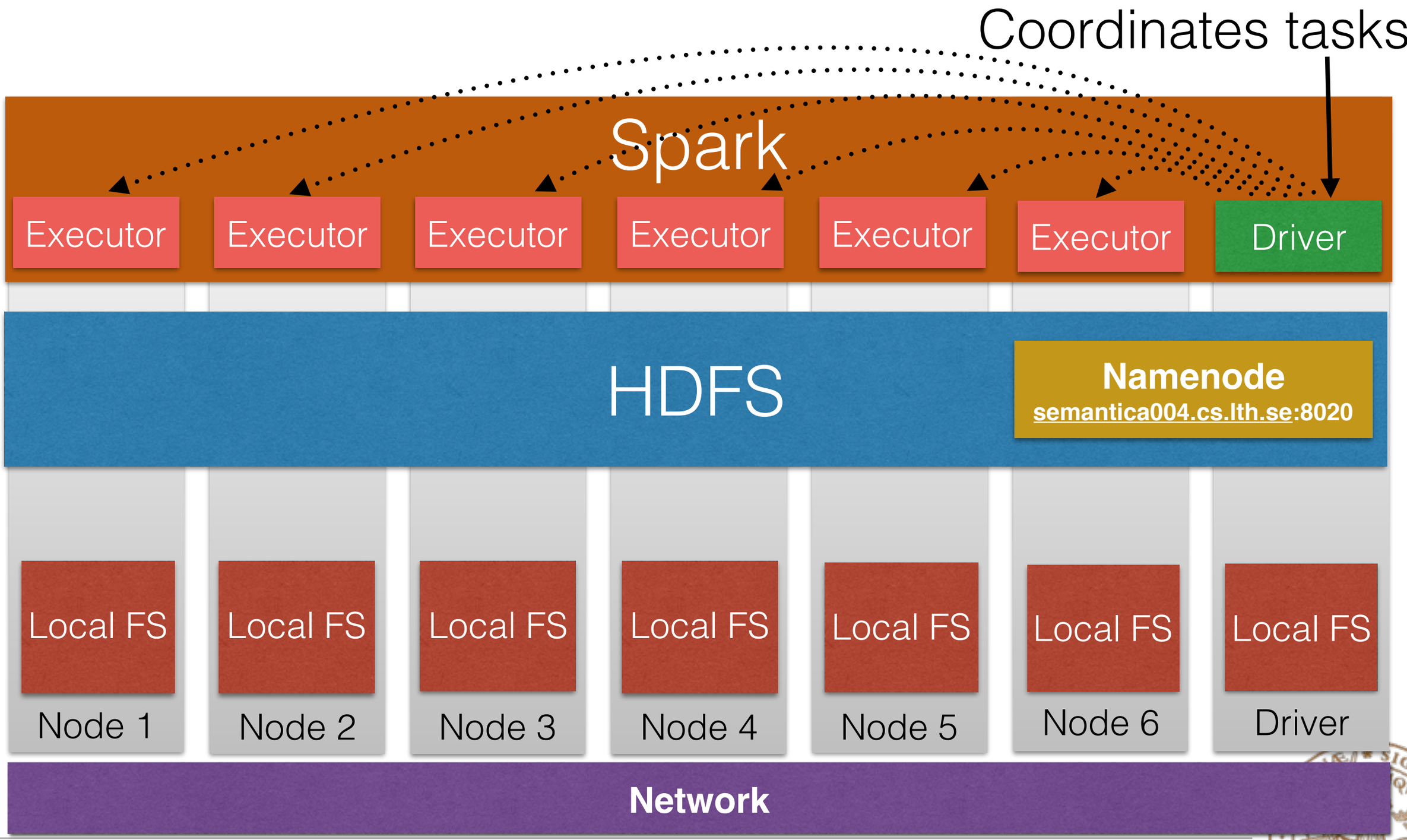
- 09:15 - 10:00 Spark Web UI, Common Tools, Job submission
- 10:15 - 11:15 Exercise 1: **Running on a real cluster!**
- 11:15 - 12:00 Structure, Dataframes, Spark SQL, User Defined Functions
- 12:00 - 13:15 Lunch break (IDEON)
- 13:15 - 14:00 Exercise 2: Spark SQL, Log analysis & Visualization
- 14:00 - 15:00 Joins, Partitions
- 15:15 - 16:00 Exercise 3: Joins & Wikipedia

Spark: Parts

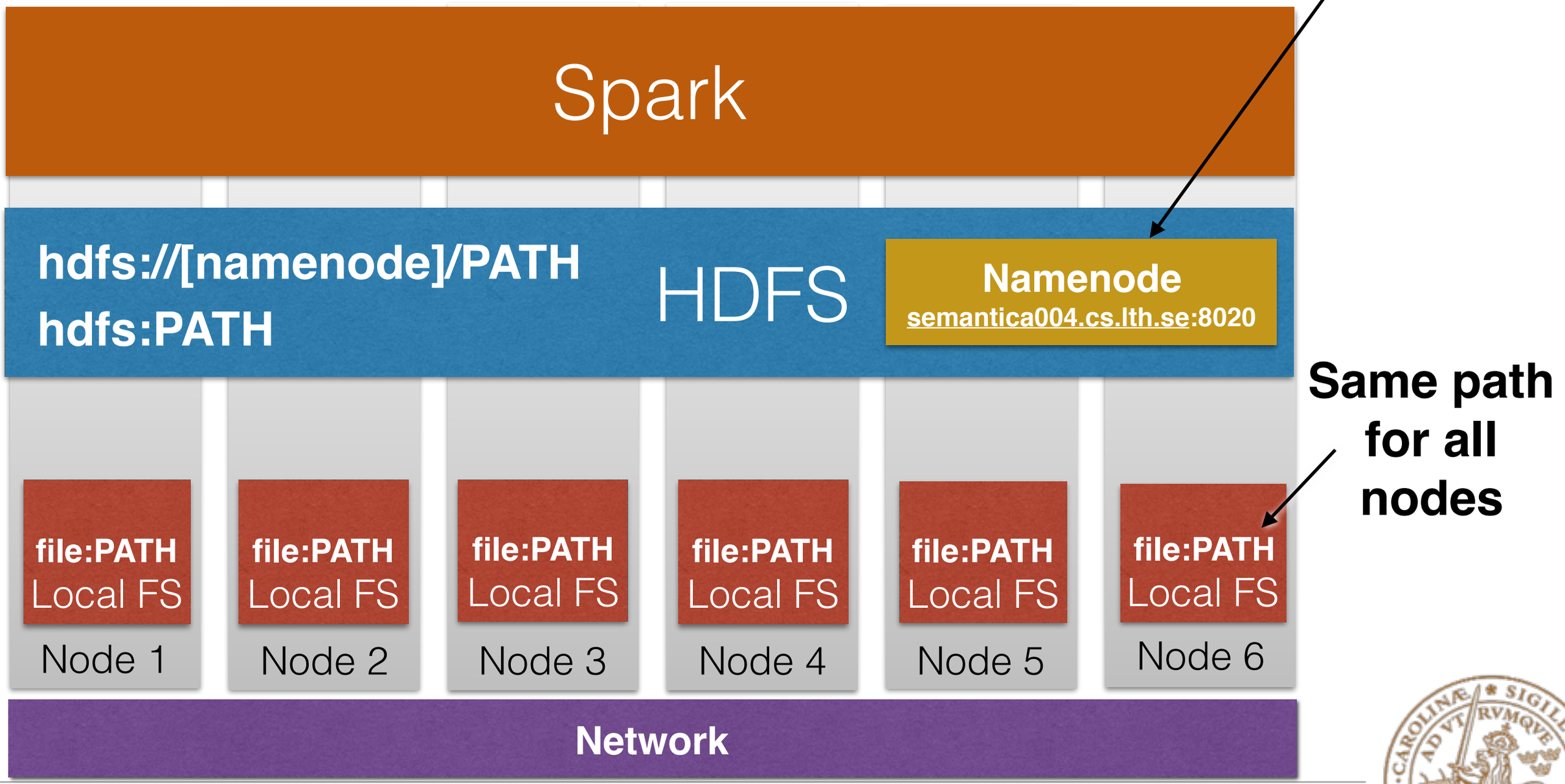


Source: https://spark.apache.org/docs/latest/cluster-overview.html

Spark: Cluster

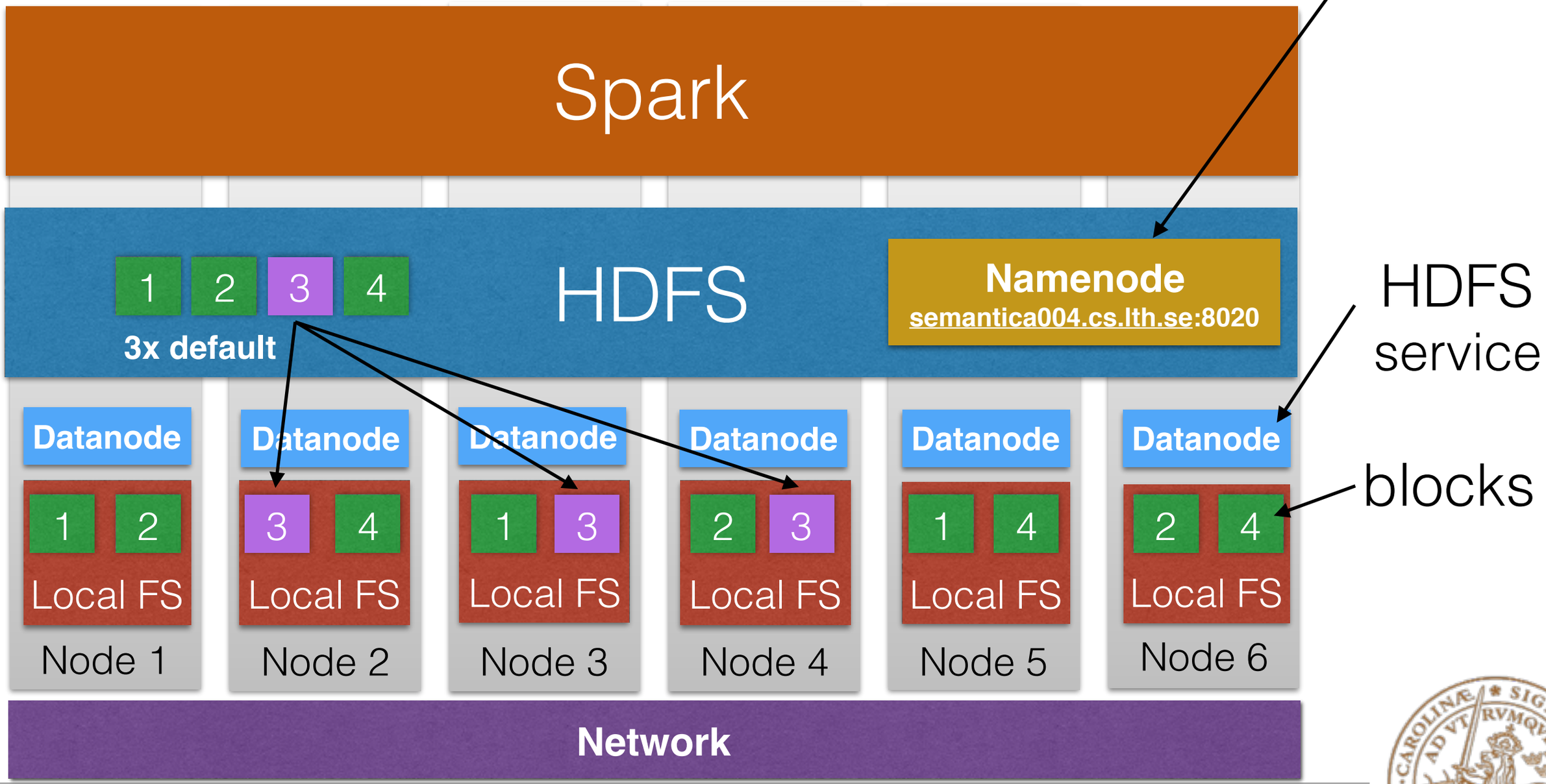


Spark: Cluster



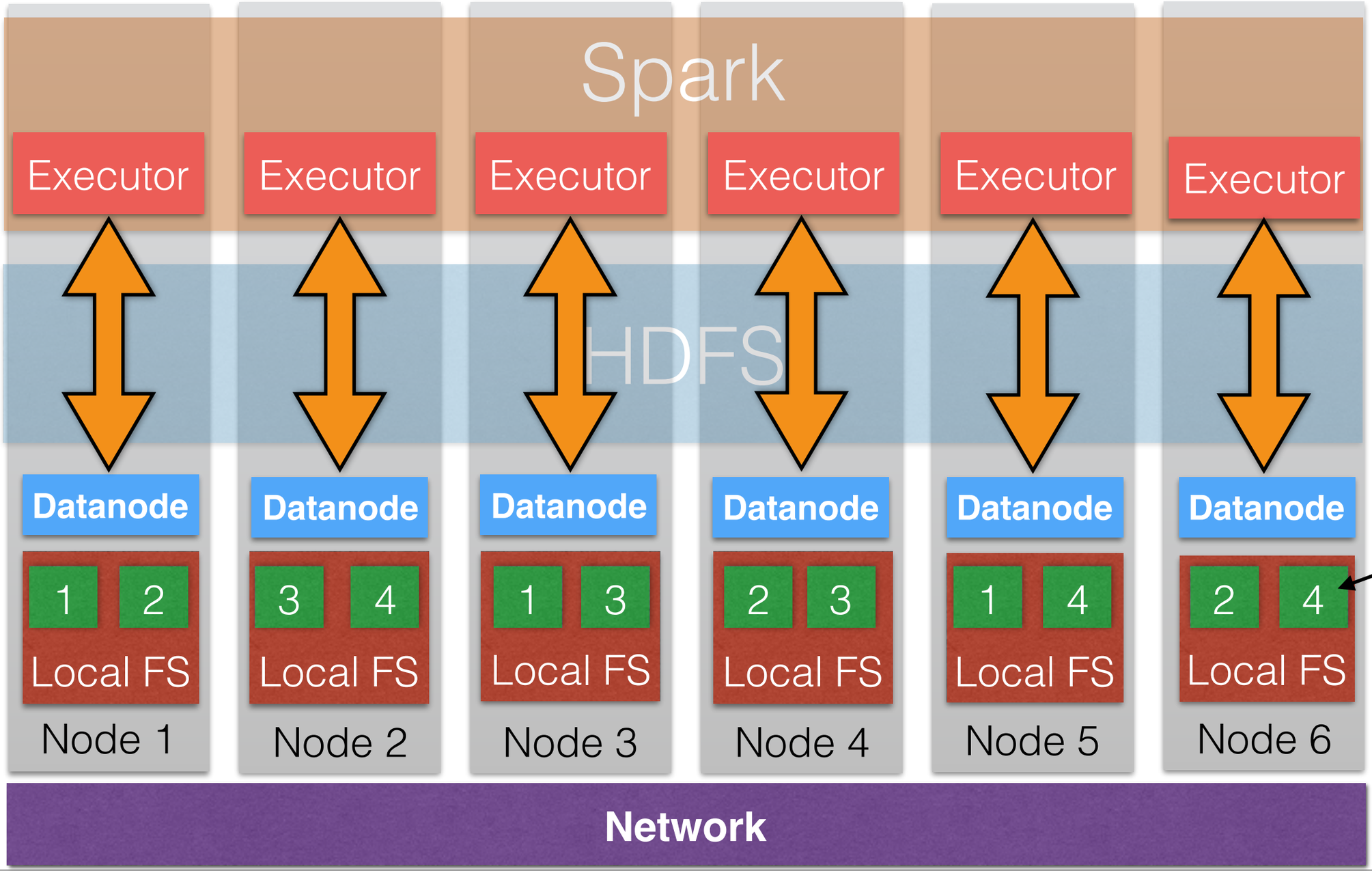
Spark: Cluster

Id Large binary block of data, might be partial files.



Spark: Cluster

Data is read locally if possible.



Cluster

- **What is the major difference between running locally and on a cluster?**
  - Different file systems
    - Local per node, global per cluster (e.g. HDFS, S3)
  - Scheduling of tasks takes more time
    - Minimal compared to traditional MapReduce on the Hadoop platform.
  - Shuffling is more expensive due to limited network bandwidth compared to RAM locally

Cluster

- **What is the major difference between running locally and on a cluster?**
  - Large cluster incurs high probability that a machine fails during a job
    - Spark will try to reschedule parts which fail onto other nodes transparently.
  - Environment and version differences
    - Our cluster does not run a version identical to the official Spark release.
    - Cloudera has recompiled and changed some aspects of the codebase.

Common Tools

- **Spark Web UI**
  - Information about current or previous jobs
- **File system tools: S3/HDFS/(Local)**
  - **Hadoop Distributed File System (HDFS)**: Our Cluster
    - Common directory, file operations onto HDFS
- **Pyspark job submission**
  - Tools to submit a job for execution

Spark Web UI

- Multiple interfaces

  - **History server (default port: 18088)**

    ➡ Statistics and information about archived jobs, outputs per node that can be useful for debugging.

  - **Cluster manager (default port: 18080)**

    ➡ Shows global status, e.g. what applications are running

  - **Application (default port: 4040, 4041, 4042, …)**

    ➡ Current status of a job, only exists while a SparkContext for a job is active.

Spark UI
History server

History Server

semantica004.cs.lth.se:18088/?page=1&showIncomplete=false

Spark 1.3.0 **History Server**

Event log directory: hdfs://semantica004.cs.lth.se:8020/user/spark/applicationHistory

Showing 1-20 of 57

1 2 3 >

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
app-20150824094731-0056	PySparkShell	2015/08/24 09:47:28	2015/08/24 10:43:23	56 min	pnugues	2015/08/24 10:43:23
app-20150824090504-0042	PySparkShell	2015/08/24 09:05:01	2015/08/24 09:10:41	5.7 min	pnugues	2015/08/24 09:10:42
app-20150821173652-0041	PySparkShell	2015/08/21 17:36:50	2015/08/21 17:40:51	4.0 min	pnugues	2015/08/21 17:40:52
app-20150821165233-0040	PySparkShell	2015/08/21 16:52:30	2015/08/21 17:19:36	27 min	pnugues	2015/08/21 17:19:36
app-20150821164329-0039	PySparkShell	2015/08/21 16:43:26	2015/08/21 16:52:19	8.9 min	pnugues	2015/08/21 16:52:19
app-20150821163248-0035	PySparkShell	2015/08/21 16:32:46	2015/08/21 16:36:00	3.2 min	pnugues	2015/08/21 16:36:00
app-20150821161453-0033	PySparkShell	2015/08/21 16:14:51	2015/08/21 16:15:15	24 s	pnugues	2015/08/21 16:15:15
app-20150821155955-0031	PySparkShell	2015/08/21 15:59:52	2015/08/21 16:04:06	4.2 min	pnugues	2015/08/21 16:04:06
app-20150821155025-0029	PySparkShell	2015/08/21 15:50:22	2015/08/21 15:53:52	3.5 min	pnugues	2015/08/21 15:53:52
app-20150821154154-0026	PySparkShell	2015/08/21 15:41:51	2015/08/21 15:47:02	5.2 min	pnugues	2015/08/21 15:47:02
app-20150821152705-0024	PySparkShell	2015/08/21 15:27:03	2015/08/21 15:39:08	12 min	pnugues	2015/08/21 15:39:08
app-20150821145251-0022	PySparkShell	2015/08/21 14:52:49	2015/08/21 14:53:15	27 s	pnugues	2015/08/21 14:53:15
app-20150821144131-0016	PySparkShell	2015/08/21 14:41:29	2015/08/21 14:43:34	2.1 min	pnugues	2015/08/21 14:43:34
app-20150821142806-0012	PySparkShell	2015/08/21 14:28:03	2015/08/21 14:38:45	11 min	pnugues	2015/08/21 14:38:46
app-20150821133241-0010	PySparkShell	2015/08/21 13:32:38	2015/08/21 14:27:24	55 min	pnugues	2015/08/21 14:27:24
app-20150821111000-0017	PySparkShell	2015/08/21 11:09:52	2015/08/21 11:12:18	2.4 min	pnugues	2015/08/21 11:12:18
app-20150821103652-0016	PySparkShell	2015/08/21 10:36:49	2015/08/21 11:05:52	29 min	pnugues	2015/08/21 11:05:52
app-20150821102858-0015	PySparkShell	2015/08/21 10:28:54	2015/08/21 10:36:32	7.6 min	pnugues	2015/08/21 10:36:32
local-1440145363298	PySparkShell	2015/08/21 10:22:40	2015/08/21 10:36:27	14 min	pnugues	2015/08/21 10:36:27
app-20150820170530-0002	PySparkShell	2015/08/20 17:05:26	2015/08/21 09:08:09	16.0 h	pnugues	2015/08/21 09:08:09

Show incomplete applications

Spark UI - History server

- The Spark Application UI shutdowns when the job has finished or failed
- The data gathered by the application is saved on the history server:
  - Limited time
  - All statistics gathered during execution is saved here

Spark UI

Cluster manager

Spark Master at spark://se

semantica004.cs.lth.se:18080

Spark 1.3.0

Spark Master at spark://semantica004.cs.lth.se:7077

URL: spark://semantica004.cs.lth.se:7077
REST URL: spark://semantica004.cs.lth.se:6066 *(cluster mode)*
Workers: 12
Cores: 144 Total, 0 Used
Memory: 234.4 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20150827150930-semantica005.cs.lth.se-7078	semantica005.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)
worker-20150827150930-semantica007.cs.lth.se-7078	semantica007.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)
worker-20150827150930-semantica009.cs.lth.se-7078	semantica009.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)
worker-20150827150930-semantica010.cs.lth.se-7078	semantica010.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)
worker-20150827150930-semantica012.cs.lth.se-7078	semantica012.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)
worker-20150827150930-semantica013.cs.lth.se-7078	semantica013.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)
worker-20150827150930-semantica014.cs.lth.se-7078	semantica014.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)
worker-20150827150931-semantica001.cs.lth.se-7078	semantica001.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)
worker-20150827150931-semantica002.cs.lth.se-7078	semantica002.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)
worker-20150827150931-semantica003.cs.lth.se-7078	semantica003.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)
worker-20150827150931-semantica008.cs.lth.se-7078	semantica008.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)
worker-20150827150931-semantica011.cs.lth.se-7078	semantica011.cs.lth.se:7078	ALIVE	12 (0 Used)	19.5 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Spark UI
Cluster manager

Spark Master at spark://se ×

semantica004.cs.lth.se:18080

Spark 1.3.0

Spark Master at spark://semantica004.cs.lth.se:7077

URL: spark://semantica004.cs.lth.se:7077
REST URL: spark://semantica004.cs.lth.se:6066 *(cluster mode)*
Workers: 12
Cores: 144 Total, 0 Used
Memory: 234.4 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State
worker-20150827150930-semantica005.cs.lth.se-7078	semantica005.cs.lth.se:7078	ALIVE
worker-20150827150930-semantica007.cs.lth.se-7078	semantica007.cs.lth.se:7078	ALIVE
worker-20150827150930-semantica009.cs.lth.se-7078	semantica009.cs.lth.se:7078	ALIVE
worker-20150827150930-semantica010.cs.lth.se-7078	semantica010.cs.lth.se:7078	ALIVE
worker-20150827150930-semantica012.cs.lth.se-7078	semantica012.cs.lth.se:7078	ALIVE
worker-20150827150930-semantica013.cs.lth.se-7078	semantica013.cs.lth.se:7078	ALIVE
worker-20150827150930-semantica014.cs.lth.se-7078	semantica014.cs.lth.se:7078	ALIVE
worker-20150827150931-semantica001.cs.lth.se-7078	semantica001.cs.lth.se:7078	ALIVE
worker-20150827150931-semantica002.cs.lth.se-7078	semantica002.cs.lth.se:7078	ALIVE

Spark UI — Cluster Manager

- Global status
- Shows jobs queued and running
- Access to individual node outputs

Spark UI — Application

- Status of job
- Provides failure information
- Current progress and detailed per partition statistics: such as size, record count, and shuffle sizes
- Provides means of terminating jobs.
- Provide a REST API to interact with other applications
- One per driver is started.

Spark UI
Application - Jobs

PySparkShell - Spark Jobs

klang.cs.lth.se:4041/jobs/

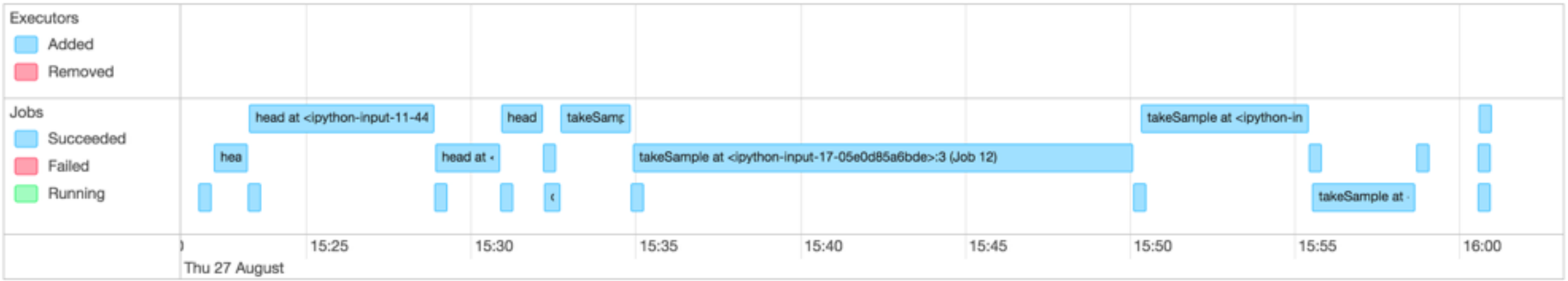
Spark 1.4.0 | Jobs | Stages | Storage | Environment | Executors | PySparkShell application UI

Spark Jobs (?)

Total Uptime: 115,5 h
Scheduling Mode: FIFO
Completed Jobs: 57
Failed Jobs: 5

▾ Event Timeline

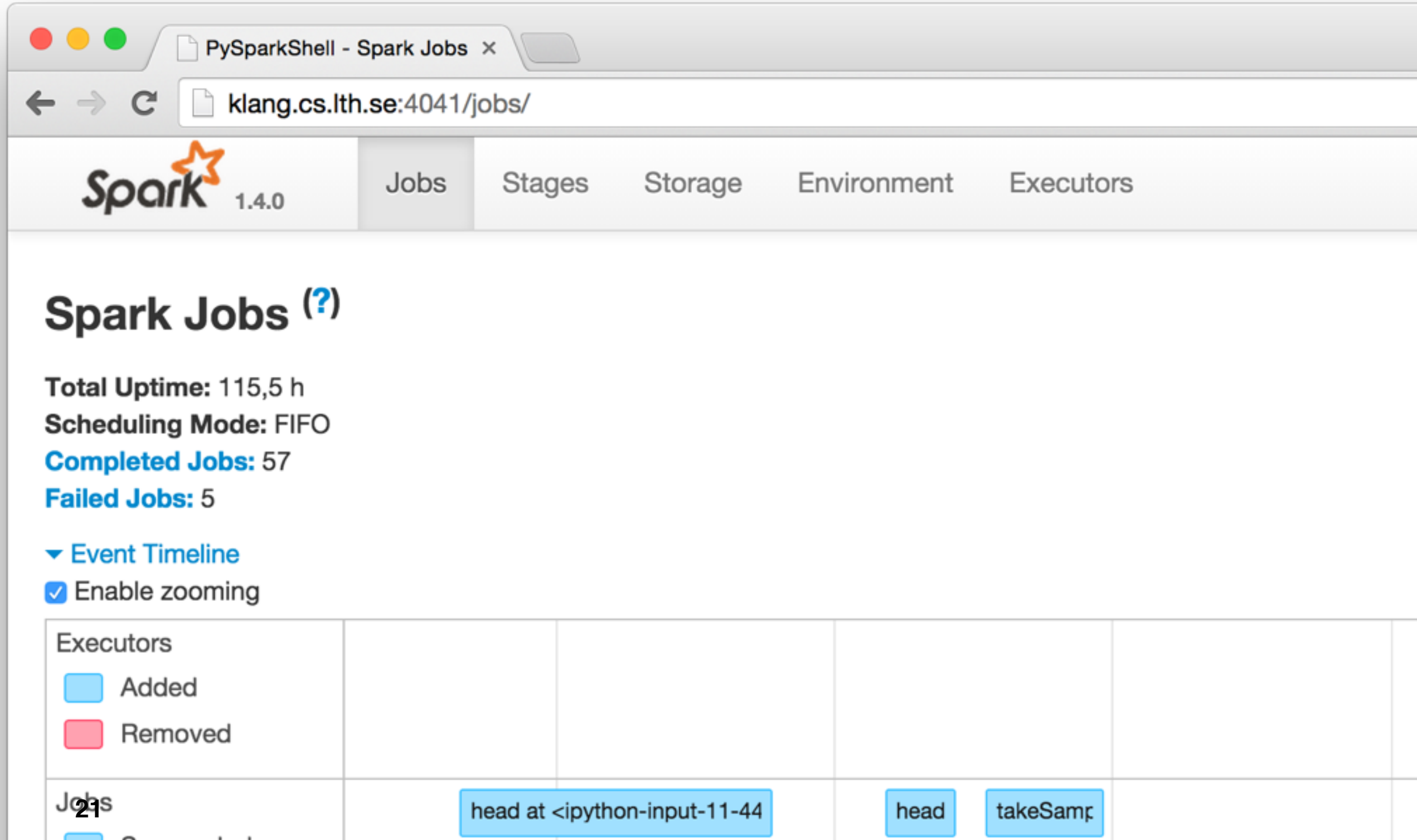
☑ Enable zooming

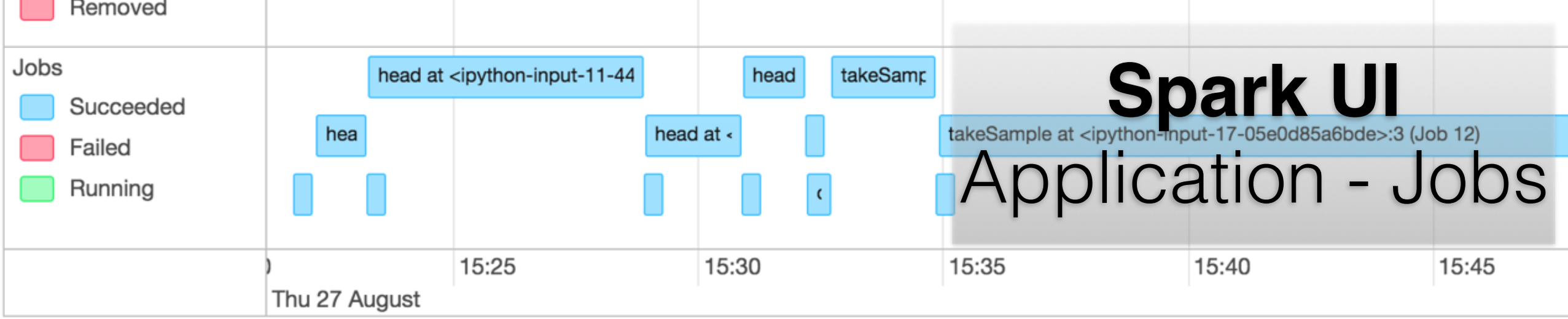


Completed Jobs (57)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
61	parquet at NativeMethodAccessorImpl.java:-2	2015/08/31 16:55:19	1,2 min	1/1 (4 skipped)	88/88 (7418 skipped)
60	runJob at PythonRDD.scala:366	2015/08/31 16:55:18	0,4 s	1/1 (4 skipped)	1/1 (7418 skipped)
59	runJob at PythonRDD.scala:366	2015/08/31 16:50:06	5,2 min	2/2 (3 skipped)	2935/2935 (4484 skipped)
58	runJob at PythonRDD.scala:366	2015/08/31 16:45:05	3 s	1/1	1/1
57	collect at <ipython-input-150-2269b22c6317>:1	2015/08/31 16:43:47	1,1 min	2/2	2342/2342
56	runJob at PythonRDD.scala:366	2015/08/31 16:41:23	39 s	1/1	467/467
55	runJob at PythonRDD.scala:366	2015/08/31 16:41:15	8 s	1/1	100/100
54	runJob at PythonRDD.scala:366	2015/08/31 16:40:58	17 s	1/1	20/20
53	runJob at PythonRDD.scala:366	2015/08/31 16:40:48	10 s	1/1	4/4
52	runJob at PythonRDD.scala:366	2015/08/31 16:40:35	13 s	1/1	1/1

Spark UI
Application - Jobs





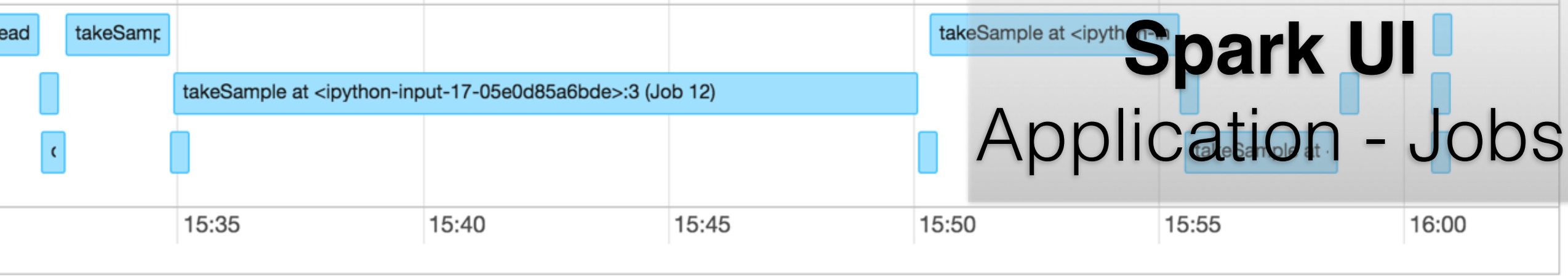
Spark UI

Application - Jobs

Completed Jobs (57)

Job Id	Description	Submitted	Duration	Stages: Succeeded
61	parquet at NativeMethodAccessorImpl.java:-2	2015/08/31 16:55:19	1,2 min	1/1 (4 skipped)
60	runJob at PythonRDD.scala:366	2015/08/31 16:55:18	0,4 s	1/1 (4 skipped)
59	runJob at PythonRDD.scala:366	2015/08/31 16:50:06	5,2 min	2/2 (3 skipped)
58	runJob at PythonRDD.scala:366	2015/08/31 16:45:05	3 s	1/1
57	collect at <ipython-input-150-2269b22c6317>:1	2015/08/31 16:43:47	1,1 min	2/2
56	runJob at PythonRDD.scala:366	2015/08/31 16:41:23	39 s	1/1
55	runJob at PythonRDD.scala:366	2015/08/31 16:41:15	8 s	1/1
54	runJob at PythonRDD.scala:366	2015/08/31 16:40:58	17 s	1/1
53	runJob at PythonRDD.scala:366	2015/08/31 16:40:48	10 s	1/1
52	runJob at PythonRDD.scala:366	2015/08/31 16:40:35	13 s	1/1

Spark UI
Application - Jobs



	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
	2015/08/31 16:55:19	1,2 min	1/1 (4 skipped)	88/88 (7418 skipped)
	2015/08/31 16:55:18	0,4 s	1/1 (4 skipped)	1/1 (7418 skipped)
	2015/08/31 16:50:06	5,2 min	2/2 (3 skipped)	2935/2935 (4484 skipped)
	2015/08/31 16:45:05	3 s	1/1	1/1
	2015/08/31 16:43:47	1,1 min	2/2	2342/2342
	2015/08/31 16:41:23	39 s	1/1	467/467
	2015/08/31 16:41:15	8 s	1/1	100/100
	2015/08/31 16:40:58	17 s	1/1	20/20
	2015/08/31 16:40:48	10 s	1/1	4/4
	2015/08/31 16:40:35	13 s	1/1	1/1

Spark UI
Application - Stages

PySparkShell - Stages for

klang.cs.lth.se:4041/stages/

Spark 1.4.0 | Jobs | Stages | Storage | Environment | Executors

Stages for All Jobs

Completed Stages: 91
Failed Stages: 5

Completed Stages (91)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
111	parquet at NativeMethodAccessorImpl.java:-2	+details	2015/08/31 16:55:19	1,2 min	88/88			2046.6 MB	
106	runJob at PythonRDD.scala:366	+details	2015/08/31 16:55:18	0,4 s	1/1				
101	runJob at PythonRDD.scala:366	+details	2015/08/31 16:55:18	0,3 s	1/1				
100	repartition at NativeMethodAccessorImpl.java:-2	+details	2015/08/31 16:50:06	5,2 min	2934/2934	40.9 GB		370.4 MB	2046.6 MB
96	runJob at PythonRDD.scala:366	+details	2015/08/31 16:45:05	3 s	1/1	1339.3 KB			
95	collect at <ipython-input-150-2269b22c6317>:1		2015/08/31 16:44:47	4 s	200/200			45.3 MB	
94	javaToPython at null:-1	+details	2015/08/31 16:43:47	1,0 min	2142/2142	80.2 GB			45.3 MB
93	runJob at PythonRDD.scala:366	+details	2015/08/31 16:41:23	39 s	467/467	609.0 MB			
92	runJob at PythonRDD.scala:366	+details	2015/08/31 16:41:15	8 s	100/100	126.9 MB			
91	runJob at PythonRDD.scala:366	+details	2015/08/31 16:40:58	17 s	20/20	29.9 MB			
90	runJob at PythonRDD.scala:366	+details	2015/08/31 16:40:48	10 s	4/4	4.0 MB			
89	runJob at PythonRDD.scala:366	+details	2015/08/31 16:40:35	13 s	1/1	1339.3 KB			
88	collect at <ipython-input-144-ba77ca76e8be>:1		2015/08/31 16:40:05	3 s	200/200			45.3 MB	
87	collect at <ipython-input-144-ba77ca76e8be>:1		2015/08/31 16:39:01	1,1 min	2142/2142	80.2 GB			45.3 MB
85	runJob at PythonRDD.scala:366	+details	2015/08/31 16:38:26	5 s	1/1	5.9 MB			
83	runJob at PythonRDD.scala:366	+details	2015/08/31 16:32:48	0,2 s	1/1			1893.7 KB	
82	javaToPython at null:-1	+details	2015/08/31 16:32:24	25 s	200/200	37.3 MB			20.1 MB
80	javaToPython at null:-1	+details	2015/08/31 16:32:24	23 s	2142/2142	40.1 GB			350.2 MB
74	javaToPython at null:-1	+details	2015/08/31 16:30:37	4 s	200/200			45.3 MB	20.1 MB
73	cache at NativeMethodAccessorImpl.java:-2	+details	2015/08/31 16:29:06	1,5 min	2142/2142	80.2 GB			45.3 MB

torage Environment Executors

PySparkShell application UI

Spark UI
Application - Stages

		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
va:-2	+details	2015/08/31 16:55:19	1,2 min	88/88			2046.6 MB	
	+details	2015/08/31 16:55:18	0,4 s	1/1				
	+details	2015/08/31 16:55:18	0,3 s	1/1				
java:-2	+details	2015/08/31 16:50:06	5,2 min	2934/2934	40.9 GB		370.4 MB	2046.6 MB
	+details	2015/08/31 16:45:05	3 s	1/1	1339.3 KB			
317>:1		2015/08/31 16:44:47	4 s	200/200			45.3 MB	
	+details	2015/08/31 16:43:47	1,0 min	2142/2142	80.2 GB			45.3 MB
	+details	2015/08/31 16:41:23	39 s	467/467	609.0 MB			
	+details	2015/08/31 16:41:15	8 s	100/100	126.9 MB			
	+details	2015/08/31 16:40:58	17 s	20/20	29.9 MB			
	+details	2015/08/31 16:40:48	10 s	4/4	4.0 MB			
	+details	2015/08/31 16:40:35	13 s	1/1	1339.3 KB			
3be>:1		2015/08/31 16:40:05	3 s	200/200			45.3 MB	
3be>:1		2015/08/31 16:39:01	1,1 min	2142/2142	80.2 GB			45.3 MB
	+details	2015/08/31 16:38:26	5 s	1/1	5.9 MB			
	+details	2015/08/31 16:32:48	0,2 s	1/1			1893.7 KB	
	+details	2015/08/31 16:32:24	25 s	200/200	37.3 MB			20.1 MB
	+details	2015/08/31 16:32:24	23 s	2142/2142	40.1 GB			350.2 MB
	+details	2015/08/31 16:30:37	4 s	200/200			45.3 MB	20.1 MB
:-2	+details	2015/08/31 16:29:06	1,5 min	2142/2142	80.2 GB			45.3 MB

Spark UI
Application - Stage details

PySparkShell - Details for

klang.cs.lth.se:4041/stages/stage/?id=94&attempt=0

Spark 1.4.0 | Jobs | Stages | Storage | Environment | Executors

Details for Stage 94 (Attempt 0)

Total Time Across All Tasks: 24 min
Input Size / Records: 80.2 GB / 18277096
Shuffle Write: 45.3 MB / 2132216

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 2142 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0,2 s	0,5 s	0,6 s	0,6 s	7 s
GC Time	0 ms	0 ms	0 ms	0 ms	2 s
Input Size / Records	2.3 MB / 470	12.4 MB / 5006	24.0 MB / 7074	39.2 MB / 9008	115.3 MB / 21294
Shuffle Write Size / Records	5.0 KB / 60	12.6 KB / 394	16.1 KB / 594	22.6 KB / 1028	50.3 KB / 2984

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
driver	localhost:56679	24 min	2142	0	2142	80.2 GB / 18277096	45.3 MB / 2132216

Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	27456	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	6 s		11.8 MB (hadoop) / 8902	0,1 s	12.6 KB / 401	
2	27458	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	6 s		12.6 MB (hadoop) / 9142	80 ms	12.8 KB / 404	
1	27457	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	7 s		12.8 MB (hadoop) / 8948	0,5 s	12.8 KB / 404	
3	27459	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	7 s		11.2 MB (hadoop) / 8896	0,5 s	12.9 KB / 405	
6	27462	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	6 s		11.0 MB (hadoop) / 8882	52 ms	12.9 KB / 418	
5	27461	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	6 s		12.5 MB (hadoop) / 8936	83 ms	12.9 KB / 410	
4	27460	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	7 s		13.7 MB (hadoop) / 8986	0,5 s	12.7 KB / 399	

Spark UI
Application - Stage details

PySparkShell - Details for S

klang.cs.lth.se:4041/stages/stage/?id=94&attempt=0

Spark 1.4.0 | Jobs | Stages | Storage | Environment | Executors | application UI

Details for Stage 94 (Attempt 0)

Total Time Across All Tasks: 24 min
Input Size / Records: 80.2 GB / 18277096
Shuffle Write: 45.3 MB / 2132216

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 2142 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0,2 s	0,5 s	0,6 s	0,6 s	7 s
GC Time	0 ms	0 ms	0 ms	0 ms	2 s
Input Size / Records	2.3 MB / 470	12.4 MB / 5006	24.0 MB / 7074	39.2 MB / 9008	115.3 MB / 21294
Shuffle Write Size / Records	5.0 KB / 60	12.6 KB / 394	16.1 KB / 594	22.6 KB / 1028	50.3 KB / 2984

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
driver	localhost:56679	24 min	2142	0	2142	80.2 GB / 18277096	45.3 MB / 2132216

Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	27456	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	6 s		11.8 MB (hadoop) / 8902	0,1 s	12.6 KB / 401	
2	27458	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	6 s		12.6 MB (hadoop) / 9142	80 ms	12.8 KB / 404	
1	27457	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	7 s		12.8 MB (hadoop) / 8948	0,5 s	12.8 KB / 404	
3	27459	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	7 s		11.2 MB (hadoop) / 8896	0,5 s	12.9 KB / 405	
6	27462	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	6 s		11.0 MB (hadoop) / 8882	52 ms	12.9 KB / 418	
5	27461	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	6 s		12.5 MB (hadoop) / 8936	83 ms	12.9 KB / 410	
4	27460	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	7 s		13.7 MB (hadoop) / 8986	0,5 s	12.7 KB / 399	

Spark UI
Application -
Stage details

Details for Stage 94 (Attempt 0)

Total Time Across All Tasks: 24 min
Input Size / Records: 80.2 GB / 18277096
Shuffle Write: 45.3 MB / 2132216

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 2142 Completed Tasks

Metric	Min	25th percentile	Median
Duration	0,2 s	0,5 s	0,6 s
GC Time	0 ms	0 ms	0 ms
Input Size / Records	2.3 MB / 470	12.4 MB / 5006	24.0 MB / 7074
Shuffle Write Size / Records	5.0 KB / 60	12.6 KB / 394	16.1 KB / 594

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tas
driver	localhost:56679	24 min	2142	0	2142

Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC
0	27456	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/08/31 16:43:47	6 s	

Spark UI

Application - Stage details

	Median	75th percentile	Max
	0,6 s	0,6 s	7 s
	0 ms	0 ms	2 s
	24.0 MB / 7074	39.2 MB / 9008	115.3 MB / 21294
	16.1 KB / 594	22.6 KB / 1028	50.3 KB / 2984

asks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
	2142	80.2 GB / 18277096	45.3 MB / 2132216

ime	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
31 16:43:47	6 s		11.8 MB (hadoop) / 8902	0,1 s	12.6 KB / 401	

HDFS

- HDFS: Hadoop Distributed File System
- The primary command is **hadoop fs**
- Has a lot of basic commands
- On our cluster only: HDFS CLI ”hadoopfs”
  https://github.com/dstreev/hdfs-cli
  fork of
  https://github.com/ptgoetz/hdfs-cli

  - Interactive shell for HDFS operations

```
[semantica@semantica004 ~]$ hadoop fs
Usage: hadoop fs [generic options]
   [-appendToFile <localsrc> ... <dst>]
   [-cat [-ignoreCrc] <src> ...]
   [-checksum <src> ...]
   [-chgrp [-R] GROUP PATH...]
   [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
   [-chown [-R] [OWNER][:[GROUP]] PATH...]
   [-copyFromLocal [-f] [-p] [-l] <localsrc> ... <dst>]
   [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
   [-count [-q] [-h] <path> ...]
   [-cp [-f] [-p | -p[topax]] <src> ... <dst>]
   [-createSnapshot <snapshotDir> [<snapshotName>]]
   [-deleteSnapshot <snapshotDir> <snapshotName>]
   [-df [-h] [<path> ...]]
   [-du [-s] [-h] <path> ...]
   [-expunge]
   [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
   [-getfacl [-R] <path>]
   [-getfattr [-R] {-n name | -d} [-e en] <path>]
   [-getmerge [-nl] <src> <localdst>]
   [-help [cmd ...]]
   [-ls [-d] [-h] [-R] [<path> ...]]
   [-mkdir [-p] <path> ...]
   [-moveFromLocal <localsrc> ... <dst>]
   [-moveToLocal <src> <localdst>]
   [-mv <src> ... <dst>]
   [-put [-f] [-p] [-l] <localsrc> ... <dst>]
   [-renameSnapshot <snapshotDir> <oldName> <newName>]
   [-rm [-f] [-r|-R] [-skipTrash] <src> ...]
   [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
   [-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>]|[--set <acl_spec>
<path>]]
   [-setfattr {-n name [-v value] | -x name} <path>]
   [-setrep [-R] [-w] <rep> <path> ...]
   [-stat [format] <path> ...]
   [-tail [-f] <file>]
   [-test -[defsz] <path>]
   [-text [-ignoreCrc] <src> ...]
   [-touchz <path> ...]
```


HDFS

Common commands

- **-ls [path]**
  list directories, if no path is specified it assumes your home directory: /user/[username]

- **copyFromLocal [local] [hdfs]**
  copy data from the local filesystem to HDFS

- **-copyToLocal [hdfs] [local]**
  copy data from HDFS to the local filesystem

- **-cat [hdfs]**
  show the contents of files directly from HDFS

- **-mv, -cp, -df, -du**
  move, copy, disk free, disk used

- **-rm, -rmdir, -expunge**
  removal commands

Spark: Job submit

- Running on a cluster

- Append the following code to the header of your Python script

```
from pyspark import *
sc = SparkContext() #Remove master!
sqlContext = SQLContext(sc)
```

- To submit a job for execution run:

```
spark-submit [script].py
```

Spark-submit: Example

- **`spark-submit`** `pagerank.py`
- Data is located at: *hdfs:/share/person-network*
- Cluster manager: *semantica004.cs.lth.se*

Spark: Configuration

- `spark-submit --master [Master]`

  The cluster manager

`--driver-memory [MEM]`

The amount of heap memory to give the driver.

`--executor-memory [MEM]`

The amount of heap memory to give each worker

`--conf [PROP]=[VALUE]`

Special configuration values to give to Spark

Spark: Masters

- Three major cluster managers
  - **Standalone:**
    spark://[address] — o*ur cluster uses this*
    local[threads]
    local[*] — all logical cores on the local machine

    Note: On our cluster local[*] does not work due to a Python 2/3 conflict issue in the environment.
  - **Mesos:** mesos://[address]
  - **YARN:** yarn-client

Spark: Special configuration

- **spark.cores.max**
  The maximum number of cores to allocate. These are reserved at start up of the application

- **spark.driver.maxResultSize**
  the maximum amount of data that is allowed to be returned by e.g. a **collect()** action. Defaults to 1g.

- **Many others**
  https://spark.apache.org/docs/latest/configuration.html

Spark: Amazon EMR

- EMR: Elastic map-reduce
- Bootstrapping needed to get defaults configured correct such as number of cores, amount of memory
- Run this script: s3://support.elasticmapreduce/spark/install-spark -x

Spark-submit: Example

- **spark-submit**
  --master spark://semantica004.cs.lth.se:7077
  --driver-memory 2g
  --executor-memory 1g
  --conf spark.cores.max=44
  pagerank.py

- Data is located at: *hdfs:/share/person-network*

- Cluster manager: *semantica004.cs.lth.se*

Review of first hour

- Spark provides a wealth of web based user interfaces to peek into what is happening
- HDFS
- Job submission

Exercise one

- Run the Pagerank assignment on **a real cluster**
- If you are not ready with the Pagerank, then take Lab 1.3 and reformat it as a standalone script.
- Upload/Download to/from HDFS
- Submit jobs with spark-submit via CLI
- View and understand the statistics generated by Spark

Second hour

- **Goal:** Give you the tools to analyze a web server log
- **Parts**
  - Data structure
  - Dataframes
  - Spark SQL

The structure spectrum

Structured	**Semi-structured**	**Unstructured**
Relational Databases	HTML	Plain text
Parquet	XML	Generic media
Formatted Messages	JSON	

Tabular data

- Simple data format
- Common variants:
  - Comma Separated Values (CSV),
  - Tab Separated Values (TSV)

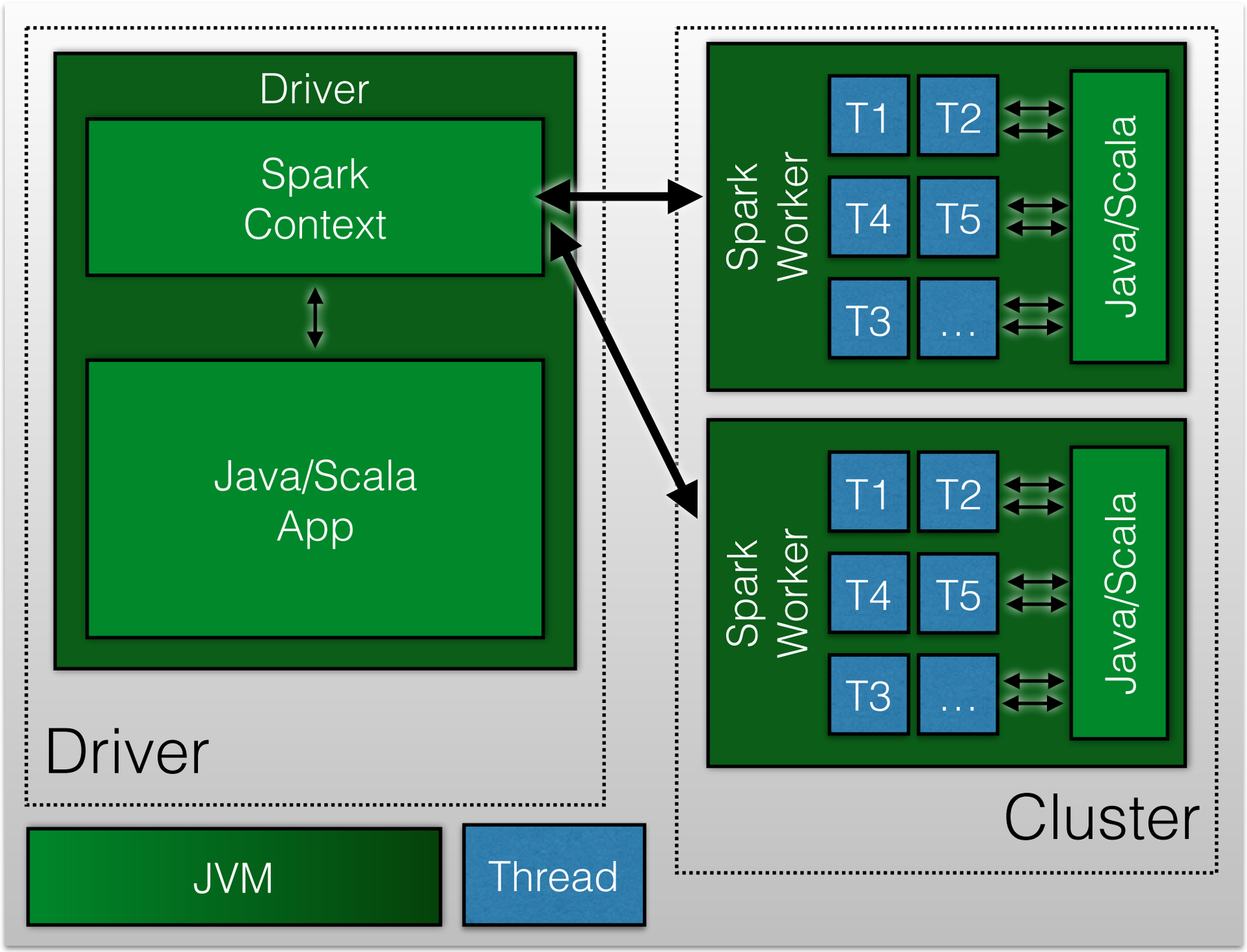
Tabular data

1	Genom	_	PR	PR	_	3	AA	_	_
2	skattereformen	_	NN	NN	_	1	PA	_	_
3	införs	_	VV	VV	_	0	ROOT	_	_
4	individuell	_	AJ	AJ	_	5	AT	_	_
5	beskattning	_	VN	VN	_	3	SS	_	_
6	(	_	IR	IR	_	5	IR	_	_
7	särbeskattning	_	VN	VN	_	5	AN	_	_
8	)	_	IR	IR	_	5	JR	_	_
9	av	_	PR	PR	_	5	ET	_	_
10	arbetsinkomster	_	NN	NN	_	9	PA	_	_
11	.	_	IP	IP	_	3	IP	_	_

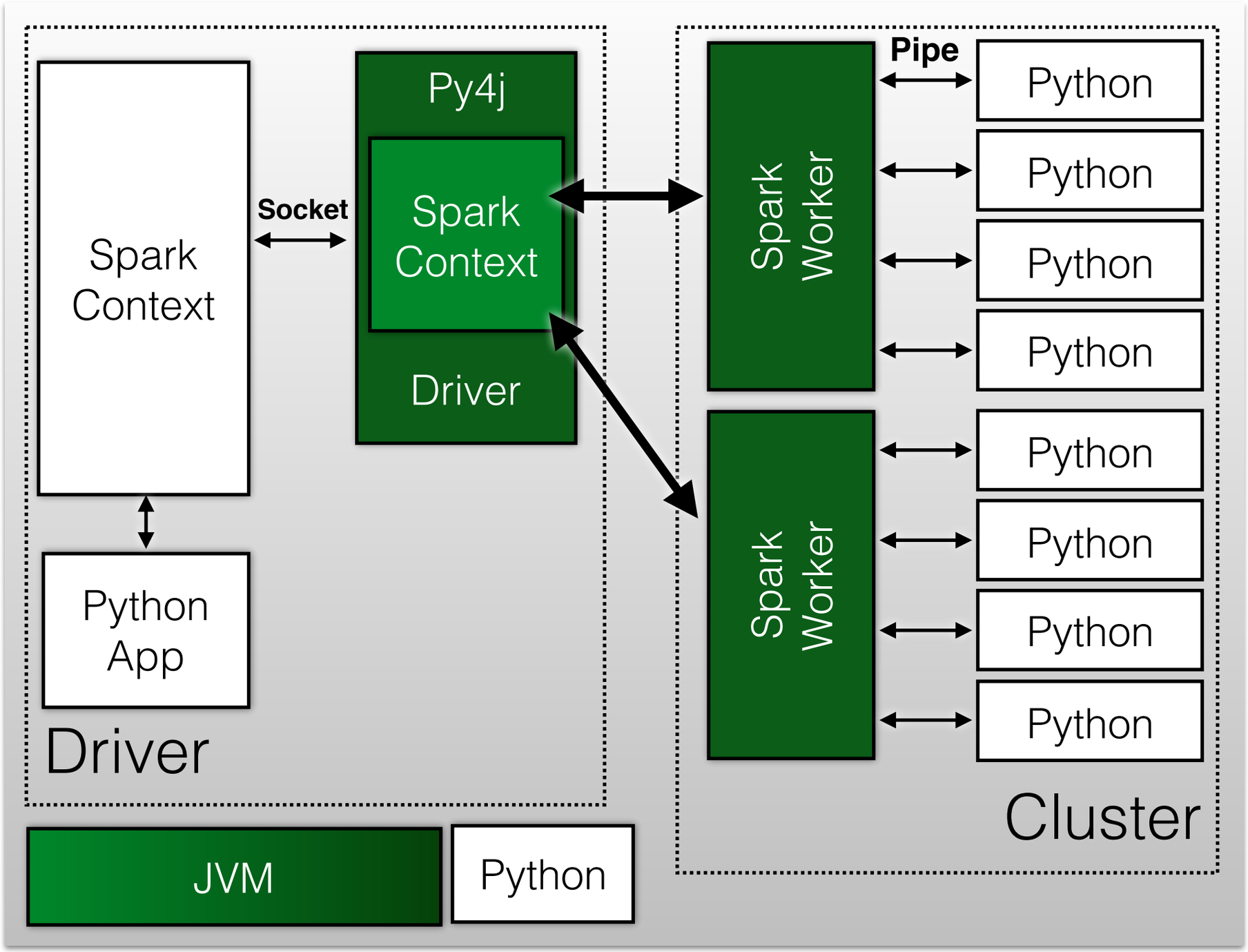
Spark SQL

- **Why another API?**
  - Performance

Spark: Java/Scala

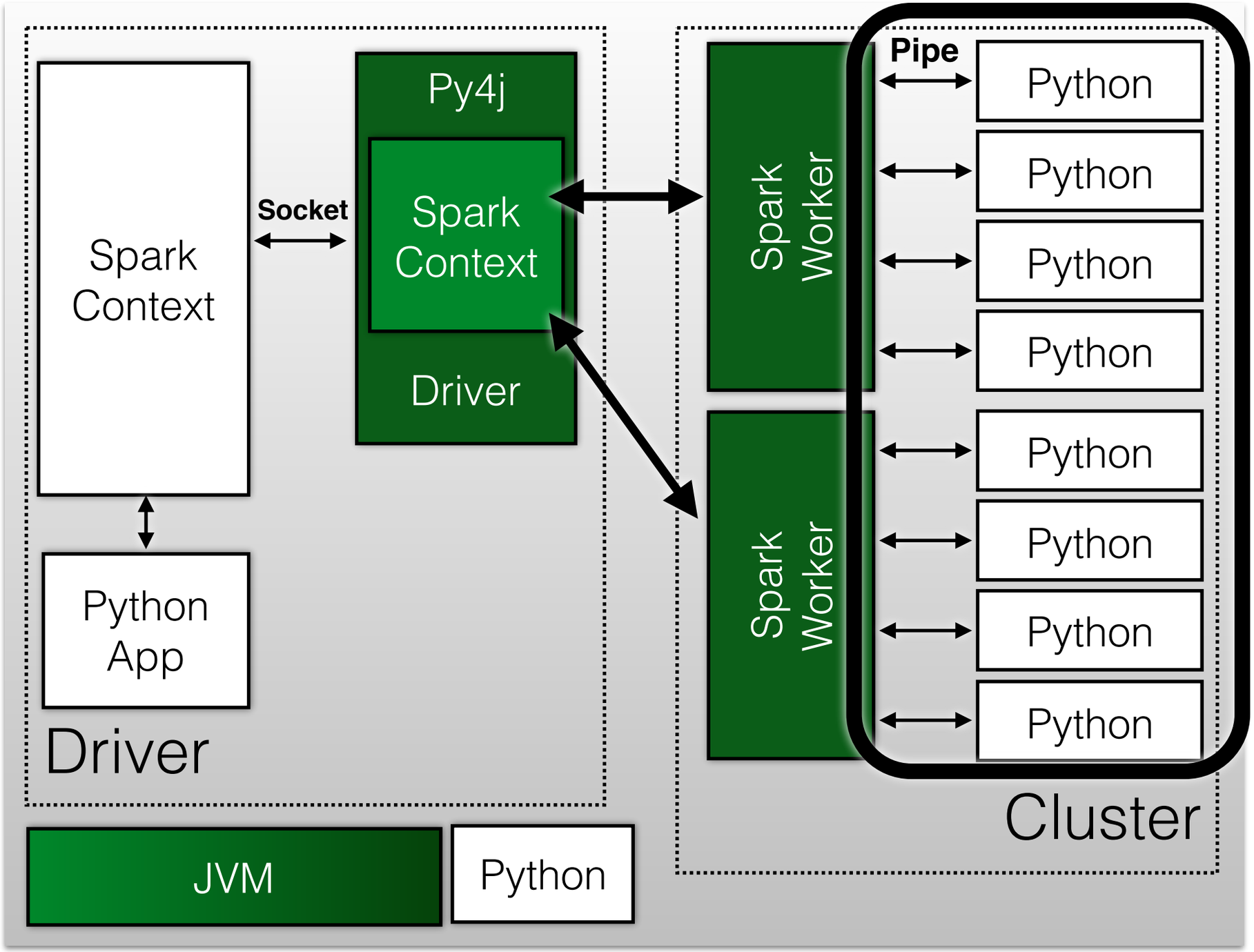


Pyspark


Spark Context
Socket
Py4j
Spark Context
Driver
Python App
Driver
Pipe
Spark Worker
Spark Worker
Python
Python
Python
Python
Python
Python
Python
Python
Cluster
JVM
Python

Pyspark

Python and pipe Overhead



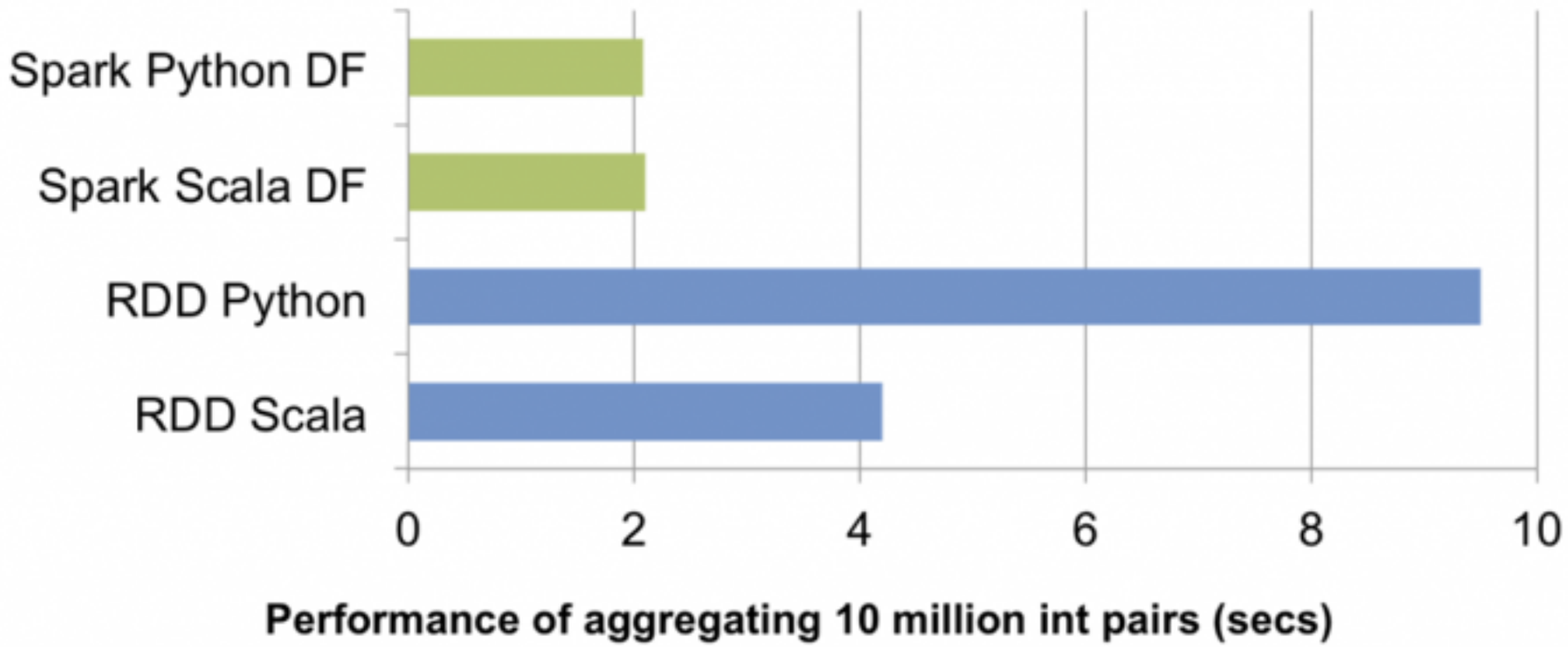
Pyspark: Performance

- Serialization/Deserialization overhead
- CPython / PyPy is officially supported.
  CPython still has most support since numpy is not yet supported in PyPy
- Python code performance issues if most time spent computing is not within native libraries
- **How can most of the headache of overhead be reduced?**

Pyspark SQL

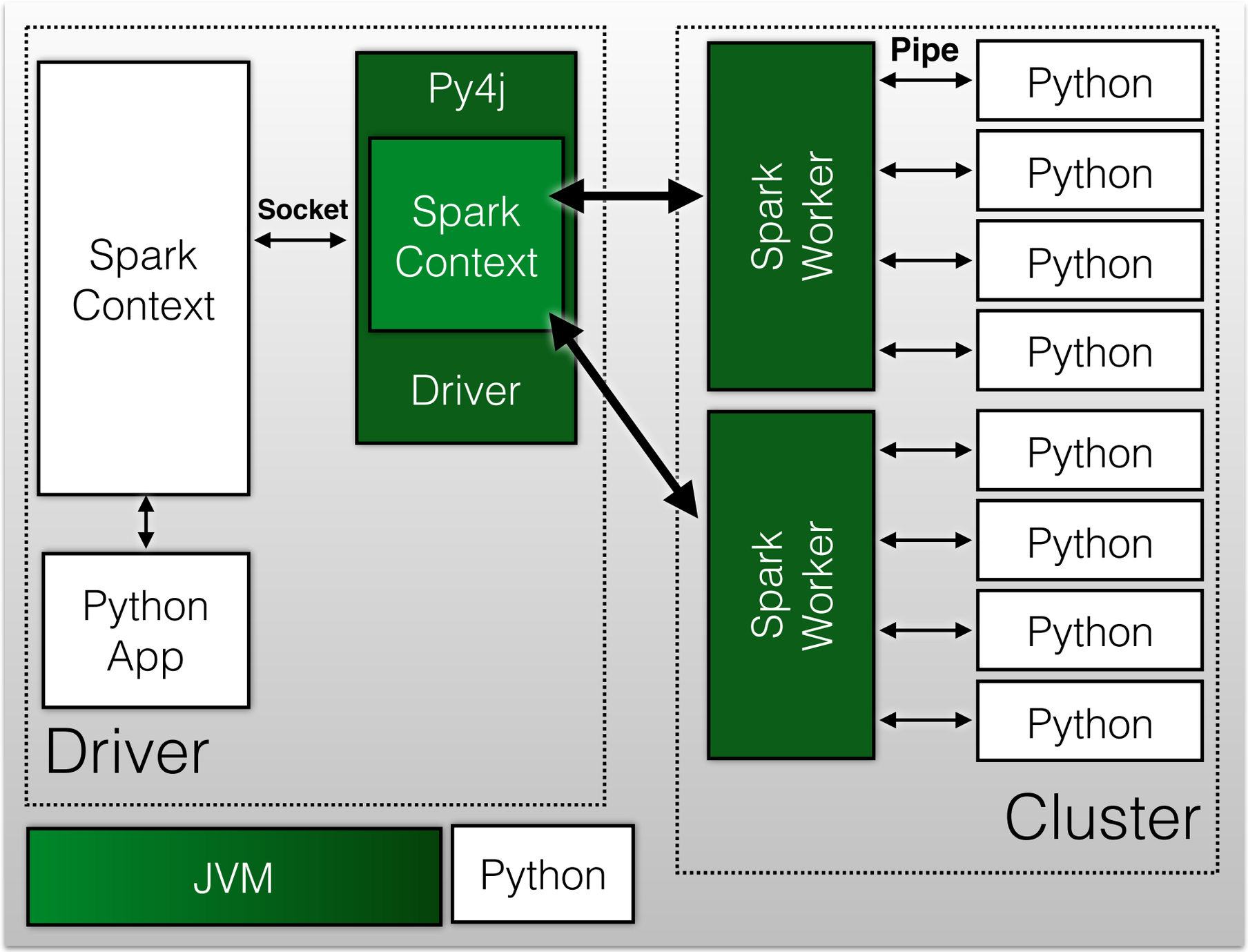
- Optimized data processing using optimized data structures and spark code, no extra overhead.

- New in Spark 1.3.0: Dataframes (SQL)

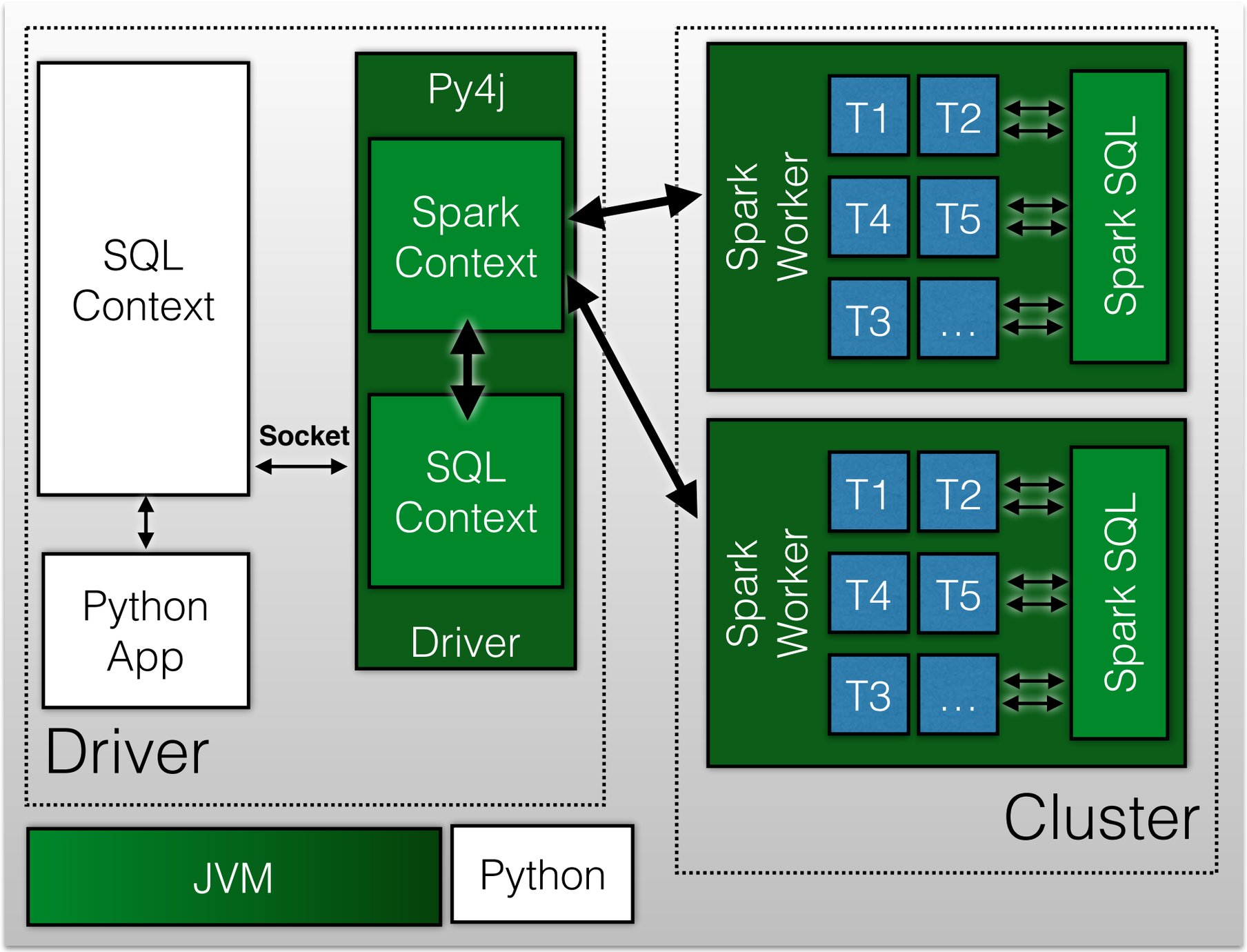


Source: https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html

Pyspark

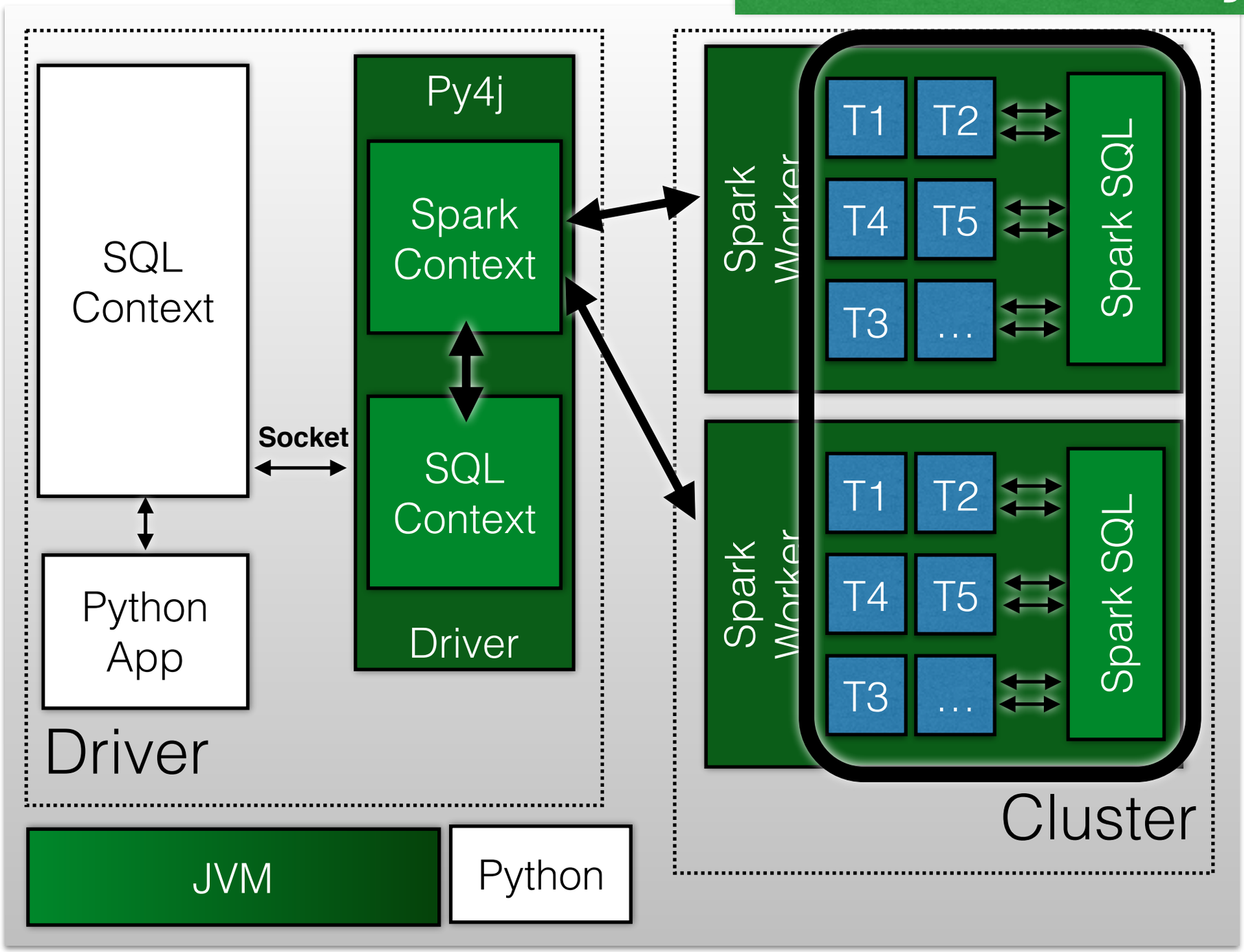

Spark Context
Socket
Py4j
Spark Context
Driver
Python App
Driver
Pipe
Spark Worker
Spark Worker
Python
Python
Python
Python
Python
Python
Python
Python
Cluster
JVM
Python

Pyspark SQL


Py4j
Spark Context
SQL Context
Socket
SQL Context
Python App
Driver
Driver
JVM
Python
Spark Worker
T1
T2
T4
T5
T3
…
Spark SQL
Spark Worker
T1
T2
T4
T5
T3
…
Spark SQL
Cluster

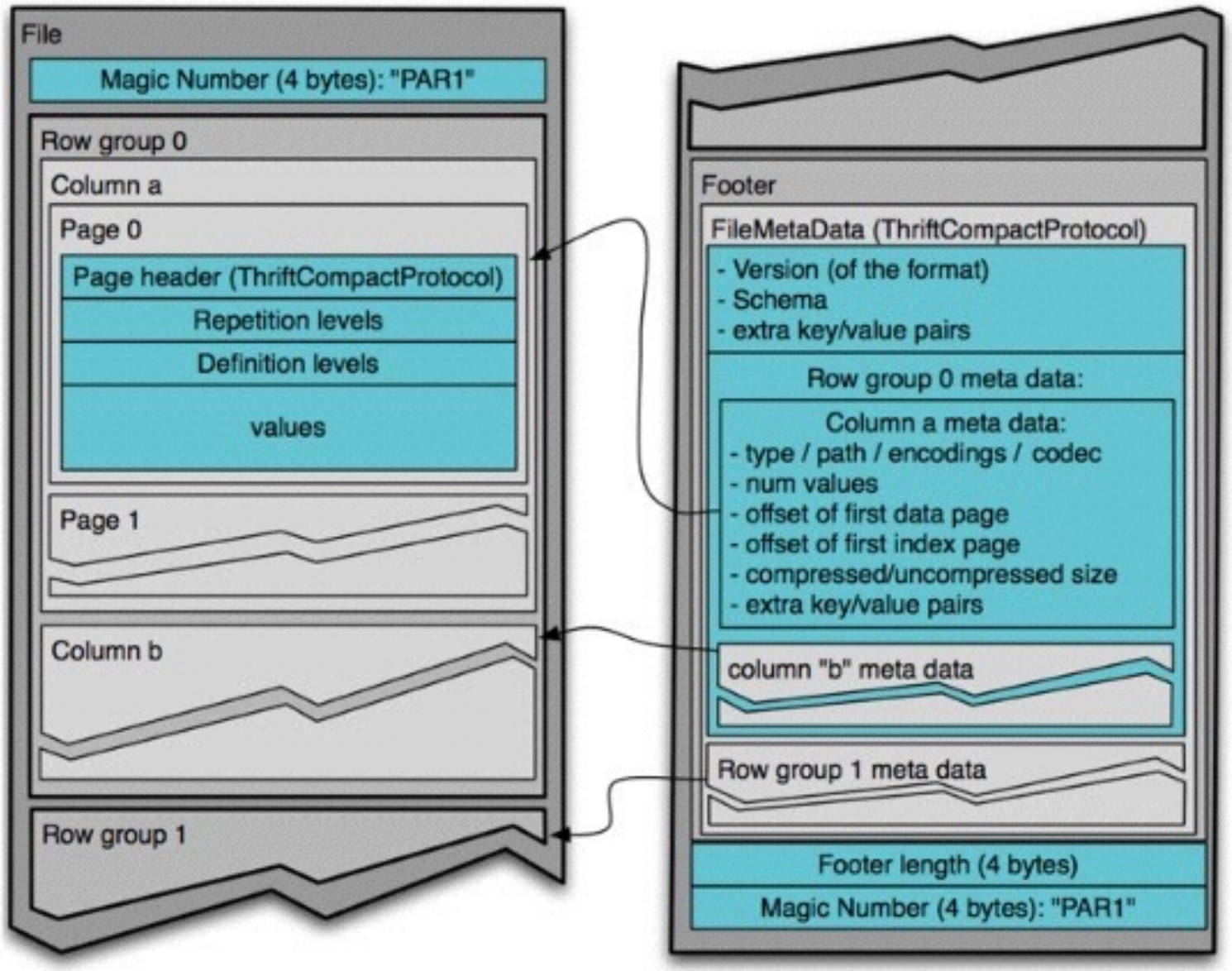
Pyspark

No overhead
+ Native Spark SQL
Code only.



Parquet

- Default storage format for Spark SQL
- Optimized data storage
- Read only what you need, skip other columns if not needed.



Source: http://www.slideshare.net/julienledem/parquet-stratany-hadoopworld2013

SQL Context

- Created from SparkContext

```
>>> from pyspark import SparkContext
>>> from pyspark.sql import SQLContext
>>> sc = SparkContext()
>>> sqlContext = SQLContext(sc)
```

Pandas DataFrame

- Pandas is a library with easy to use data structures for data analysis. The API for Sparks Dataframes is inspired by the Pandas API.

- Spark provides a way to convert to and from pandas Dataframe via `toPandas()` and `sqlContext.createDataFrame(pandasDataFrame)`

- One powerful and easy way to visualize data is: `dataframe.toPandas().plot()`

Dataframe: I/O

- **sqlContext.read.[format]**

```
>>> sqlContext.read.parquet(path)
>>> sqlContext.read.json(path, [schema])
>>> sqlContext.read.jdbc(url, table)
>>> sqlContext.read.load(path, [format])
```

Source: https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameReader

- **sqlContext.write.[format]**

```
>>> sqlContext.write.parquet(path)
>>> sqlContext.write.json(path, [mode])
>>> sqlContext.write.jdbc(url, table, [mode])
>>> sqlContext.write.save(path, [format], [mode])
```

Source: https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameWriter

Dataframe and RDDs

- **Create from RDD of tuples**

```
>>> rdd = sc.parallelize([(”a”, 1), (”b”, 2), (”c”, 3)])

>>> df = sqlContext.createDataFrame(rdd, [”name”, ”id”])
```

```
>>> df.show()
+----+--+
|name|id|
+----+--+
|   a| 1|
|   b| 2|
|   c| 3|
+----+--+
```

Dataframe and RDDs

- **Create from RDD of Row**

```
>>> from pyspark.sql import Row
>>> ExampleRow = Row(”name”, ”id”) #Specifies the column names

>>> rdd = sc.parallelize([
    ExampleRow(”a”, 1), ExampleRow(”b”, 2), ExampleRow(”c”, 3) ])

>>> df = sqlContext.createDataFrame(rdd)

>>> df.show()
+----+--+
|name|id|
+----+--+
|   a| 1|
|   b| 2|
|   c| 3|
+----+--+
```

Dataframe and RDDs

- **Create from RDD of Row (subtle behavior difference)**

```
>>> from pyspark.sql import Row

>>> rdd = sc.parallelize([
   Row(name=”a”, id=1), Row(name=”b”, id=2), Row(name=”c”, id=3) ])
```

```
>>> df = sqlContext.createDataFrame(rdd)

>>> df.show()
+--+----+
|id|name|
+--+----+
| 1|   a|
| 2|   b|
| 3|   c|
+--+----+
```

Notice that the order has changed!

Row() constructor	➔	Keyname sorted order
ExampleRow = Row(col1, col2, …)	➔	Index order
sqlContext.read.*	➔	Index order

Dataframe: Schema

- All dataframes has a schema which defines its structure.

```
>>> df.printSchema()

root
|-- name: string (nullable = false)
|-- id: integer (nullable = false)
```

- Per default Spark will try to infer it from the data.
  - When using JSON, it has to go through the data and infer the schema, this can be avoided by specifying a Schema directly.

Dataframe: Schema

Specify schemas programmatically (**rdd** already defined)

```
>>> from pyspark.sql.types import *

>>> schema = StructType(
     StructField(”name”, StringType(), False),
     StructField(”id”, IntegerType(), False)
)

>>> df = sqlContext.createDataFrame(rdd, schema)

>>> df.printSchema()

root
 |-- name: string (nullable = false)
 |-- id: integer (nullable = false)
```

Dataframe: Types

- Common types:
  - `StringType()`
  - `IntegerType(), LongType()`
  - `BooleanType()`
  - `ArrayType([internal type])`
  - `FloatType(), DoubleType()`
  - `StructField(name, [internal type], [Nullable])`

- Many more: https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.types

Dataframe and RDDs

- Dataframe to RDD

```
>>> type(df.rdd)
pyspark.rdd.PipelinedRDD
```

- The content is Row objects, which can be indexed by name directly. Common use is unpacking:

```
>>> df.rdd.map(lambda row: row.name).collect()
[u'a', u'b', u’c']
```

- So common that Spark provides alias for map: `df.map` returns an RDD

Pyspark SQL

- Pyspark SQL == Query API on top of Dataframes
- Two sets of APIs:
  - Traditional SQL: construct a string and pass along
  - Dataframe API (DF API), a **domain specific language** (DSL) for querying. Implemented by using method chaining, i.e. fluent interfaces.

Pyspark SQL: Traditional

```
>>> sqlContext.registerDataFrameAsTable(df,
”tblData”)

>>> resultDf = sqlContext.sql(
      ”SELECT name, id FROM tblData”)
```

```
>>> resultDf.show()
+----+--+
|name|id|
+----+--+
|   a| 1|
|   b| 2|
|   c| 3|
+----+--+
```

Pyspark SQL: DataFrame API

```
>>> resultDf = df.select(df.name, df.id)

>>> resultDf.show()
+----+--+
|name|id|
+----+--+
|   a| 1|
|   b| 2|
|   c| 3|
+----+--+
```


Pyspark SQL: Comparison

```
>>> resultDf = (df.select(df.name, df.id)
                  .where(df.id == 2))

>>> resultDf = sqlContext.sql(
”SELECT name, id FROM tblData WHERE id = 2”)
```

Pyspark SQL: DF vs Trad.

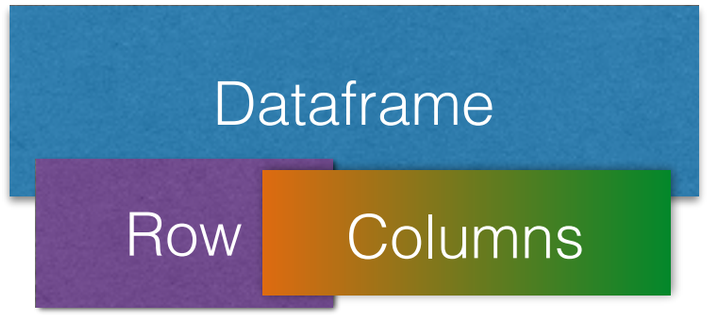
- DataFrame API performance is most of the time equivalent to traditional SQL.

- In older versions of Spark (pre 1.5.0): many useful functions were not available in the DataFrame API.

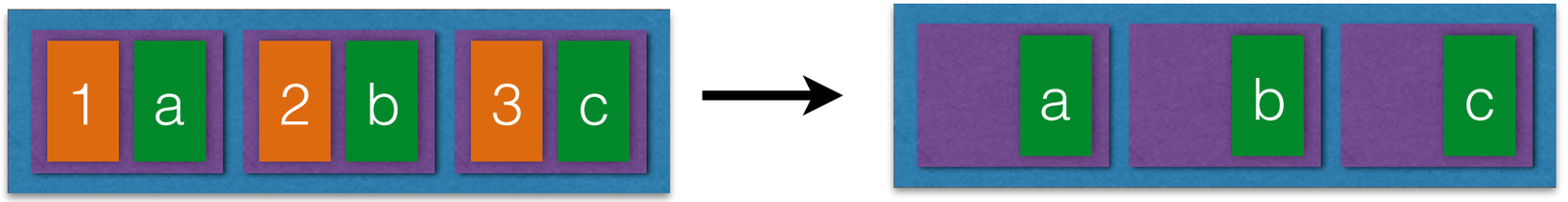
- **Which one is the winner?**
  Tie. It depends on your background. Some queries are shorter to express in traditional, others with the DF API.

- **My advice**
  Choose the one that generates the most understandable and simple code

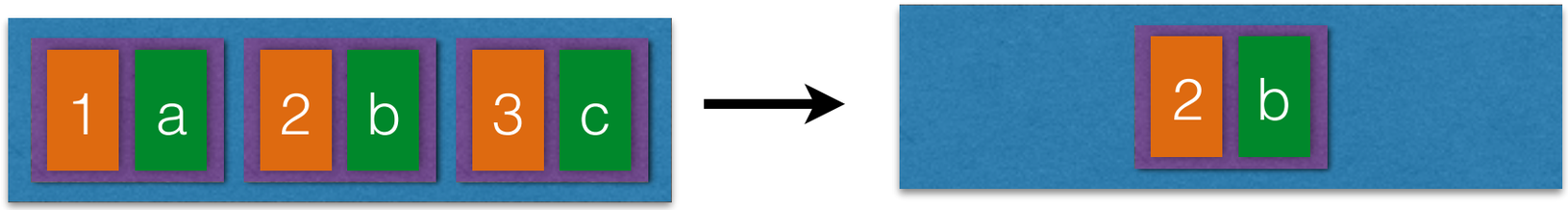
SQL operations



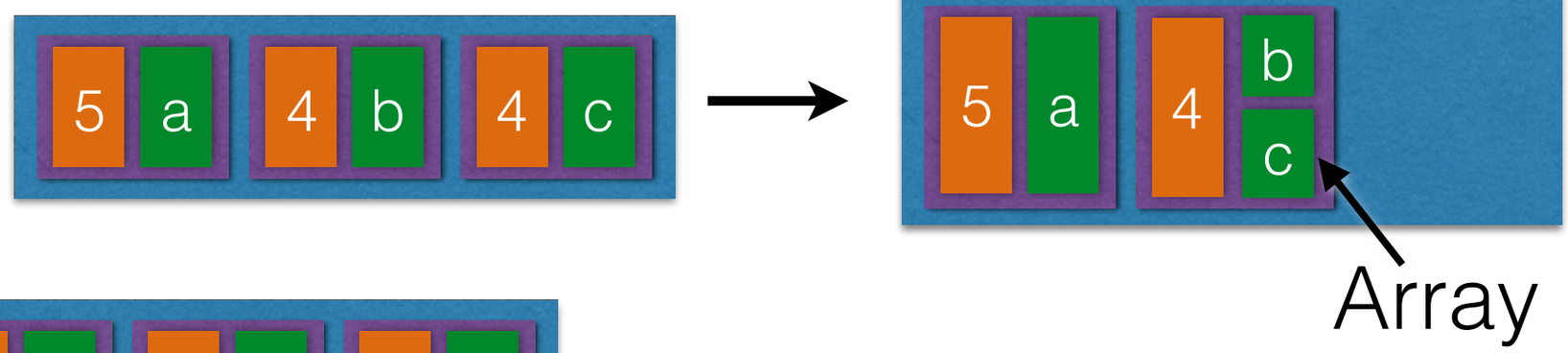
- Projection



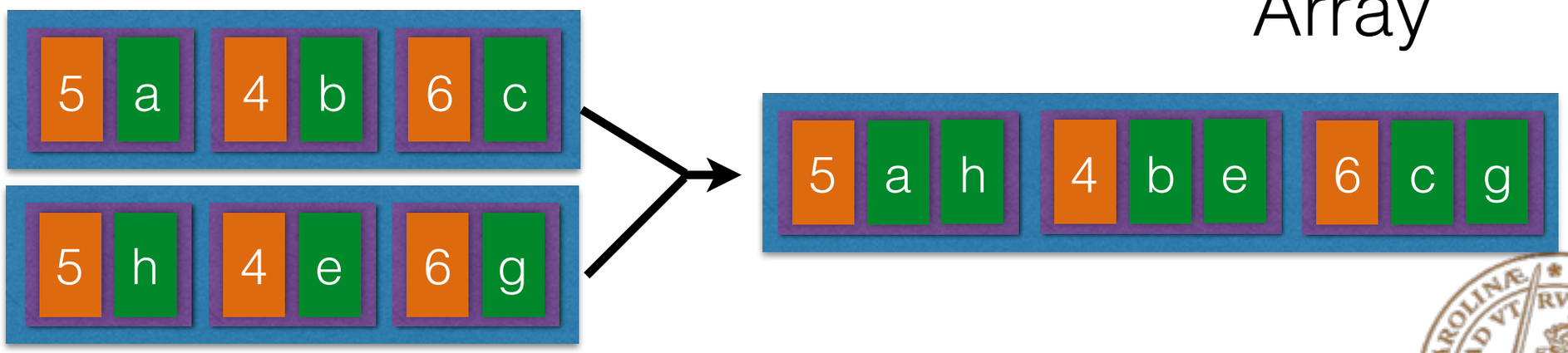
- Filtering



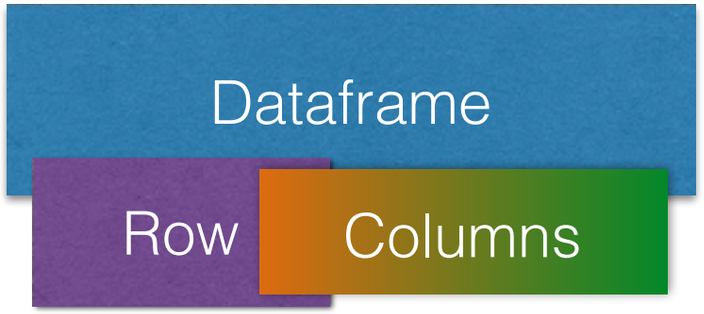
- Grouping



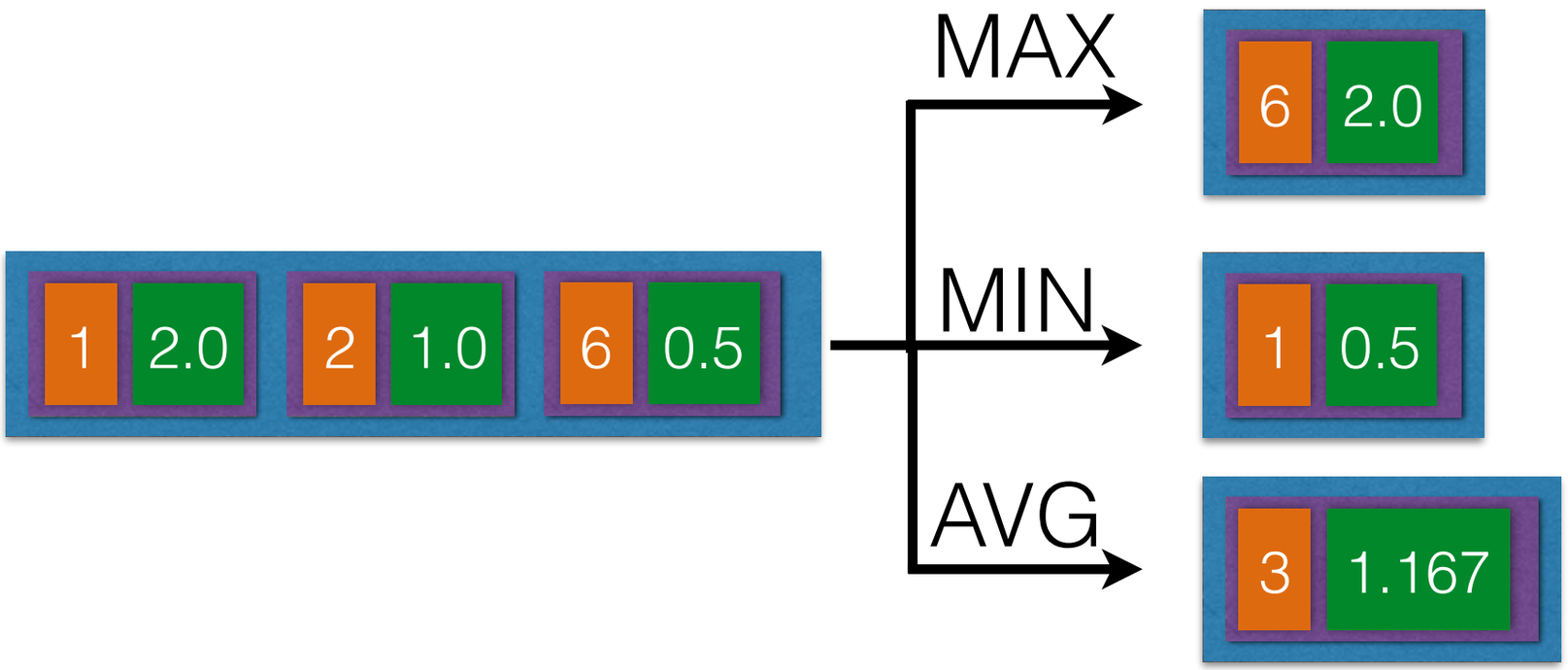
- Joins



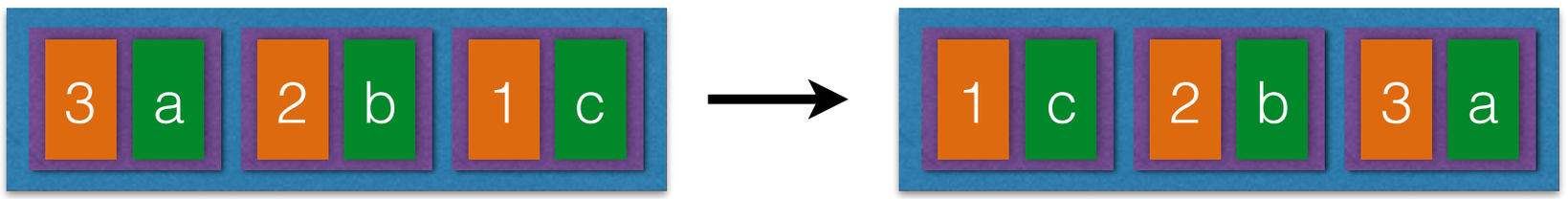
SQL operations



- Aggregation



- Ordering



Pyspark SQL: Projection

Pure selection of columns

```
>>> res = sqlContext.sql(”SELECT name FROM tblData”)
```

```
>>> from pyspark.sql.functions import column
>>> res = df.select(df.name)
>>> res = df.select(column(”name”))
>>> res = df.select(df[”name”])
```

All equivalent

```
>>> res.show()
+----+
|name|
+----+
|   a|
|   b|
|   c|
+----+
```

Pyspark: Projection

Renaming + math operation on column

```
>>> res = sqlContext.sql(
”SELECT name, id + 1 AS idAlias FROM tblData”)
```

```
>>> res = df.select(df.name, (df.id + 1).alias(”idAlias”))
```

```
>>> res.show()
+----+-------+
|name|idAlias|
+----+-------+
|   a|      2|
|   b|      3|
|   c|      4|
+----+-------+
```

Spark SQL: Filtering

```
>>> res = sqlContext.sql(
”SELECT * FROM tblGameData
WHERE score BETWEEN 1000 AND 2000”)
```

```
>>> res =
gameDf.where(gameDf.score.between(1000,2000))
```

```
>>> res.show()
+--+----+-----+
|id|name|score|
+--+----+-----+
| 2| Sue| 1401|
+--+----+-----+
```

tblGameData / gameDf

Id	Name	Score
1	Alfred	320
2	Sue	1401
3	Thomas	4302
4	Abby	5102

Spark SQL: Filtering

```
>>> res = sqlContext.sql(
”SELECT * FROM tblGameData WHERE id = 4)
```

```
>>> res = gameDf.where(gameDf.id == 4)
```

```
>>> res.show()
+--+----+-----+
|id|name|score|
+--+----+-----+
| 4|Abby| 5102|
+--+----+-----+
```

tblGameData / gameDf

Id	Name	Score
1	Alfred	320
2	Sue	1401
3	Thomas	4302
4	Abby	5102

Spark SQL: Filtering

```
>>> res = sqlContext.sql(
”SELECT * FROM tblGameData WHERE score > 3000”)
```

```
>>> res = gameDf.where(gameDf.score > 3000)
```

```
>>> res.show()
+--+------+-----+
|id|  name|score|
+--+------+-----+
| 3|Thomas| 4302|
| 4|  Abby| 5102|
+--+------+-----+
```

tblGameData / gameDf

Id	Name	Score
1	Alfred	320
2	Sue	1401
3	Thomas	4302
4	Abby	5102

Spark SQL: Filtering

```
>>> res = sqlContext.sql("SELECT * FROM tblGameData
WHERE name > 'Dennis' AND score > 4000")
```

```
>>> res = gameDf.where(
  (gameDf.name > "Dennis") & (gameDf.score > 4000) )
```

& = AND, | = OR, ~ = NOT

```
>>> res.show()
+--+------+-----+
|id|  name|score|
+--+------+-----+
| 3|Thomas| 4302|
+--+------+-----+
```

tblGameData / gameDf

Id	Name	Score
1	Alfred	320
2	Sue	1401
3	Thomas	4302
4	Abby	5102

Spark SQL: Aggregation

- A few aggregation functions
  - **MIN** = Minimum of the group
  - **AVG** = Average value of the group
  - **MAX** = Maximum of the group
  - **SUM** = Sum of the group
  - **COUNT**(*), **COUNT**(col) = Number in all, col group
  - **COUNT**(**DISTINCT** *), **COUNT**(**DISTINCT** col) gives the number of unique in the group

Spark SQL: Aggregation

Toy data for the examples

tblGameData / gameDf

Id	Name	Score	Team	Handicap
1	Alfred	320	Red	5
2	Sue	1401	Blue	4
3	Thomas	4302	Red	3
4	Abby	5102	Blue	2
5	Anonymous	15680	Green	1
6	Rocker	7820	Blue	2

Spark SQL: Aggregation

Traditional

```
>>> res = sqlContext.sql("""
SELECT
  team,
  MIN(score) AS min,
  AVG(score) AS avg,
  MAX(score) AS max,
  SUM(score) AS sum,
  COUNT(*) AS count,
  COUNT(DISTINCT handicap) AS numUnique
FROM tblGameData
GROUP BY team""")

>>> res.show()
+-----+-----+------------------+-----+-----+-----+---------+
| team|  min|               avg|  max|  sum|count|numUnique|
+-----+-----+------------------+-----+-----+-----+---------+
| Blue| 1401|4774.333333333333| 7820|14323|    3|        2|
|Green|15680|           15680.0|15680|15680|    1|        1|
|  Red|  320|            2311.0| 4302| 4622|    2|        2|
+-----+-----+------------------+-----+-----+-----+---------+
```

Spark SQL: Aggregation

Dataframe API

```
>>> from pyspark.sql import functions as F
>>> res = (
gameDf.select(gameDf.team, gameDf.score, gameDf.handicap)
      .groupBy("team")
      .agg(F.min(gameDf.score).alias("min"),
           F.avg(gameDf.score).alias("avg"),
           F.max(gameDf.score).alias("max"),
           F.sum(gameDf.score).alias(”sum”),
           F.count("*")       .alias(”count"),
           F.countDistinct(”handicap”).alias("numUnique")
)

>>> res.show()
+-----+-----+------------------+-----+-----+-----+---------+
| team|  min|               avg|  max|  sum|count|numUnique|
+-----+-----+------------------+-----+-----+-----+---------+
| Blue| 1401|4774.333333333333| 7820|14323|    3|        2|
|Green|15680|           15680.0|15680|15680|    1|        1|
|  Red|  320|            2311.0| 4302| 4622|    2|        2|
+-----+-----+------------------+-----+-----+-----+---------+
```

Spark SQL: Ordering

Traditional

- ```
  res = sqlContext.sql(
  """SELECT name, score, handicap
      FROM tblGameData
      ORDER BY name DESC""")
  ```

- ```
  >>> res.show()
  +---------+-----+--------+
  |     name|score|handicap|
  +---------+-----+--------+
  |   Thomas| 4302|       3|
  |      Sue| 1401|       4|
  |   Rocker| 7820|       2|
  |Anonymous|15680|       1|
  |   Alfred|  320|       5|
  |     Abby| 5102|       2|
  +---------+-----+--------+
  ```

Spark SQL: Ordering

Dataframe API

```
res = (
    gameDf.select(
                gameDf.name, gameDf.score, gameDf.handicap
    ).orderBy(gameDf.name.desc())
)
```

```
>>> res.show()
+---------+-----+--------+
|     name|score|handicap|
+---------+-----+--------+
|   Thomas| 4302|       3|
|      Sue| 1401|       4|
|   Rocker| 7820|       2|
|Anonymous|15680|       1|
|   Alfred|  320|       5|
|     Abby| 5102|       2|
+---------+-----+--------+
```

Spark SQL: Ordering

Dataframe API

```
res = (
    gameDf.select(
                gameDf.name.alias(”n”), gameDf.score, gameDf.handicap
    ).orderBy(desc(”n”))
)
```

```
>>> res.show()
+---------+-----+--------+
|        n|score|handicap|
+---------+-----+--------+
|   Thomas| 4302|       3|
|      Sue| 1401|       4|
|   Rocker| 7820|       2|
|Anonymous|15680|       1|
|   Alfred|  320|       5|
|     Abby| 5102|       2|
+---------+-----+--------+
```

Spark SQL: Functions

- Traditional SQL contains a wealth of functions.
- Functions supported in Hive are mostly supported in Spark, but not all.
- Spark 1.5.0 implemented many functions in the Dataframe API previously only available in traditional SQL.

Spark SQL: Functions

- A few useful ones (traditional SQL)
  - **PRINTF** - string formatting, e.g
    PRINTF(”%04d-%02d-%02d”, 2015, 9, 14) => 2015-09-14
  - **INSTR** - get the location of string in a string
    INSTR(”text”, ”t”) => 1, INSTR(”text”, ”o”) => 0
  - **LENGTH** - the length of a string, **SIZE** - size of an array.
    Computes the length of a unicode string correctly!
  - **EXPLODE** - a way to explode an array inside a column into multiple rows.
  - **COLLECT_LIST**, **COLLECT_SET** - a way of getting the contents of a group into a list.

    Beware of possible memory limitations, the list/set have fit in memory.

Spark SQL: UDFs

- User Defined Functions (UDF)

- Powerful way of extending SQL capabilities with your own functions and missing functionality

Spark SQL: UDFs

Traditional

- ```
  >>> sqlContext.registerFunction("plusOne", lambda i: i + 1,
  IntegerType())
  ```

- ```
  >>> sqlContext.sql("""
     SELECT plusOne(id) AS plusOne_id
     FROM tblGameData
  """).show()
  ```

```
+----------+
|plusOne_id|
+----------+
|         2|
|         3|
|         4|
|         5|
|         6|
|         7|
+----------+
```

Spark SQL: UDFs

Dataframe API

- ```
  >>> from pyspark.sql.functions import udf
  >>> plusOne = udf(lambda i: i + 1, IntegerType())
  ```

- ```
  >>> gameDf.select(
      plusOne("id").alias("plusOne_id")
  ).show()
  ```

```
+----------+
|plusOne_id|
+----------+
|         2|
|         3|
|         4|
|         5|
|         6|
|         7|
+----------+
```

Apache log

- Web server log
- Contains a log of requests made to the server
- Configurable format, two common formats:
  - Combined format
  - Common format
- One request per line, simple text format

Apache log

```
127.0.0.1 - frank [10/Oct/2000:13:55:36
-0700] "GET /apache_pb.gif HTTP/1.0" 200
2326 "http://www.example.com/start.html"
"Mozilla/4.08 [en] (Win98; I ;Nav)”
```

Apache log

`127.0.0.1`	Remote Host
`-`	RFC 1413 User Identity
`frank`	Authenticated User
`[10/Oct/2000:13:55:36 -0700]`	Date/Time
`"GET /apache_pb.gif HTTP/1.0"`	Request
`200`	Response Code
`2326`	Data sent
`"http://www.example.com/start.html"`	Referrer
`"Mozilla/4.08 [en] (Win98; I ;Nav)"`	User Agent

Apache log: Fileadmin

- In the exercise, you will process logs from the fileadmin.cs.lth.se web-server
- Contains data between 2011 and 2015
- 19 million requests
- The data has been initially filtered and converted into a columnar format (parquet)

Apache log: Fileadmin

1. **Data acquisition:** Contacted the administrator for Fileadmin and got the raw log

2. **Data preparation:** Some issues with the data was quickly found

   a. The log filename was prefixed in the beginning of each line

   b. Two lines were corrupted, possibly because of a hardware failure

   c. Some requests were very frequent and used by internal server scripts. We filtered out them as they polluted the data.

Apache log: Fileadmin

3. Regex in multiple stages to extract information

    1. Regex to extract the major parts

    2. Regex to extract date information
    `[10/Oct/2000:13:55:36 -0700]`

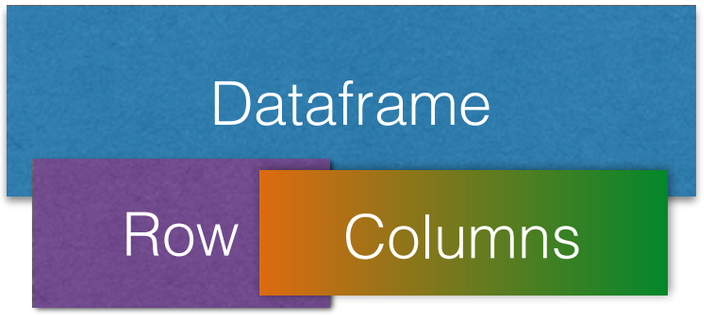
    3. Regex to extract method, resource and http version
    `"GET /apache_pb.gif HTTP/1.0” ->`
    `Row(method=”GET”, resource=”/apache_pb.gif”, version=”1.0”)`

4. Save to parquet format for the exercises

Exercise two

- Experiment with Spark SQL
  - aggregation, ordering, filtering, grouping
- Fileadmin log analysis and visualization

SQL operations

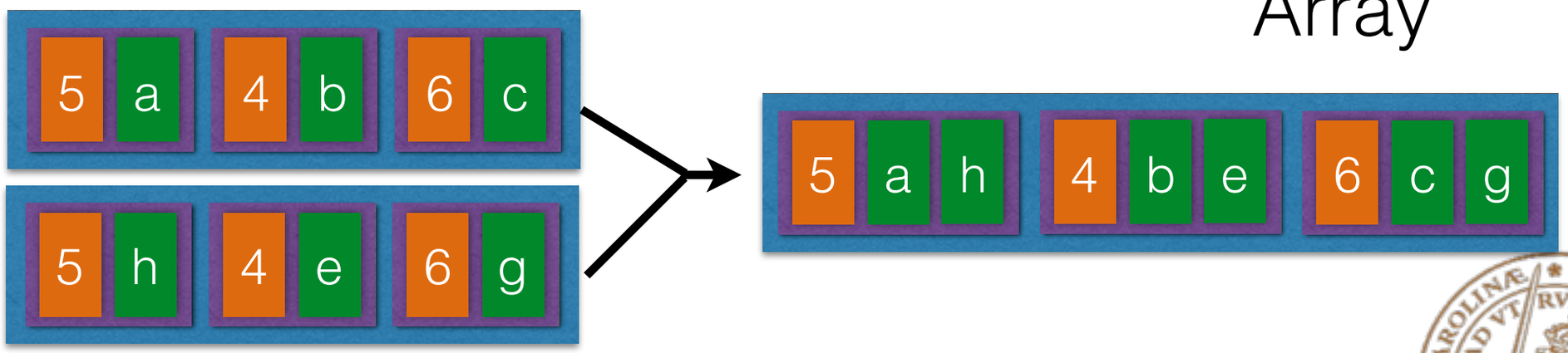


- Projection

- Filtering

- Grouping

- Joins



Spark SQL: Joins

tblGameData / gameDf

Id	Name	Score	Team	Handicap
1	Alfred	320	Red	5
2	Sue	1401	Blue	4
3	Thomas	4302	Red	3
4	Abby	5102	Blue	2
5	Anonymous	15680	Green	1
6	Rocker	7820	Blue	2

tblGameInfo /gameInfoDf

GameId	Character	UserId
1	Dragon	1
2	Mouse	5
3	Wizard	3
4	Lizard	4

Spark SQL: Inner join

Traditional

```
>>> sqlContext.sql("""
SELECT Id, Name, Character
FROM tblGameData
INNER JOIN tblGameInfo
ON tblGameData.Id = tblGameInfo.UserId
""").show()

+--+---------+---------+
|Id|     Name|Character|
+--+---------+---------+
| 1|   Alfred|   Dragon|
| 3|   Thomas|   Wizard|
| 4|     Abby|   Lizard|
| 5|Anonymous|    Mouse|
+--+---------+---------+
```

Spark SQL: Inner join

Dataframe API

```
>>> from pyspark.sql.functions import column as col
>>> gameDf.join(
      gameInfoDf,
      gameDf.id == gameInfoDf.userId,
      "inner"
    ).select(
      col("Id"), col("Name"), col("Character")
    ).show()

+--+---------+---------+
|Id|     Name|Character|
+--+---------+---------+
| 1|   Alfred|   Dragon|
| 3|   Thomas|   Wizard|
| 4|     Abby|   Lizard|
| 5|Anonymous|    Mouse|
+--+---------+---------+
```

Spark SQL: Left join

Traditional

- ```
  >>> sqlContext.sql("""
  SELECT Id, Name, Character
  FROM tblGameData
  LEFT JOIN tblGameInfo
  ON tblGameData.Id = tblGameInfo.UserId
  """).show()
  ```

```
+--+---------+---------+
|Id|     Name|Character|
+--+---------+---------+
| 1|   Alfred|   Dragon|
| 2|      Sue|     null|
| 3|   Thomas|   Wizard|
| 4|     Abby|   Lizard|
| 5|Anonymous|    Mouse|
| 6|   Rocker|     null|
+--+---------+---------+
```


```

Spark SQL: Left join

Dataframe API

-
```
>>> from pyspark.sql.functions import column as col
>>> gameDf.join(
      gameInfoDf,
      gameDf.id == gameInfoDf.userId,
      "left"
     ).select(
        col("Id"), col("Name"), col("Character")
     ).show()
```

```
+--+---------+---------+
|Id|     Name|Character|
+--+---------+---------+
| 1|   Alfred|   Dragon|
| 2|      Sue|     null|
| 3|   Thomas|   Wizard|
| 4|     Abby|   Lizard|
| 5|Anonymous|    Mouse|
| 6|   Rocker|     null|
+--+---------+---------+
```

Partitions

- change the number of partitions of records via a shuffle

```
>>> rdd.repartition(numPartitions)
```

- change the number of partitions with optional shuffling

```
>>> rdd.coalesce(numPartitions,
shuffle=False)
```

Partitions

- The minimal unit of work per task, usually one per core in sequence.

- **Optimal number of partitions?**
  It is a balance between parallelism and throughput.

- **Optimal:** The number of partitions that gives the lowest job execution time overall

  ➡ Use the statistics that have is available in Spark Web UI to decide.

Partitions

- Small amount of partitions:
  (low overhead, high throughput per core, low parallelism)

- Huge numbers of partitions:
  (high overhead, low throughput per core, high parallelism)

- Optimal number of partitions:
  (low overhead, high throughput per core, optimal parallelism)

  Does not necessarily mean that all cores should be used! (depends on job size)

Spark: Files

- Simple means of adding files that will be distributed to each node.
- A broadcast of files to each node, similar to a broadcast of a value but instead it is files.
- These reside in temporary directories in the local filesystem of each cluster node.
- Driver: `sc.addFiles(path)` — schedule file for distribution when needed.
- Task: `SparkFiles.get(filename)` — get the local filesystem path

Exercise three

- Experiment with Spark SQL
  - Joining
  - Wikipedia
- Dataframe performance evaluation

This weeks assignment

- Multilingual paragraph matching
- The task is to investigate if we can match the first paragraph in 1000 wikipedia person articles across languages.

Backup slides

Spark: mapPartitions

- Apply a single function over the entire partition and output the results

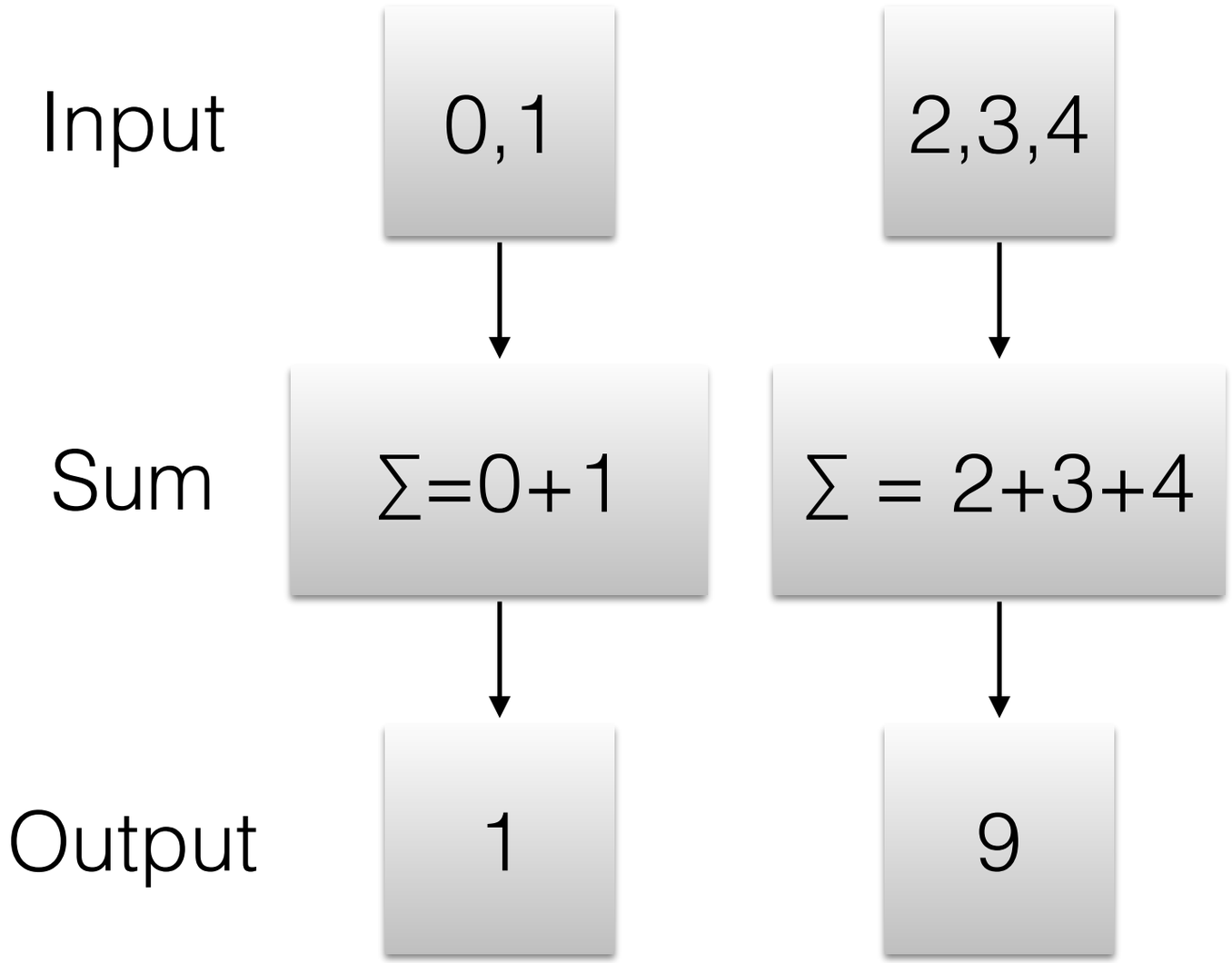
- ```
  >>> rdd = sc.parallelize([0,1,2,3,4],2)

  >>> rdd.mapPartitions(lambda partition: [sum(partition)]
  ).collect()

  [1,9]
  ```


  ```

Spark: mapPartitions



Spark: zipWithIndex

- Append a record index, starting from the first record in the partition to the last record in the last partition

- ```
  >>> rdd = sc.parallelize([4,3,2,1,0],2)
  ```

- ```
  >>> rdd.zipWithIndex().collect()
  [(4, 0), (3, 1), (2, 2), (1, 3), (0, 4)]
  ```

Spark: fold

- Combine many values into a single one
  `fold(initial, op)`

- `>>> rdd = sc.parallelize([1, 2, 3, 4, 5])`

```
>>> from operator import add
>>> rdd.fold(0, add)
15
```

- **Ordering is not guaranteed**, assumes that the function is associative and commutative i.e f(x,y) = f(y,x) **and** f(f(x,y),z) = f(f(x,z),y)

Example: N-gram counting across boundaries

- **Unigrams (n = 1)**
  [Line 0] from, fairest, creatures, we, desire, increase
  [Line 1] that, thereby, beauty's, rose, might,never, die
  [Line 2] but, as, the, riper, should, by, time, decease

- **Bigrams (n = 2)**
  [Line 0] (from, fairest), (fairest, creatures) …
  (desire, increase)
  [Line 1] (that, thereby) … (never, die)
  …

Example: N-gram counting across boundaries

- **Unigrams (n = 1)**

```
[Line 0] from, fairest, creatures, we, desire, increase
[Line 1] that, thereby, beauty's, rose, might,never, die
[Line 2] but, as, the, riper, should, by, time, decease
```

- **Bigrams (n = 2)**

```
[Line 0] (from, fairest), (fairest, creatures) …
         (desire, increase)
[Line 1] (that, thereby) … (never, die)
…
```

- **Where is:** (increase, that), (die, but)**?**

Does not exist! **Bigrams did not cross the line boundary.**

Example: N-gram counting across boundaries

- How can we solve this?

- The key transformation is zipWithIndex

Example: N-gram counting across boundaries

Shakespare

```
1609

THE SONNETS

by William Shakespeare

                    1
From fairest creatures we desire increase,
That thereby beauty's rose might never die,
But as the riper should by time decease,
```

Example: N-gram counting across boundaries

- `>>> tokenized = shakespeareText.flatMap(tokenize, preservesPartitioning=True)`
- `>>> tokenized.take(10)`

```
[u'the',
 u'sonnets',
 u'by',
 u'william',
 u'shakespeare',
 u'from',
 u'fairest',
 u'creatures',
 u'we',
 u'desire']
```

Example: N-gram counting across boundaries

- `>>> tokenized = shakespeareText.flatMap(tokenize, preservesPartitioning=True)`
- `>>> tokenized.take(10)`

```
[u'the',
u'sonnets',
u'by',
u'william',
u'shakespeare',
u'from',
u'fairest',
u'creatures',
u'we',
u'desire']
```

Important! Guarantees ordering.

Example: N-gram counting across boundaries

- >>> indexed = tokenized.zipWithIndex()
- >>> indexed.take(10)

```
[(u'the', 0),
(u'sonnets', 1),
(u'by', 2),
(u'william', 3),
(u'shakespeare', 4),
(u'from', 5),
(u'fairest', 6),
(u'creatures', 7),
(u'we', 8),
(u'desire', 9)]
```

Example: N-gram counting across boundaries

- The goal is to be able to group the N-grams together, and I used the convention of looking back N-1 steps.

- Let say we have `(u'sonnets', 1)`
  Then we want to place this in group 0 and 1 for n = 2.

- `[(0, (u'sonnets', 1)), (1, (u'sonnets', 1)]`

Example: N-gram counting across boundaries

- >>> n = 2
- >>> distributed = (
    indexed.flatMap(**lambda** (tok,k):
      [(i, tup) **for** i **in** xrange(k-n+1, k+1)]
)
)

Example: N-gram counting across boundaries

- `>>> distributed.take(10)`

```
[(-1, (u'the', 0)),
 (0, (u'the', 0)),
 (0, (u'sonnets', 1)),
 (1, (u'sonnets', 1)),
 (1, (u'by', 2)),
 (2, (u'by', 2)),
 (2, (u'william', 3)),
 (3, (u'william', 3)),
 (3, (u'shakespeare', 4)),
 (4, (u'shakespeare', 4))]
```

Needs to be filtered out in the end.

Example: N-gram counting across boundaries

Ordering of group items not guaranteed!

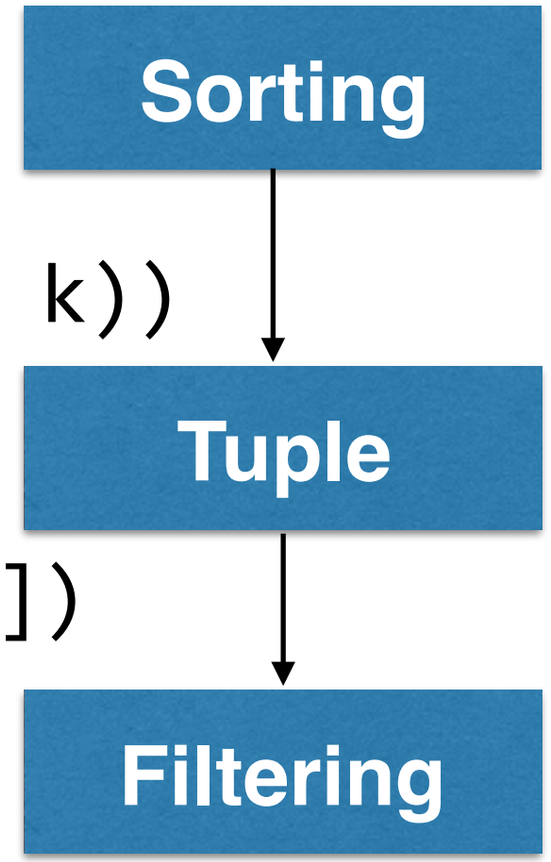
- `>>> grouped = distributed.groupByKey()`

- `>>> grouped.take(10)`

```
[(0,      [(u'the', 0),              (u'sonnets', 1)]),
 (262144, [(u'for', 262144),         (u'the', 262145)]),
 (872904, [(u'thou', 872904),        (u'speak', 872905)]),
 (8,      [(u'desire', 9),           (u'we', 8)]),
 (803072, [(u'bosoms', 803073),      (u'their', 803072)]),
 (16,     [(u'might', 16),           (u'never', 17)]),
 (87384,  [(u'of', 87384),           (u'the', 87385)]),
 (733240, [(u'for', 733240),         (u'me', 733241)]),
 (24,     [(u'by', 24),              (u'time', 25)]),
 (61104,  [(u'alexandria', 61104), (u'enter', 61105)])]
```

Example: N-gram counting across boundaries

```
>>> tuples = grouped.map(
    lambda (k, grp):
    (k, sorted(grp, key=lambda(tok,k): k))
).map(
    lambda (k, grp):
    tuple([token for (token, k) in grp])
).filter(
    lambda tup: len(tup) == n)
)
```



Example: N-gram counting across boundaries

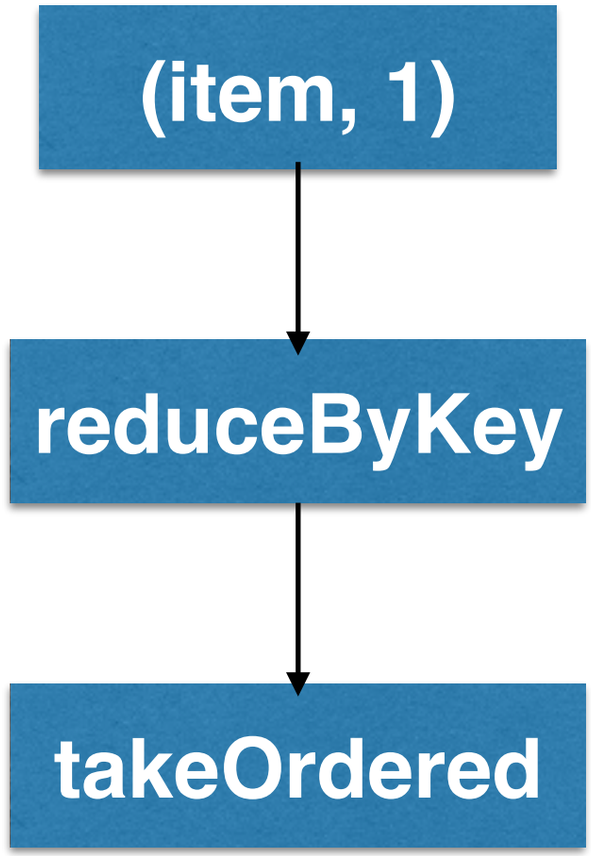
```
>>> tuples.take(10)

[(u'the', u'sonnets'),
 (u'for', u'the'),
 (u'thou', u'speak'),
 (u'we', u'desire'),
 (u'their', u'bosoms'),
 (u'might', u'never'),
 (u'of', u'the'),
 (u'for', u'me'),
 (u'by', u'time'),
 (u'alexandria', u'enter')]
```

Example: N-gram counting across boundaries

```
>>> tuples.map(lambda tup: (tup,1)
).reduceByKey(lambda x,y: x+y
).takeOrdered(10, lambda (tup,s): -s)

[((u'i', u'am'), 1855),
 ((u'i', u'll'), 1745),
 ((u'my', u'lord'), 1666),
 ((u'in', u'the'), 1643),
 ((u'i', u'have'), 1620),
 ((u'i', u'will'), 1566),
 ((u'of', u'the'), 1496),
 ((u'to', u'the'), 1430),
 ((u'it', u'is'), 1078),
 ((u'to', u'be'), 973)]
```



Example: N-gram counting across boundaries

```
>> n = 4

>> ngram(shakespeareText, 4)
[((u'what', u's', u'the', u'matter'), 76),
 ((u'with', u'all', u'my', u'heart'), 47),
((u'another', u'part', u'of', u'the'), 44),
((u'i', u'know', u'not', u'what'), 40),
((u'exeunt', u'act', u'v', u'scene'), 37),
((u'william', u'shakespeare', u'dramatis', u'personae'), 36),
((u'by', u'william', u'shakespeare', u'dramatis'), 36),
((u'the', u'duke', u'of', u'york'), 36),
((u'what', u'say', u'st', u'thou'), 35),
((u'exeunt', u'act', u'ii', u'scene'), 33)]
```

Accumulators

- Shared Write Only variables
- Optimized to collect side outputs from parallel execution of jobs and sending the information to the driver.
- In Hadoop, MapReduce counters were a primitive form of Accumulators

Accumulators

- Accumulators are a generalization of counters, to accept any data.

- Not meant to be used for large outputs because it resides in memory and is transmitted directly to the driver.

Accumulators

- The simplest accumulators are those based of counters:

- ```
  >>> counter = sc.accumulator(0)
  >>> counter.value
  0
  ```

```
>>> def identity(x):
>>>     counter.add(1)
>>>     return x

>>> sc.parallelize([0,1,4,9]).map(identity).count()
4

>>> counter.value
4
```


```

Accumulators

- The simplest accumulators are those based of counters:
- `>>> sc.parallelize([0,1,4,9]).map(identity).count()`
  `4`

```
>>> counter.value
4

>>> sc.parallelize([4,2]).map(identity).count()
2

>>> counter.value
6
```

Will just continue counting…

Accumulators

```
from pyspark import AccumulatorParam

class ListAccumulatorParam(AccumulatorParam):
    def zero(self, initialValue):
        return initialValue

    #v1 and v2 are lists
    def addInPlace(self, v1, v2):
        v1.extends(v2)
        return v1
```

Initial value initialization

Adding function