PROFINIT

Apache Spark - basics

Martin Oharek

Outline

Outline

PROFINIT

1. Spark overview
2. How Spark works
3. Spark Dataframes
4. Spark architecture
5. Spark configuration
6. Spark vs Databricks

Spark overview

The What, Why and When of Apache Spark

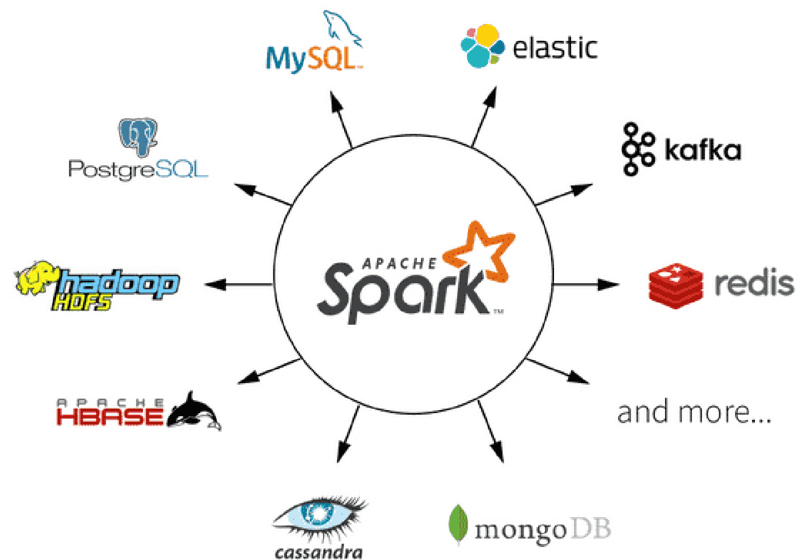
PROFINIT

- **What:**
  - Unified engine for big data and machine learning
  - Distributed data processing engine -> up to petabytes of data up to thousands of physical or virtual machines
  - Open Source with over 1000 contributors from 250+ organizations
  - Founded by people who founded Databricks
- **Why:**
  - High speed data querying, analysis, and transformation with large data sets.
  - Great for iterative algorithms (using a sequence of estimations based on the previous estimate).
  - Supports multiple languages (Java, Scala, R, Python)
  - Free of charge
- **When:**
  - When you're using functional programming (output of functions only depends on their arguments, not global states)
  - Performing ETL or SQL batch jobs with large data sets
  - Processing streaming
  - Machine Learning tasks

Spark - facts

PROFINIT

- In-memory Map-Reduce engine
- Written in Scala
- Fault-tolerant
- Connected with all major big data technologies
- Runs „Everywhere“



Apache Spark Evolution

PROFINIT

- **Spark 1.x – 2014 :**
  - Spark CORE - Fault-tolerant in memory computation engine
  - Spark RDD (Resilient Distributed Dataset) API
  - API for Streaming and Mlib
  - Spark SQL
- **Spark 2.x - 2016:**
  - Speedups the computation 5 to 20 times.
  - API for structured Streaming
  - API for graph data processing
  - SQL 2003 support
  - Datasets API over RDD
- **Spark 3.x - 2020:**
  - adaptive query execution, dynamic partition pruning and other optimizations
  - Significant improvements in pandas APIs, including Python type hints and additional pandas UDFs
  - Up to 40x speedup for calling R user-defined functions
  - SQL ANSI supports

When does Spark work best?

PROFINIT

- On distributed data systems or NoSQL Databases
- Collaboration – Data engineers, data scientist, BI analyst, ..
- Batch and streaming tasks

Common uses:

1. Calculation of client scores (risk score, fraud detection)
2. ETL or SQL batch jobs
3. Using streaming data to trigger a response
4. Machine Learning tasks
5. Graph algorithms

When not to use Spark?

PROFINIT

- Small data
- Low computing capacity (memory)
- Poorly parallelizable
- real-time

e.g.:

1. Modeling on small data
2. Ingesting data in a publish-subscribe model
3. Median calculation
4. JOIN of very big tables

How to work with Spark?

PROFINIT

- Interactively
  - Command line (shell for both Python and Scala)
  - **Databricks notebook**
  - Zeppelin/Jupyter notebook
  - From IDE (Pycharm, IntelliJ, …)

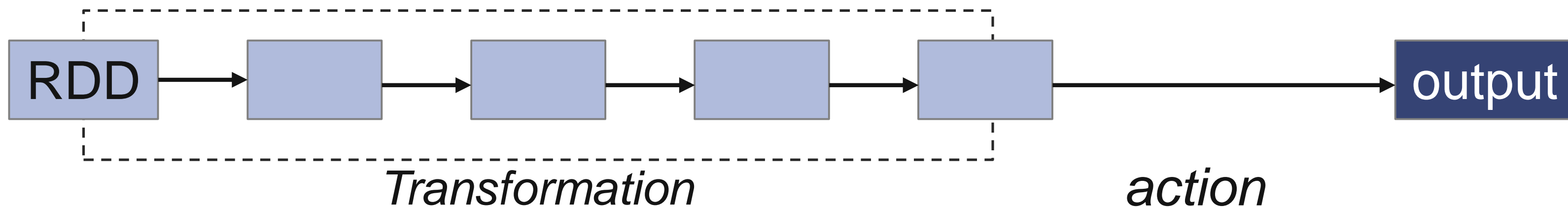
- Batch / application
  - compiled .jar file
  - *.py file

- Learning path:
  - http://spark.apache.org
  - https://www.databricks.com/spark/getting-started-with-apache-spark

How Spark works

Logical point of view

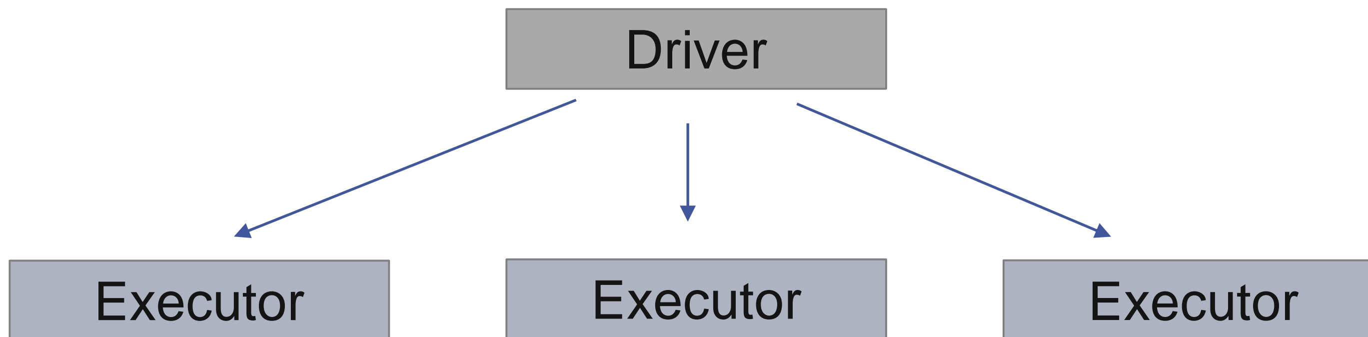
PROFINIT



- **RDD:**
  - resilient distributed dataset - the abstractions of Spark. It is used to handle distributed collection of data elements (e.g.: rows in text file, data matrix, set of binary data) across all the nodes in a cluster.
  - is immutable
- **Transformation:**
  - are planned and optimized, but not evaluated
  - planned as DAG – Direct acyclic graph
- **Action:**
  - action is a trigger that started the whole process

Technical point of view

PROFINIT



- **Driver:**
  - Control all processes
  - Convert user code to transformations and actions -> tasks
  - Distribute tasks across executors
- **Executor:**
  - „worker“ – run tasks and return result to a driver
- **Both run as JVM**

Example – word count

PROFINIT

- Task: **count number of words in document**
- Source: text file splitted to lines
- Approach:
  - Load file from disk
  - Transformation of lines: line ⇨ split to words ⇨ split to items (word, 1)
  - Group items with the same word and sum up ones
- Result of transformation: RDD with items (word, frequency)

Example – word count

PROFINIT

Transformation:

```
lines = sc.textFile("bible.txt")
words = lines.flatMap(lambda line: line.split(" "))
items = words.map(lambda word: (word, 1))
counts = items.reduceByKey(lambda a, b: a + b)
```

Action:

```
counts.take(5)
```

Spark Dataframes

Spark SQL and DataFrames (DataSets)

PROFINIT

- New from spark 2.x ⇨ Enhances the classical RDD approach
- Data structure **DataFrame** = „RDD with columns“
  - similar to database relation table
  - with metadata (field names, types)
  - works with columns –> SQL syntax can be used

- RDD

1;Andrea;35;64.3;Praha
2;Martin;43;87.1;Ostrava
3;Simona;18;57.8;Brno

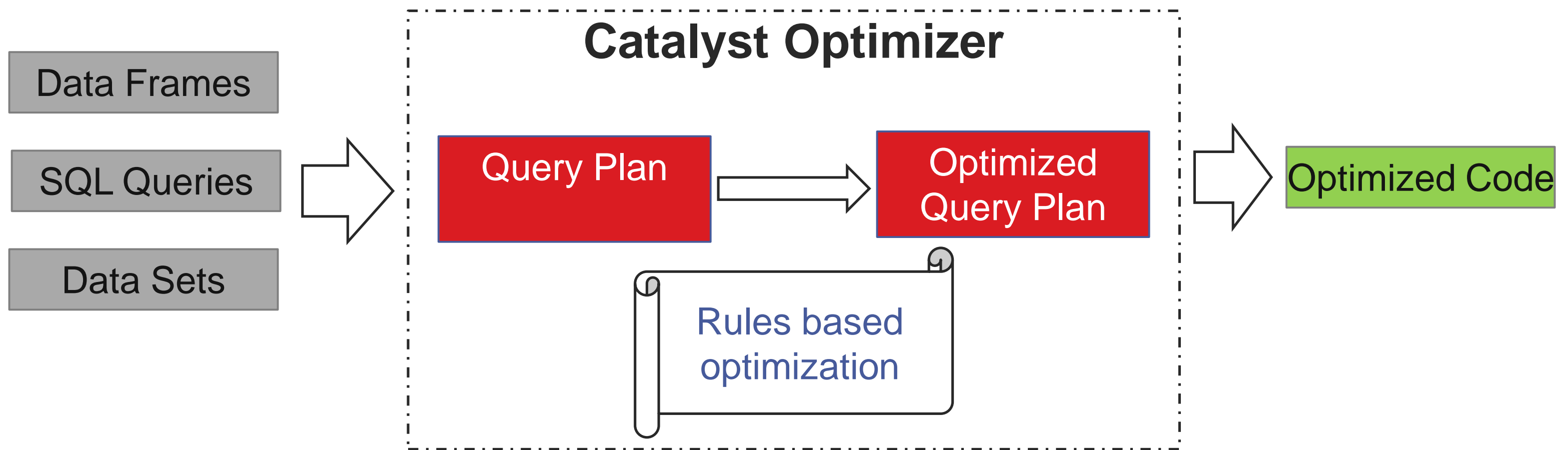
- Dataframe

id	name	age	weight	city
1	Andrea	35	64.3	Praha
2	Martin	42	87.1	Ostrava
3	Simona	18	57.8	Brno

WHY use DataFrames

PROFINIT

- Advantage over Spark RDD:
  - Dataframe API - shorter and easier code
  - Columns and Types
  - SQL language can be used
  - Simplified work with databases
  - **Catalyst Optimizer can be applied ⇨ is faster**



How to get a DataFrame?

PROFINIT

- transformation from existing RDD
  - if convertable
  - `sqlContext.createDataFrame(RDD, schema)`
- direct input of file
  - schema may be defined (Parquet, ORC) or inferred (CSV)
  - `sqlContext.read.format(format).load(path)`
- Hive query
  - `sqlContext.sql(sql_query)`

How to work with a DataFrame?

PROFINIT

1. registration of temporary table + SQL querying
   - `DF.registerTempTable("table")`
   - `sqlContext.sql("select * from table")`
2. SPARK API
   - `DF.operations`; select, filter, join, groupBy, sort...
3. Convert to RDD -> RDD operation (map, flatMap, …) and then convert back -> Dataframe

Example – word count with Dataframes

PROFINIT

- Transformation

```
df_final = (
    df.withColumn("word", explode(split(col("lines"), ' ')))
    .groupBy("word")
    .count()
)
```

- Action

```
df_final.show()
```

Example – word count with Spark SQL

PROFINIT

- Transformation
- `df.registerTempTable("temp_df")`
- `df_final = (`
- `sqlContext.sql("`
- `    SELECT word, count(*) FROM`
- `    (SELECT explode(split(Description, ' ')) AS word FROM temp_df)`
- `    GROUP BY word`
- `")`

- Action

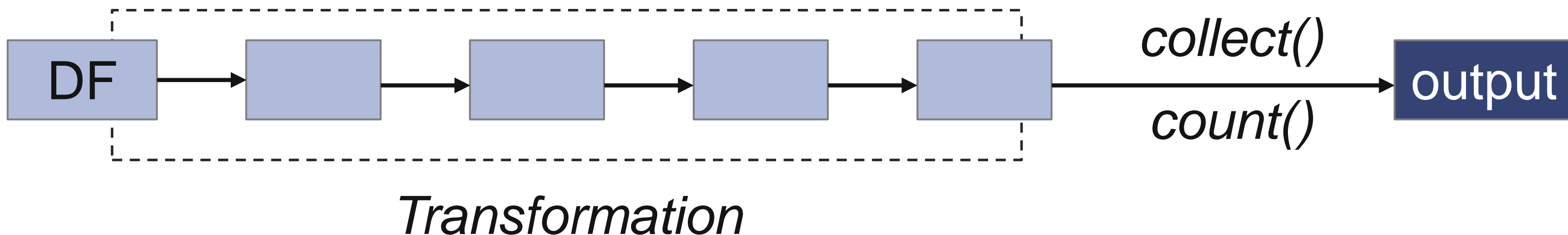
```
df_final.show()
```

Spark Actions

Spark Actions

PROFINIT

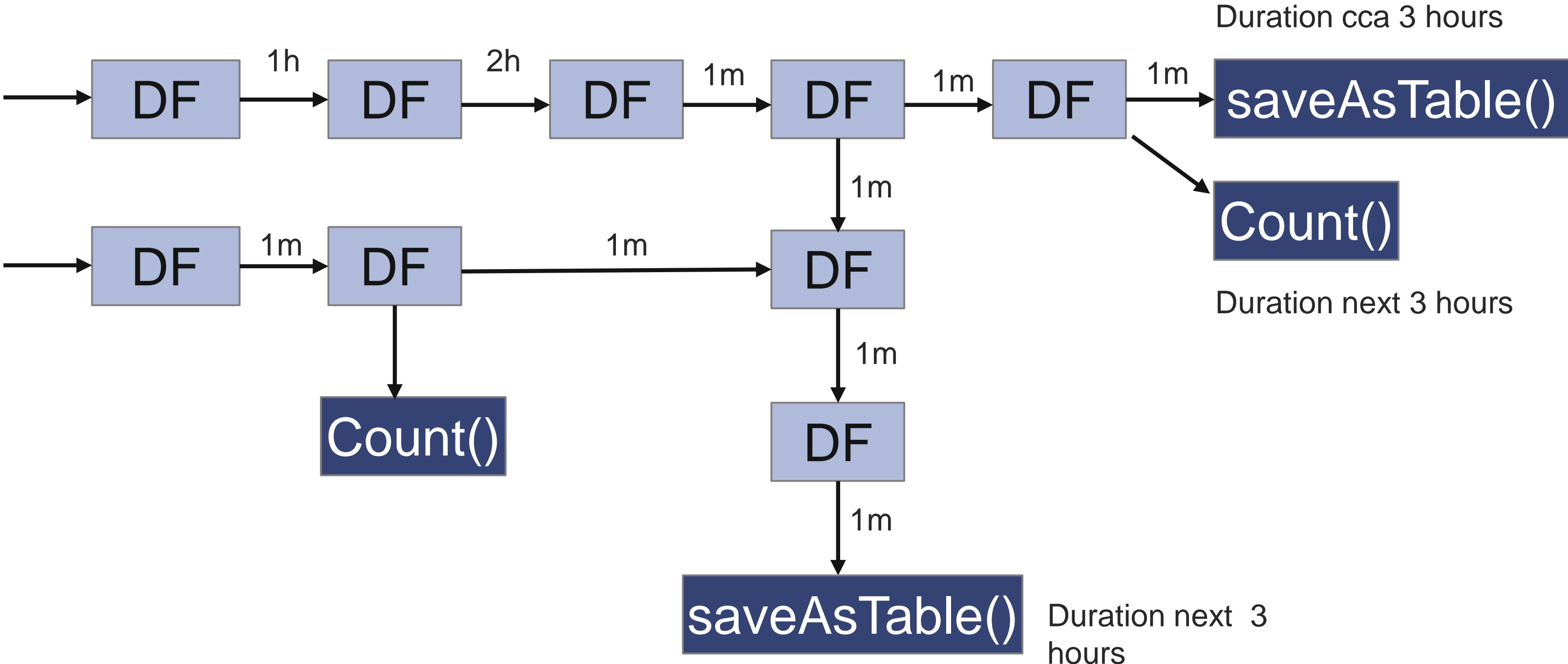
RDD	Dataframe	Description
take	take,show	Show first n rows
count	count	Count of rows
collect	collect	Show rdd/dataframe as list of rows
saveAsTextFile	saveAsTable, write	Save file/create table
…	…	



- Every action starts all steps of transformation from the beginning!

Spark Actions

PROFINIT



Spark Caching

PROFINIT

- **Methods:**
  - `persist()` (several options)
  - `cache()` (use persist with MEMORY_ONLY option)
  - `unpersist()` (release persisted data)

- **Persist options:**
  - MEMORY_ONLY – Default –> deserialized JVM memory
  - MEMORY_AND_DISK –> excessed partitions into disk.
  - MEMORY_ONLY_SER -> serialized JVM memory
  - MEMORY_AND_DISK_SER -> etc.

- Persist is not an action!

Spark Caching

PROFINIT

- Different from (proprietary) **Databricks Disk Cache – optimized caching on SSDs**

Feature	disk cache	Apache Spark cache
Stored as	Local files on a worker node.	In-memory blocks, but it depends on storage level.
Applied to	Any Parquet table stored on S3, ABFS, and other file systems.	Any DataFrame or RDD.
Triggered	Automatically, on the first read (if cache is enabled).	Manually, requires code changes.
Evaluated	Lazily.	Lazily.
Force cache	`CACHE SELECT` command	`.cache` + any action to materialize the cache and `.persist`.
Availability	Can be enabled or disabled with configuration flags, enabled by default on certain node types.	Always available.
Evicted	Automatically in LRU fashion or on any file change, manually when restarting a cluster.	Automatically in LRU fashion, manually with `unpersist`.

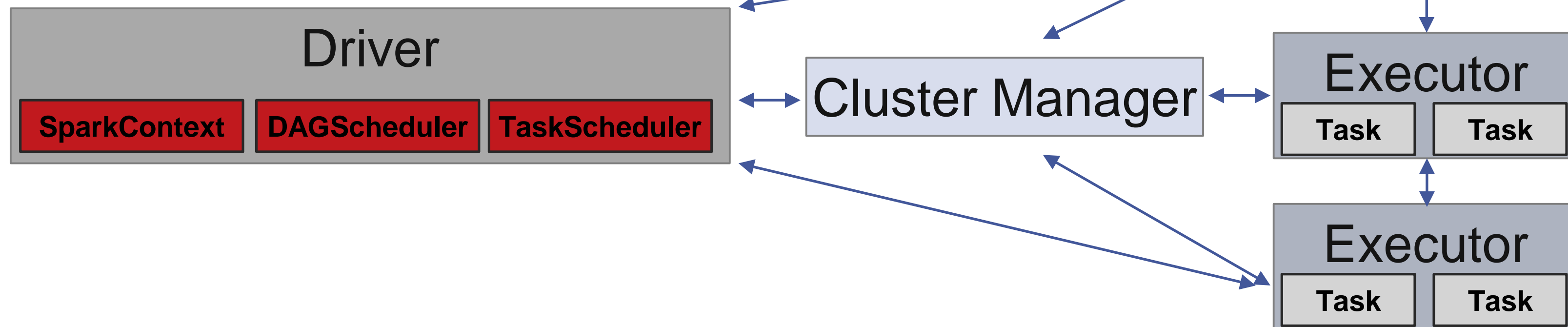
- Cache consistency:
  - Databricks disk caching – changes are automatically detected and cache is updated
  - Spark caching – cache must be manually invalidated and refreshed

Spark Architecture

Components of Spark Architecture

PROFINIT

- **Driver**
  - It is a **master node**.
  - Translates user code into a specified job.
  - Schedules the job execution and negotiates with the cluster manager.
  - Stores the metadata about all RDDs as well as their partitions.
  - The key component is a SparkContext, others are DAG Scheduler, Task scheduler, backend scheduler and block manager.
- **Executors**
  - They are distributed agents those are responsible for the execution of tasks
  - They perform all the data processing
- **Cluster Manager**
  - Responsible for acquiring resources



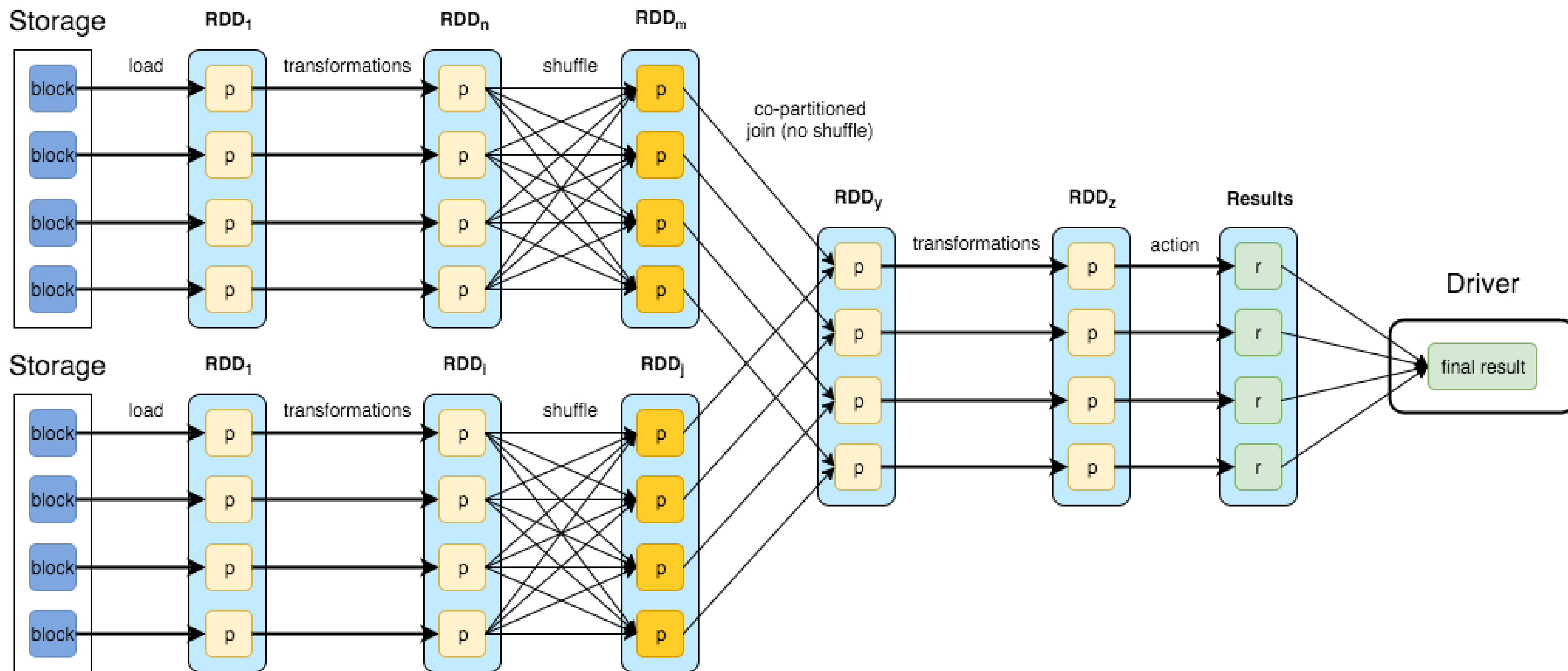
Spark data partitions

PROFINIT

- **Partition**
  - part of data managed in one task
  - default partition = 1 HDFS block = 1 task = 1 core
  - partition is ideally managed on the node where is stored – data locality!
  - More partitions ⇨ more tasks ⇨ higher parallelization
    - ⇨ smaller data ⇨ lower efficiency ⇨ higher overhead
- **Default for Joins:**
  - The default number of partitions to use when shuffling data for joins or aggregations.
  - `spark.sql.shuffle.partitions = 200`
- **How to change number of partition?**
  - in load: `sc.textFile(file, count_of_partitions)`
  - In the code (before/after specific transformation/action):
    - `coalesce (count_of_partitions)`
    - `repartition(count_of_partitions)`
    - `partitionBy (count_of_partitions)`

Data locality & shuffling

PROFINIT



Start and configuration

- `pyspark | spark-shell | spark-submit --param value`
- **Useful parameters:**
- --name -> name of the application
- *--class -> The entry point for your application*
- *--master -> The master URL for the cluster (local, Yarn, Mesos, ..)*
- *--deploy_mode -> where the driver will be deployed (client/cluster)*
- --driver-memory -> memory for driver
- --num-executors -> count of executors
- --executor-cores -> count of cores for executor
- --executor-memory -> memory for *executor*
- *NOTE:* **Spark is deployed in Databricks clusters by default and Spark Context (Spark session) is initialized, you don‘t need to care about running Spark on your own**

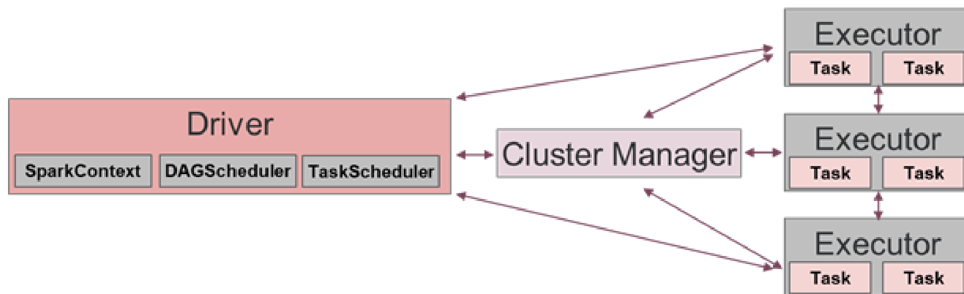
Deploy mode

Deploy mode (execution mode)

PROFINIT

- **Deploy mode**
- Determines where the resources used by Spark application are physically located
- **Deploy mode types:**
  - Local mode
  - Client mode
  - Cluster mode

- **Differences:**
- Where the driver runs – client or cluster ?
- Where the executors run - client or cluster ?
- What is cluster manager – spark CM or 3rdParty (yarn, messos, ..)



Deploy mode: Local mode

PROFINIT

- Properties:
  - The entire application is run on a single machine (paralelism through threads)
  - The Spark **driver runs on the client machine**
  - The **Executor** processes **run on the client machine**
  - **Spark CM** is used
  - Used on Databricks single node clusters
- Purpose:
  - Development
  - Debugging
  - Testing

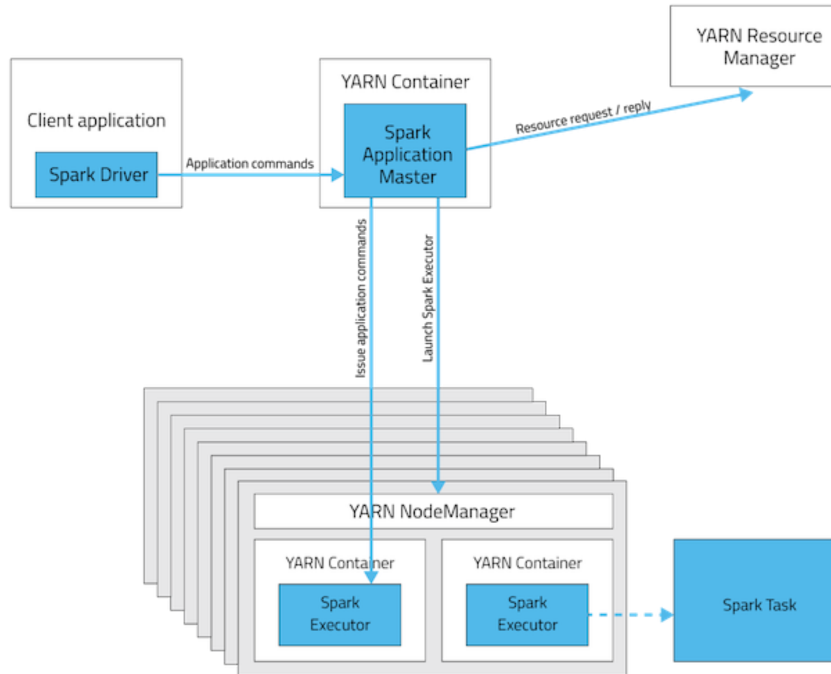
Deploy mode: Client mode

PROFINIT

- Properties
  - The **Spark driver runs on the client machine** that submitted the application (usually an edge node)
  - The **executor** processes run on **cluster**
  - Cluster manager is used
  - On Databricks multi-node clusters in interactive environment (e.g. Notebook)
- Purpose
  - Spark-shell (interactive sessions)
  - Easy debugging
    - Input and output attached
- Can overload the edge node

Deploy mode: Client mode (example for YARN)

❬ PROFINIT ❭



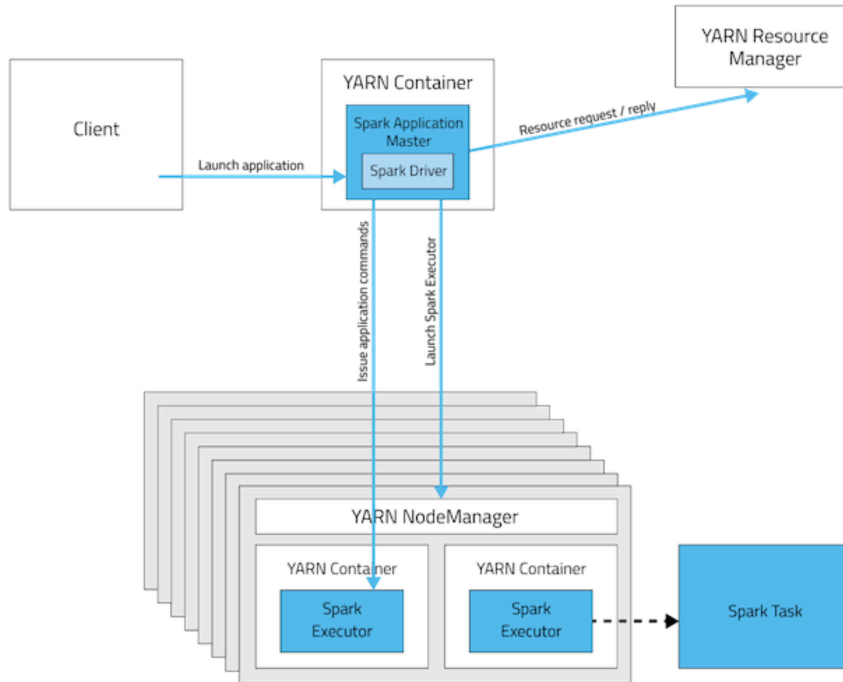
Deploy mode: Cluster mode

PROFINIT

- Properties
  - The **Spark driver runs on a worker node** inside the cluster
  - The **executor** processes run on **cluster**
  - The cluster manager maintans the executor processes
  - **Databricks job clusters**
- Purpose
  - The best deploy mode for stable applications
  - Better resource utilization than in client mode
  - More difficult debugging

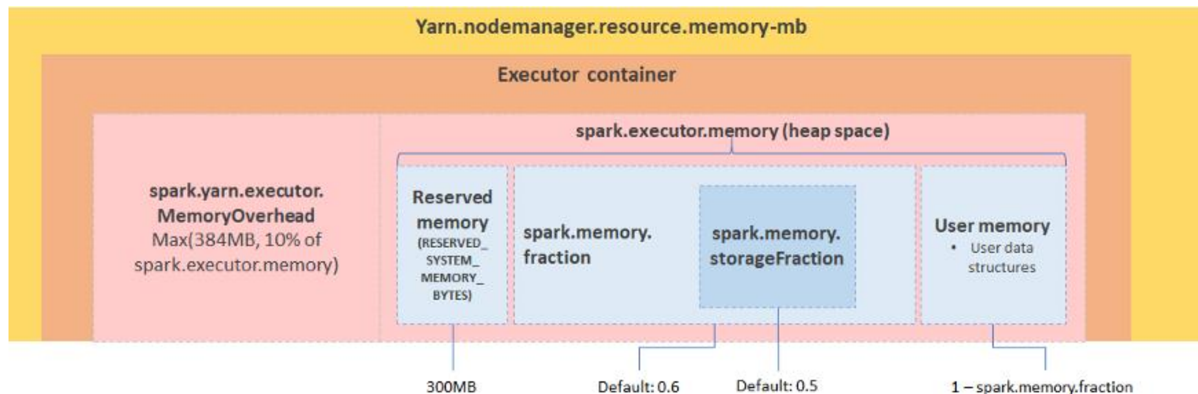
Deploy mode: Cluster mode (example for YARN)

PROFINIT



Spark configuration

Spark executor memory



- **Reserved** - the memory is reserved for the system and is used to store Spark's internal object. The size is hardcoded.
- **User Memory** - It's used for storing your data structures and data needed for RDD conversion operations, such as lineage.
- **Unified memory:**
  - **Execution memory** - It's mainly used to store temporary data in the calculation process of Shuffle, Join, Sort, Aggregation, etc.
  - **Storage Memory** - It's mainly used to store Spark cache data, such as RDD cache, Unroll data, and so on.
  - *Size* of an Execution and Storage memory can by *dynamically changed* by the **Dynamic occupancy mechanism** process.
- **Memory overhead** - Off heap (no GC). Call stacks, shared libraries, constants defined in Code, the code itself, ….

Spark executor memory example

PROFINIT

- Spark.executor.memory = 4 GB
  - **Memory overhead** = 10% of executor memory, max 384 MB
  - **Reserved memory** = 300 MB
  - **User Memory** = (Java Heap — Reserved Memory) * (1.0 — spark.memory.fraction) = (3640-300)* (1-0,6)= 1336 MB
  - **Storage Memory** = (Java Heap — Reserved Memory) * spark.memory.fraction * spark.memory.storageFraction = (3640-300)*(0,6*0,5)= 1002 MB
  - **Execution Memory** = (Java Heap — Reserved Memory) * spark.memory.fraction * (1.0 — spark.memory.storageFraction) = (3640-300)*(0,6*0,5)=1002 MB

Resources configuration: Settings

PROFINIT

Available settings:

- spark.driver.memory
  - Size of the Spark driver in MB
  - Default 1024MB
- spark.executor.memory
  - Size of the each Spark executor in MB
  - Default 1024MB
- spark.executor.cores
  - The number of virtual cores that will be allocated to each executor
  - Default 1 (YARN)
- Spark.dynamicAllocation.enabled
  - Allows Spark dynamically change the number of executors based on the workload

Resources configuration: Spark Driver

PROFINIT

Considerations:

- Client or cluster mode
- With the client mode – beware of overloading Edge node
- Size of the result returned by executor (collect action)

Resources configuration: Spark Executor

PROFINIT

Considerations

- Resources available in the cluster, sizing of cluster nodes
- Few large executors or many small executors?
  - **Small executors**
    - Higher parallelization but more shuffling
    - One partition - one executor, risk of spilling the data to disk
    - Total overhead grows (Reserved memory)
  - **Large executors**
    - Lower parallelization
    - Issue with resources allocation
    - Might be wasteful
    - GC overhead

Resources configuration: Recommendations

- Allows Spark dynamically change the number of executors based on the workload
- Number of cores – deside based on the load. Usually 2 - 4 cores/executor
- Driver memory – keep default
- Executor memory – ((data size) *1,5)/0,6) / number of executors (max 16G)
  - For start use `spark.executor.memory = 2G`.
- Number of executors - number of task / executor > 100
  - For start use `spark.dynamicAllocation.maxExecutors` < 10.
- For long running processes set `spark.sql.ui.retainedExecutions <= 100 (default 1000)`

Spark vs Databricks

Spark vs Databricks

PROFINIT

- Databricks
  - Tool/platform built **on top of Apache Spark**
  - Add other functionality, e.g. Notebooks, production jobs and workflows, etc.
  - Databricks runtime
    - Built on Apache Spark and optimized for performance
      - Photon engine
      - Disk caching, dynamic file pruning, predictive I/O, cost-based optimizers, etc.
      - Auto-scaling compute
      - Pre-installed Java, Scala, Python and R libraries
      - …
  - Apache Spark is running on Databricks clusters
    - Can set spark configuration on cluster level and change some configurations during runtime
    - Managed Delta Lake
    - You **cannot** set spark configuration for managed compute (SQL warehouse)

Q&A

PROFINIT

Thanks for attention!

Profinit EU, s.r.o.
Tychonova 2, 160 00 Praha 6

Tel.: + 420 224 316 016, web: www.profinit.eu

LINKEDIN
linkedin.com/company/profinit

TWITTER
@profinit_EU

FACEBOOK
facebook.com/Profinit.EU

YOUTUBE
Profinit EU, s.r.o.