Distributed data processing with Apache Spark

Marek Wiewiórka, Tomasz Gambin

October 2024

Agenda

1. Introduction to Apache Spark
   - core concepts
   - programming model
2. Integration with big data ecosystem
3. A closer look at Catalyst optimizer
4. Optimizations and tuning

Apache Spark

- unified programming model suitable for both data engineering(DE) and data science(DS)
- programming interfaces in Scala/Java(DE) and Python (DS), R also but less popular
- can run on clusters managed by YARN, Mesos and *Kubernetes* (GA starting from version 3.1.1 — March 2021)
- support for JDK8 and JDK11 (Spark 3.x), Scala 2.12, Python 3.8+

Hello world the Spark way in REPL 1/5

```bash
curl -s "https://get.sdkman.io" | bash
source "$HOME/.sdkman/bin/sdkman-init.sh"
export JAVA_VERSION=11.0.10.hs-adpt
export SPARK_VERSION=3.0.1
sdk install java ${JAVA_VERSION}
sdk use java ${JAVA_VERSION}
sdk install spark ${SPARK_VERSION}
sdk use spark ${SPARK_VERSION}
```

```
spark-shell --driver-memory 2g --master yarn

Spark context available as 'sc' (master = yarn, app
↪  id = yarn-1616362928683).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.0.1
      /_/

Using Scala version 2.12.10 (OpenJDK 64-Bit Server VM,
↪  Java 11.0.10)
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

Hello world the Spark way in REPL 2/5

SparkContext and SparkSession

- both serve as entrypoints for Spark app:
  - *SparkSession* – SparkSQL (now the default one), for working with Dataframe/Dataset/SQL API
  - *SparkContext* – Spark Core, for working with RDD collections (is a part of SparkSession)
  - store configuration and contain a lot of helper methods (i.e. for reading/ writing data, timing, etc.)
  - both automatically constructed in the Spark shell

Hello world the Spark way in REPL 3/5

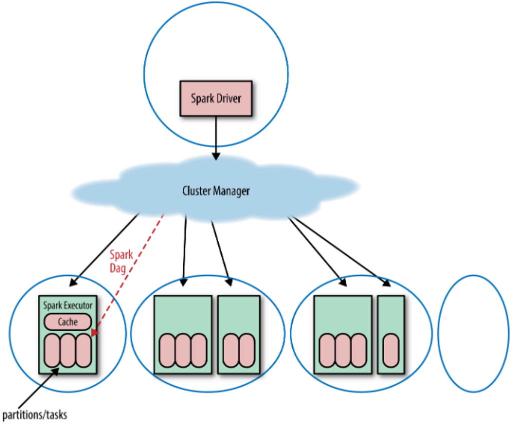
```
sc
.textFile("hdfs:///tmp/test.csv")
.flatMap(r => r.split('|') )
.map(r => s"Hello ${r} !")
.first

res18: String = Hello Stalowa Wola !
```

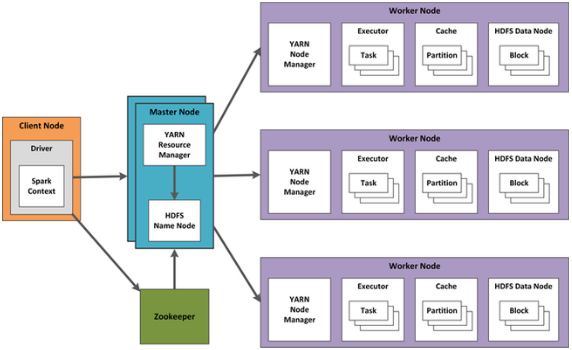
```
cat /tmp/test.csv
Stalowa Wola|72000|Poland
Sandomierz|32000|Poland
```

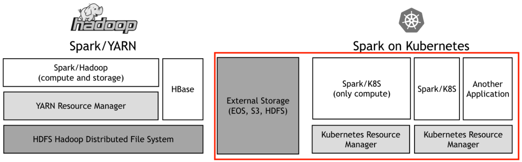
Hello world the Spark way in REPL 4/5

1. *SparkContext* defines resources, specifies deployment mode (and Cluster Manager endpoint) $=>$ *Spark Driver* is created.
2. Spark Driver negotiates resources with Cluster Manager and *Spark Executor(s)* are created on cluster worker nodes.
3. *Tasks/RDD partitions* are processed on Spark Executor(s).



Hello world the Spark way in REPL 5/5

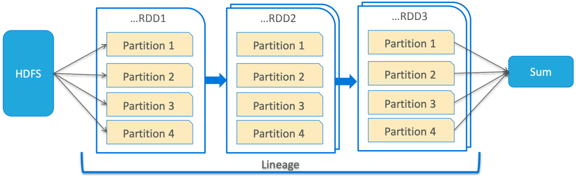




RDD - Resilient Distributed Dataset

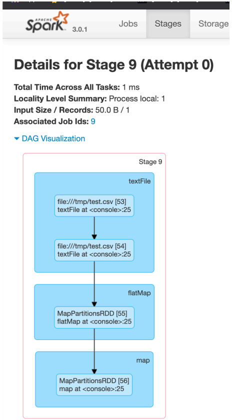
RDD Lineage

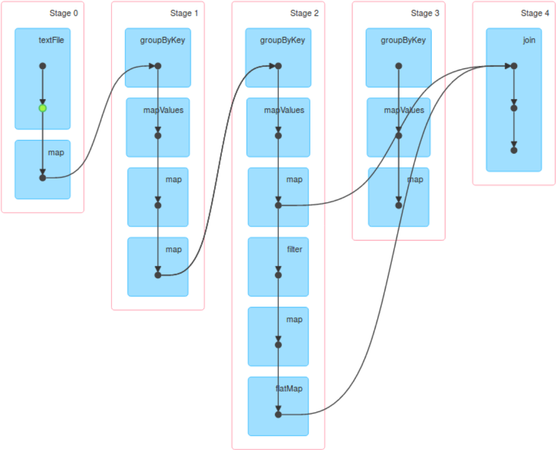
```
sc.textFile("hdfs://…", 4)
  .map((x) => x.toInt)
  .filter(_ > 10)
  .sum()
```



- distributed (partitioned) collection stored on executors
- lazy evaluated
- immutable
- can be cached in memory(and/or disk) and be replicated
- fault-tolerant thanks to lineage

Directed acyclic graph (DAG) in Spark UI

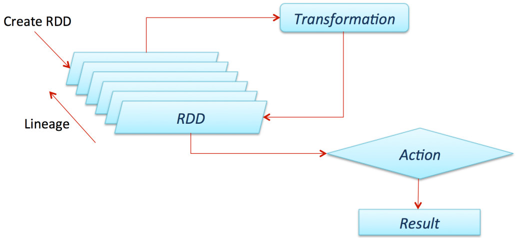




Actions vs transformations

Transformations:

- functions that return another RDD
- can be narrow or wide
- lazy by nature



Actions:

- functions that return something that is not an RDD, including a side effect
- trigger RDD computation

Transformations	Actions
map(func)	take(N)
flatMap(func)	count()
filter(func)	collect()
groupByKey()	reduce(func)
reduceByKey(func)	takeOrdered(N)
mapValues(func)	top(N)

Narrow vs wide tranformations

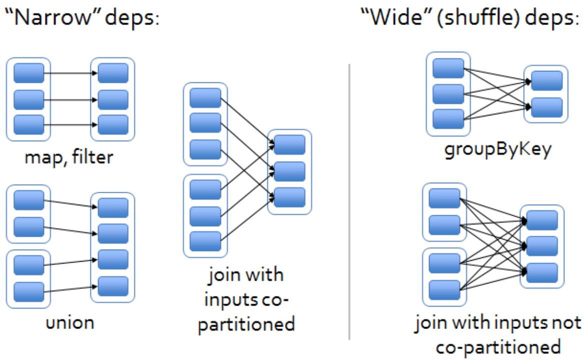


Figure: Kinds of inter-partition dependencies [3]

Anatomy of Spark Application

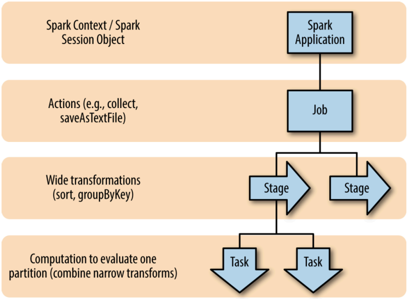


Figure: Spark Application [2]

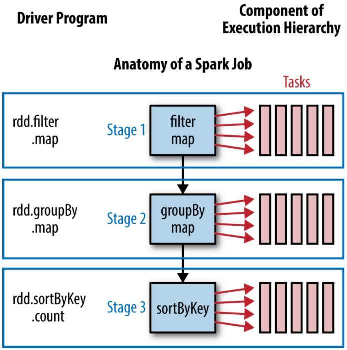
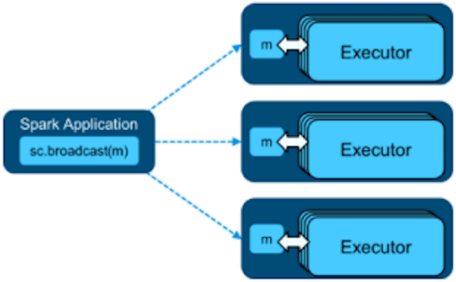


Figure: Spark Job [2]

Shared variables – broadcast variables

- read-only variables cached on each node instead shipping a copy with all tasks
- efficiently distributed using Torrent-like protocol
- used for map-side operations



Shared variables – accumulators

- write-only on executors, additive variables
- can be read on driver
- mainly for implementing various kinds of counters, sums
- by default numeric but can be customized by subclassing `AccumulatorV2`

Accumulators

Accumulable	Value
counter	45

Tasks

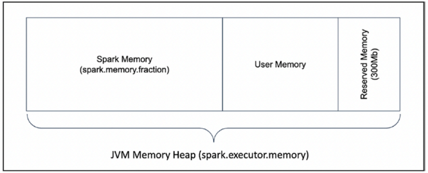
Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

RDD persistence

- `cache()` vs `persist()`
- triggered by actions
- eviction using Least Recently Used (LRU) cache policy or manually using `unpersist()`
- cache responsibly

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Spark Unified Memory Management



- when *execution* memory exceeds its compartment, it can borrow as much of the storage memory as is free
- when *storage* memory exceeds its compartment, it can borrow as much of the execution memory as is free
- when *execution* needs more memory and some of its memory was borrowed by the storage compartment, it can forcefully evict that memory occupied by storage (the other way round is *not* possible, must wait!).

SparkSQL and big data ecosystem

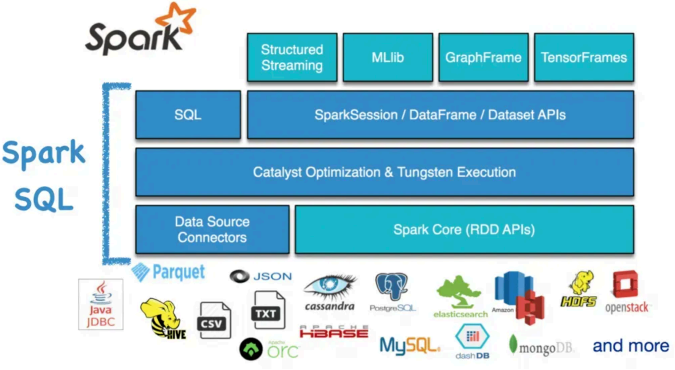


Figure: SparkSQL module [4]

SparkSQL components

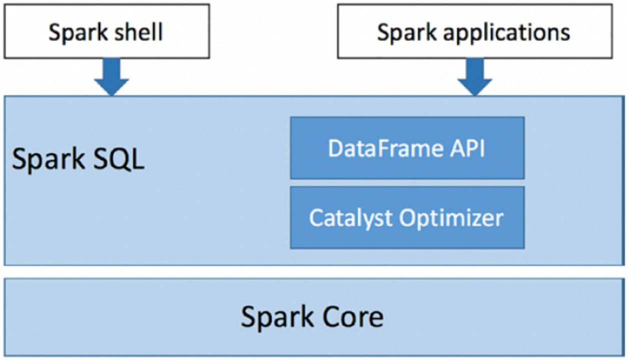


Figure: SparkSQL components[4]

SparkSQL - APIs at glance

Dataframe API:

```
spark
.read
.option("separator","|")
.option("header","true")
.csv("/tmp/test.csv")
.show
```

Dataset API

```
case class City(city: String,
 ↪  population: String,
 ↪  country:String)
spark.read
  .option("header", "true")
  .option("separator","|")
  .csv("/tmp/test.csv")
  .as[City]
  .show
```

SQL:

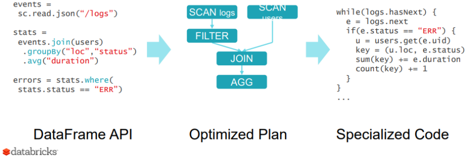
```
sql("""CREATE TABLE test
USING com.databricks.spark.csv
OPTIONS (
path "/tmp/test.csv",
header "true",
separator "|")""")
sql("SELECT * FROM test").show
```

```
+------------+----------+-------+
|        city|population|country|
+------------+----------+-------+
|Stalowa Wola|     72000| Poland|
|  Sandomierz|     32000| Poland|
+------------+----------+-------+
```

Rule vs cost optimization

- *rule based optimizer (RBO)* relies on application of predefined heuristics, e.g. : `PredicatePushdown`, `ColumnPruning`, `PartitionPruning`, `ConstantFolding`
- *cost based optimizer (CBO)* tries to estimate the cost of operators using datasets(tables, columns) statistics such as row counts, histograms to choose the best query execution plan
- Spark by default uses only RBO, but CBO can be also turned on by setting (`spark.sql.cbo.enabled`)
- CBO in Spark is used mainly for optimization of joins (e.g. joins reorder, star-schema transformation)

Catalyst optimizer - overview



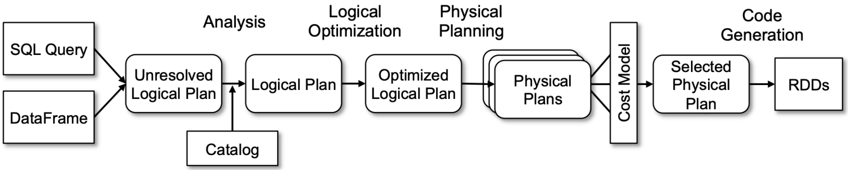


Figure: Phases of query planning in Spark SQL [1]

Catalyst optimizer - trees and rules

- The main data type in Catalyst is a *tree* composed of *node* objects. Each node has a node type and zero or more children
- Trees can be manipulated using *rules*, which are functions from a tree to another tree
- the most common approach is to use a set of pattern matching functions that find and replace subtrees with a specific structure

Catalyst optimizer constant-folding example 1/2

```sql
SELECT pos_start + (1+2) FROM reads
```

Constant-folding rule:

```scala
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)

  case Add(left, Literal(0)) => left

  case Add(Literal(0), right) => right
}
```

Catalyst optimizer constant-folding example 2/2

```
scala> val query= "SELECT pos_start + (1+2) FROM reads"
query: String = SELECT pos_start + (1+2) FROM reads

scala> sql(query).explain(true)
== Parsed Logical Plan ==
'Project [unresolvedalias(('pos_start + (1 + 2)), None)]
+- 'UnresolvedRelation [reads]

== Analyzed Logical Plan ==
(pos_start + (1 + 2)): int
Project [(pos_start#534 + (1 + 2)) AS (pos_start + (1 + 2))#614]
+- SubqueryAlias spark_catalog.default.reads
   +- Relation[sample_id#529,qname#530,flag#531,contig#532,pos#533,pos_sta
_CB#548,tag_CC#549,tag_CG#550,tag_CM#551,tag_CO#552,... 42 more fields] or

== Optimized Logical Plan ==
Project [(pos_start#534 + 3) AS (pos_start + (1 + 2))#614]
+- Relation[sample_id#529,qname#530,flag#531,contig#532,pos#533,pos_start#
#548,tag_CC#549,tag_CG#550,tag_CM#551,tag_CO#552,... 42 more fields] org.b

== Physical Plan ==
*(1) Project [(pos_start#534 + 3) AS (pos_start + (1 + 2))#614]
+- *(1) Scan org.biodatageeks.sequila.datasources.BAM.BDGAlignmentRelation(
```

Figure: Query explain plan

Catalyst optimizer - query processing phases in details

Analysis

- Looking up relations by name from the catalog, e.g. Hive Metastore
- resolving columns, aliases and data types

Physical planning

- cost-based optimization is only used to select some types of types of algorithms, e.g. for join operations

Logical optimization

- rule-based optimizations applied in batches

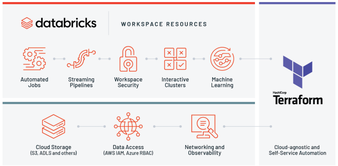
Code generation

- generating Java bytecode to run on each machine

Performance tuning

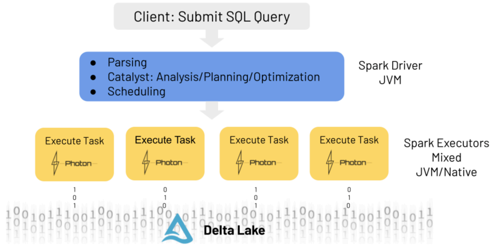
- minimize I\O operations:
  - prefer data formats/ sources that efficiently support partition and *column pruning* and *predicate pushdowns* (e.g. columnar format like ORC/Parquet over JSON, CSV)
- join optimizations:
  - prefer broadcast(map-side) joins (e.g. `BroadcastHashJoin`) over `SortMergeJoin` to avoid full data shuffle if feasible
  - use efficient data structures adjusted to the problem, e.g. tree structures as broadcasts
- use Adaptive Query Execution (AQE) (enabled by default $\geq$ 3.2.0)
  - coalescing post shuffle partitions
  - switching join strategies
  - optimizing skew joins

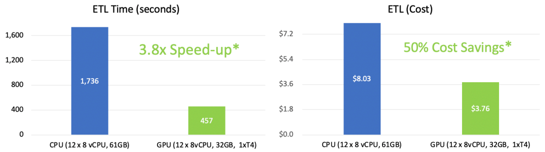
- managed Hadoop Ecosystem with Spark support: GCP Dataproc, AWS EMR, Azure HDInsight
- serverless: GCP Dataproc Batches, AWS Glue, Azure Synapse Analytics
- Kubernetes Spark Operator (GKE, EKS, AKS)
- The Databricks platform
- Snowpark (API compatible) in the Snowflake platform



Apache Spark „on steroids“ 1/2

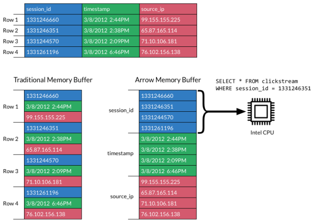
- AWS Glue 3.0 (vectorised readers in C++, SIMD extensions)
- Databricks Photon (C++, vectorized)
- spark-rapids (GPUs)
- oap-project (SIMD, native, Apache Arrow)
- gluten to enable offloading to Clickhouse and Velox

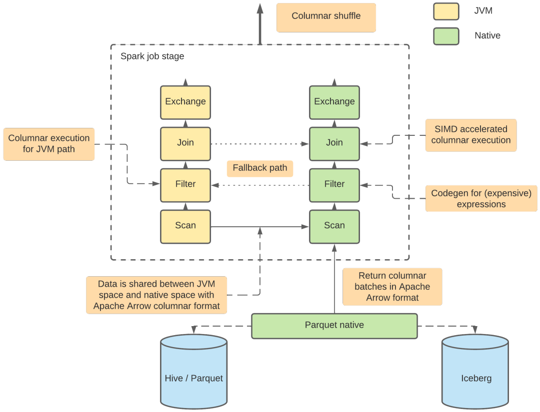




Apache Spark „on steroids“ – Datafusion Comet 2/2

- inspired by Databricks Photon
- native components implemented in Rust
- powered by Apache Datafusion
- uses in-memory columnar format Apache Arrow





Bibliography

Michael Armbrust et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1383–1394.

Holden Karau and Rachel Warren. *High performance Spark: best practices for scaling and optimizing Apache Spark*. " O'Reilly Media, Inc.", 2017.

Javier Ramos. *Apache Spark Internals: Tips and Optimizations*. en. Dec. 2020. URL: https://itnext.io/apache-spark-internals-tips-and-optimizations-8c3cad527ea2 (visited on 03/21/2021).

Spark SQL - DataFrames & Datasets. URL: https://rharshad.com/spark-sql-dataframes-datasets/ (visited on 03/21/2021).