Spark Core - RDD
& Spark Sql - Dataset / DataFrame

Course Esilv 2024

arnaud.nauwynck@gmail.com

This document:
https://github.com/Arnaud-Nauwynck/presentations/
/pres-bigdata/11-Spark-Core-RDD-Sql-Dataset

Outline

(internal) RDD abstract classes in spark-core

public Dataset class in spark-sql

Dataset class use RDD class internally ..

Sql AST parser, Expression, LogicalPlan

DataFrame = Dataset<Row>

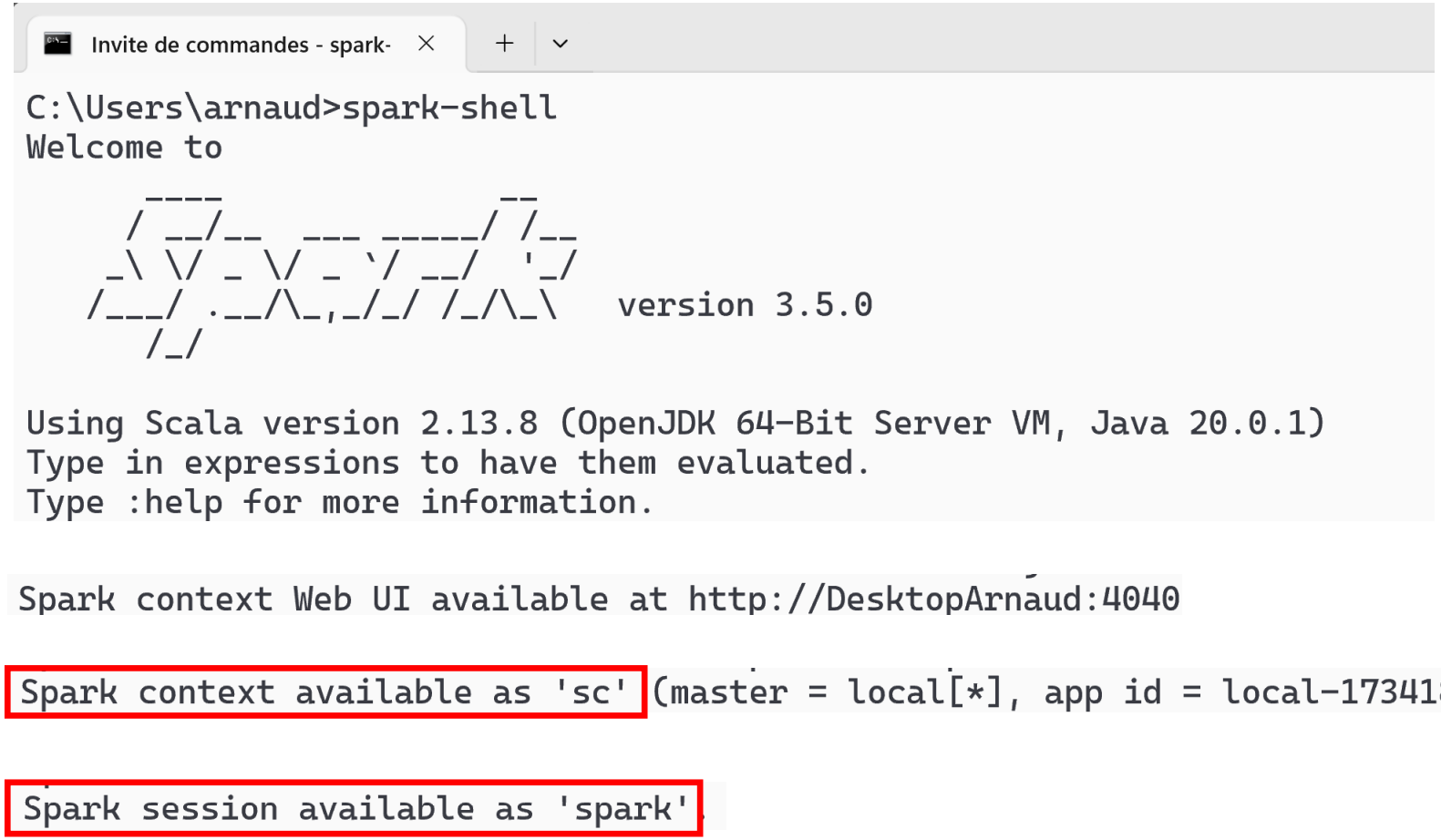
./bin/spark-shell

launch spark-shell
from terminal

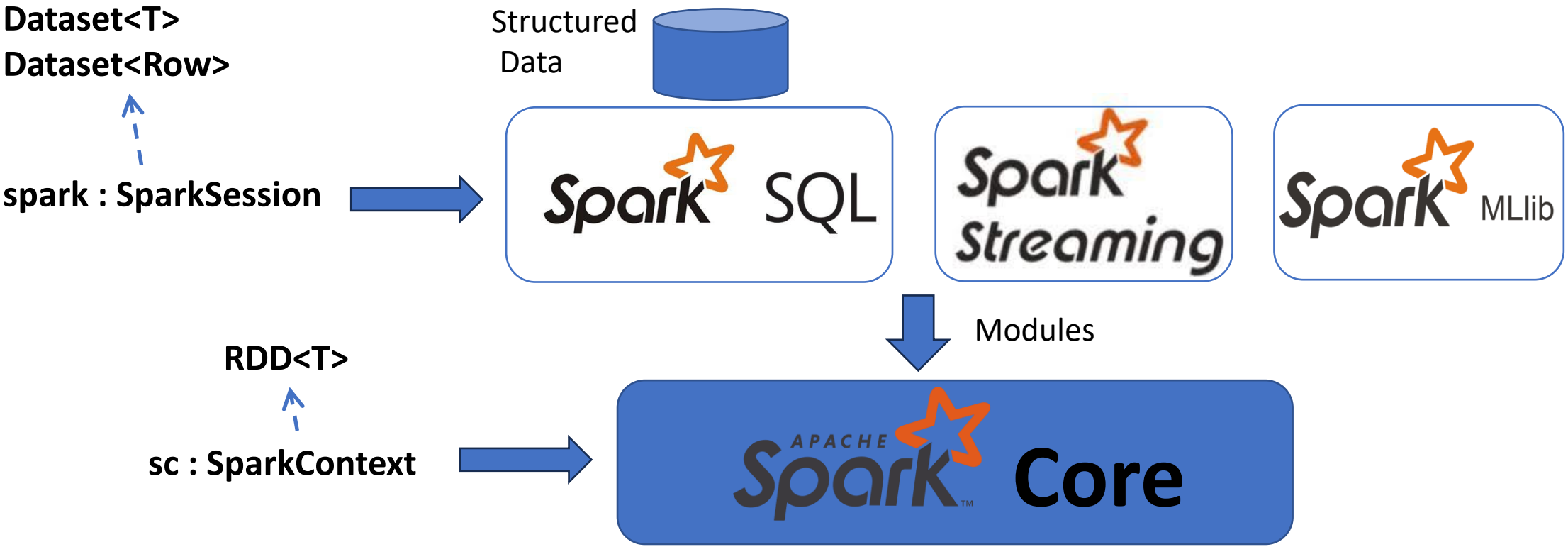
```
C:\Users\arnaud>spark-shell
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.5.0
      /_/

Using Scala version 2.13.8 (OpenJDK 64-Bit Server VM, Java 20.0.1)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context Web UI available at http://DesktopArnaud:4040
Spark context available as 'sc' (master = local[*], app id = local-1734184851521).
Spark session available as 'spark'.

scala> println("Hello spark")
Hello spark

scala>
```

inside spark
type scala code

2 built-in Variables Available : 'sc', 'spark'



sc: SparkContext (in spark-core)
spark : SparkSession (in spark-sql)




```
scala> sc
val res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@5eb39c06

scala> spark
val res1: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@4cecbf3e
```

sc.parallelize(..) vs spark.createDataset(..) ?

```
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)
```

Low-level API
using « **RDD** »

```
scala> val distData = sc.parallelize(data)
distData: org.apache.spark.rdd.RDD[Int] = ParallelCollec

scala> distData.reduce((a,b) => a+b)
res2: Int = 15
```

High-level API
using « **DataSet** »

```
scala> val ds = spark.createDataset(data)
ds: org.apache.spark.sql.Dataset[Int] = [value: int]

scala> ds.reduce((a, b) => a+b)
res4: Int = 15
```

spark-core / spark-sql maven modules

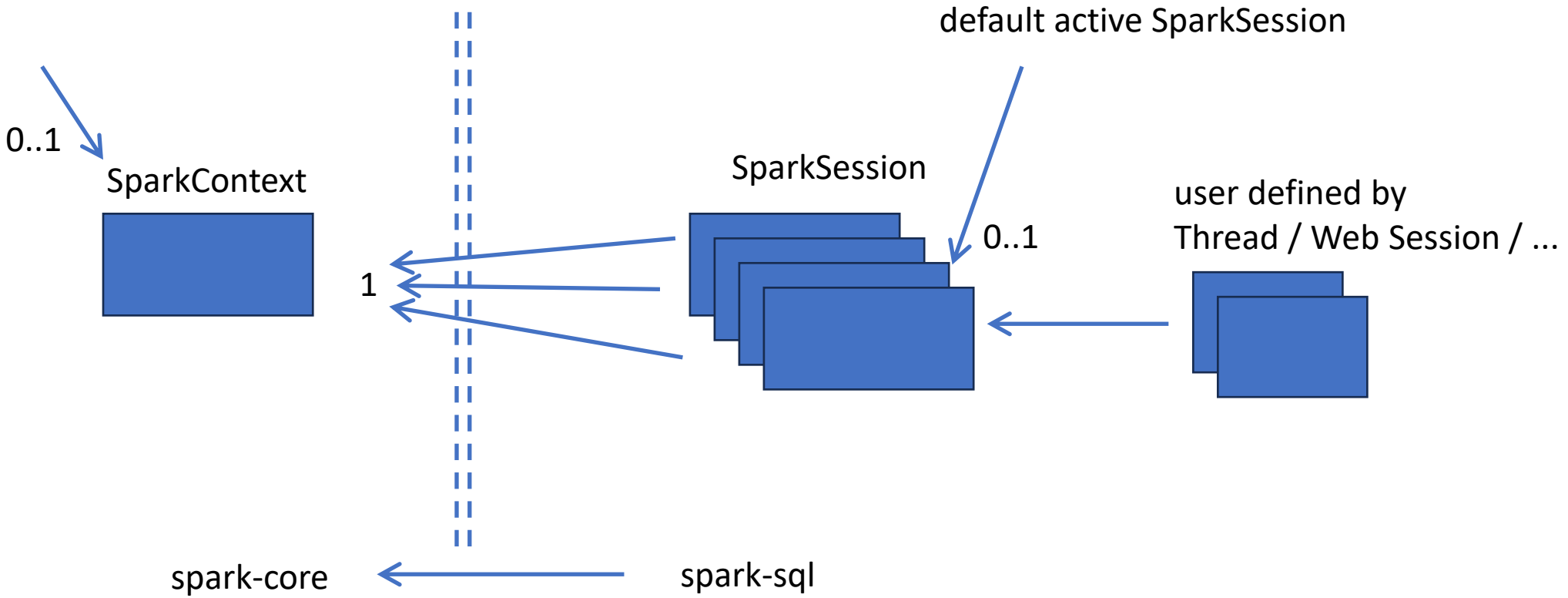
pom.xml

```
<properties>
  <spark.version>3.5.3</spark.version>
  <scala.version>2.13</scala.version>
</properties>

<dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_${scala.version}</artifactId>
      <version>${spark.version}</version>
    </dependency>

    <dependency>
      <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_${scala.version}</artifactId>
      <version>${spark.version}</version>
    </dependency>
</dependencies>
```

SparkConf / (Java)SparkContext / SparkSession

```
SparkConf sparkConf = new SparkConf()
        .setAppName(appName)
        .setMaster(master)
        ;

this.spark = SparkSession.builder()
            .config(sparkConf)
            .getOrCreate();
try {
    this.sc = spark.sparkContext();
    this.jsc = JavaSparkContext.fromSparkContext(sc);
    this.hadoopConf = jsc.hadoopConfiguration();
    this.hadoopFs = FileSystem.get(hadoopConf);
```

1 (singleton) SparkContext
1...N SparkSession
1 Active SparkSession



using Java (Scala) API

on Dataset / RDD

Spark-Java API with Lambda (ambigous method, need cast)

```
FlatMapFunction<String,String> lineToWordIterorFunc = line -> {
    String[] res = line.replace("[,.?!]", " ").replace("  ", " ").split(" ");
    return Arrays.asList(res).iterator();
};
Dataset<String> wordDs = loremIpsum.flatMap(lineToWordIterorFunc, Encoders.STRING());
```

Java API … need « Encoder » (implicit in Scala)

```
loremIpsum.flatMap(line -> line.replace("[,.?!]", " ").replace("  ", " ").split(" "));
```

The method flatMap(Function1<String,TraversableOnce<U>>, Encoder<U>) in the type Dataset<String> is not applicable for the arguments ((<no type> line) -> {})

Press 'F2' for focus

Missing second param in java ..

```
/**
 * (Scala-specific)
 * Returns a new Dataset by first applying a function to all elements of this Dataset,
 * and then flattening the results.
 *
 * @group typedrel
 * @since 1.6.0
 */
def flatMap[U : Encoder](func: T => TraversableOnce[U]): Dataset[U] =
  mapPartitions(_.flatMap(func))
```

Java API … not convenient vs Spark-Scala API

```
loremIpsum.flatMap(line -> line.replace("[,.?!]", " ").replace("  ", " ").split(" "),
        Encoders.STRING());
```

Type mismatch: cannot convert from String[] to TraversableOnce<String>

Press 'F2' for focus

```
loremIpsum.flatMap(
        line -> {
            String res[] = line.replace("[,.?!]", " ").replace("  ", " ").split(" ");
            return Arrays.asList(res).iterator();
        },
        Encoders.STRING());
```

Type mismatch: cannot convert from Iterator<String> to TraversableOnce<String>

2 quick fixes available:

Add cast to 'TraversableOnce<String>'

Change method return type to 'Iterator<String>'

Press 'F2' for focus

using scala.Function1 + inner class … NotSerializableException !

```
46          scala.Function1<String,TraversableOnce<String>> lineToWordScalaFunc =
47                  new AbstractFunction1<String,TraversableOnce<String>>() {
48              @Override
49              public TraversableOnce<String> apply(String line) {
50                  String[] res = line.replace("[,.?!]", " ").replace("  ", " ").split(" ");
51                  return JavaConverters.iterableAsScalaIterableConverter(Arrays.asList(res)).asScala();
52              }
53          };
54          Dataset<String> wordScalaDs = loremIpsum.flatMap(lineToWordScalaFunc, Encoders.STRING());
55          wordScalaDs.show();
56
```

Console × Problems Error Log Debug Shell Search Call Hierarchy

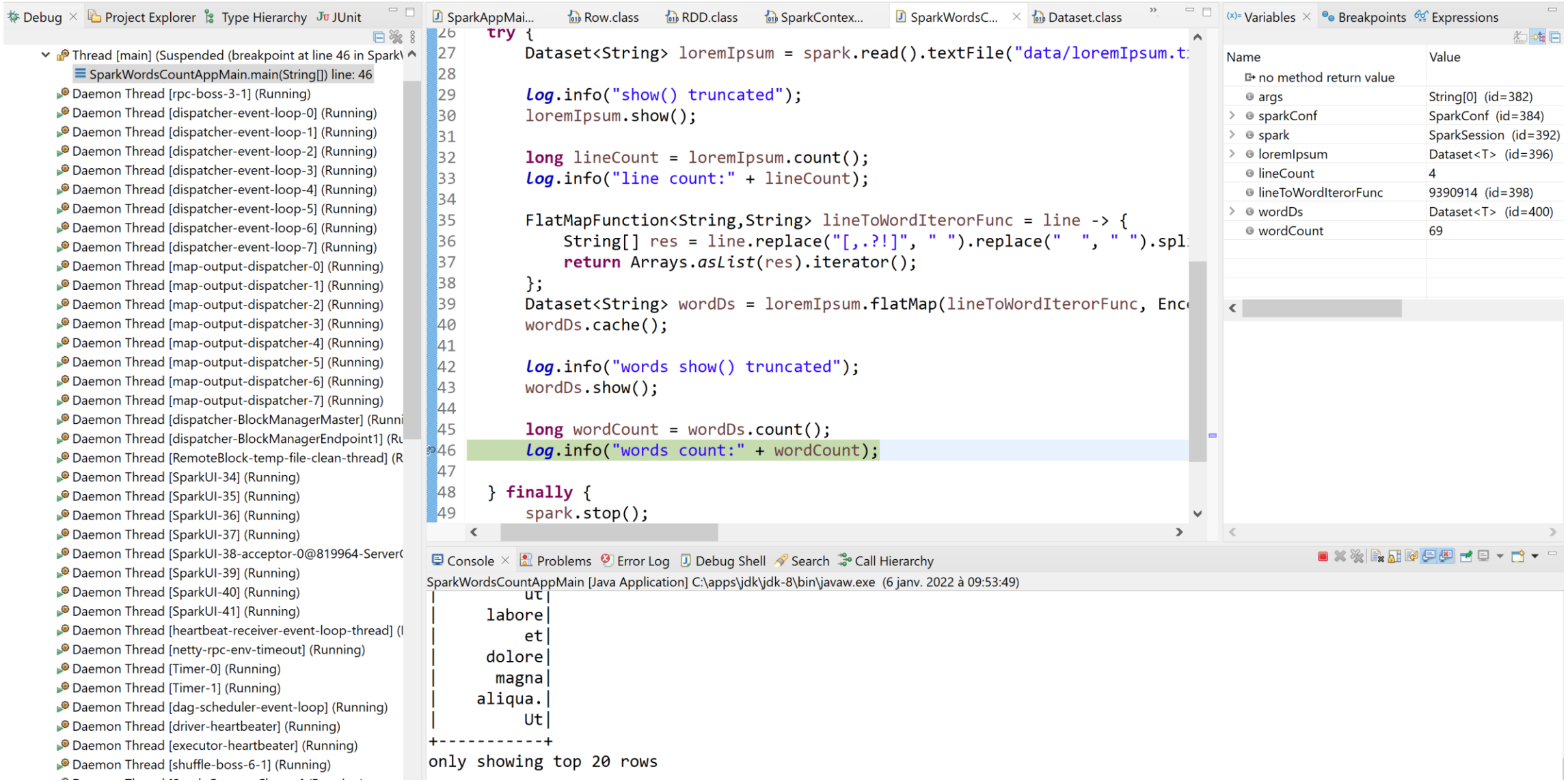
<terminated> SparkWordsCountAppMain [Java Application] C:\apps\jdk\jdk-8\bin\javaw.exe (6 janv. 2022 à 10:15:13 – 10:15:28)

```
10:15:22 INFO fr.an.tests.testspark.SparkWordsCountAppMain: line count:4
Exception in thread "main" org.apache.spark.SparkException: Job aborted due to stage failure: Task not serializable: java.io.NotSerializableExcept
Serialization stack:
        - object not serializable (class: fr.an.tests.testspark.SparkWordsCountAppMain$1, value: <function1>)
        - element of array (index: 0)
        - array (class [Ljava.lang.Object;, size 1)
        - field (class: java.lang.invoke.SerializedLambda, name: capturedArgs, type: class [Ljava.lang.Object;)
        - object (class java.lang.invoke.SerializedLambda, SerializedLambda[capturingClass=class org.apache.spark.sql.Dataset, functionalInterface
        - writeReplace data (class: java.lang.invoke.SerializedLambda)
        - object (class org.apache.spark.sql.Dataset$$Lambda$2247/2630833, org.apache.spark.sql.Dataset$$Lambda$2247/2630833@18b07ae)
        - field (class: org.apache.spark.sql.execution.MapPartitionsExec, name: func, type: interface scala.Function1)
        - object (class org.apache.spark.sql.execution.MapPartitionsExec, MapPartitions org.apache.spark.sql.Dataset$$Lambda$2247/2630833@18b07ae,
```

Isolated top-level class implements Serializable

```
private static class LineToWordFunc extends AbstractFunction1<String,TraversableOnce<String>> implements Serializable {
    private static final long serialVersionUID = 1L;
    @Override
    public TraversableOnce<String> apply(String line) {
        String[] res = line.replace("[,.?!]", " ").replace("  ", " ").split(" ");
        return JavaConverters.iterableAsScalaIterableConverter(Arrays.asList(res)).asScala();
    }
}

private static void testSparkScalaApi(Dataset<String> loremIpsum) {
    scala.Function1<String,TraversableOnce<String>> lineToWordScalaFunc = new LineToWordFunc(); // serializable sub-class
    Dataset<String> wordScalaDs = loremIpsum.flatMap(lineToWordScalaFunc, Encoders.STRING());
    wordScalaDs.show();
}
```

Local[*] Debug from IDE



Advanced Sql / Dataset

```
private static void advancedSparkSqlCode(SparkSession spark) {
    Dataset<Row> ds = spark.sql(
            "SELECT latest.id, latest.col1, latest.col2, joined.col3"
            + " FROM ( SELECT *"
            + "   , row_number() OVER (PARTITION BY id ORDER BY timestamp DESC) as rankNum"
            + "   FROM some_hive_db.input_table"
            + "   WHERE rankNum=1"
            + "   ) latest"
            + " LEFT JOIN department joined ON latest.deptno = joined.deptno"
            )
        // => convert Dataset<Row> -> Dataset<PojoBean>
        .as(Encoders.bean(PojoBean.class))
        // => compute enrich -> Dataset<PojoOutputBean>
        .map((MapFunction<PojoBean,PojoOutputBean>) (pojo -> computePojo2Output(pojo)), Encoders.bean(PojoOutputBean.cla
        // => convert -> Dataset<Row>
        .toDF();

    // optimize for read later
    ds // .repartition(1) .. equivalent to coalesce()
        // .orderBy("id")
        .repartition(2, ds.col("col1"))
        .sortWithinPartitions("id")
        // => write as PARQUET
        .write()
        .format("hive").mode(SaveMode.Overwrite)
        .insertInto("some_hive_db.output_table");
}
```

« Words Count » … Hello world of BigData

loremIpsum.txt

Lorem Ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

```
scala> val ds=spark.read.textFile("c:/data/loremIpsum.txt")
ds: org.apache.spark.sql.Dataset[String] = [value: string]

scala> ds.count  // count lines
res1: Long = 4
```

Dataset.show
default show(20 /*line*/, true /*truncate*/)

```
scala> val ds = spark.read.textFile("c:/data/loremIpsum.txt")
val ds: org.apache.spark.sql.Dataset[String] = [value: string]

scala> ds.show()
+--------------------+
|               value|
+--------------------+
|Lorem Ipsum dolor...|
|Ut enim ad minim ...|
|Duis aute irure d...|
|Excepteur sint oc...|
+--------------------+
```

```
scala> ds.show(2, false)
+-----------------------------------------------------------------------------------------------------------------------------------+
|value                                                                                                                              |
+-----------------------------------------------------------------------------------------------------------------------------------+
|Lorem Ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.|
|Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.                 |
+-----------------------------------------------------------------------------------------------------------------------------------+
only showing top 2 rows


scala>
```

Words Count : flatMap(…).count

```
scala> val wordDs = ds.flatMap(line =>
        line.replaceAll("[,;.:!?]", " ")
                .replaceAll("  ", " ")
                .split(" ")
        )
words: .. Dataset[String] = [value: string]

scala> wordDs.count
res3: Long = 69

scala> wordDs.show(10, false)
```

```
scala> wordDs.count()
val res5: Long = 69

scala> wordDs.show(10)
+-----------+
|      value|
+-----------+
|      Lorem|
|      Ipsum|
|      dolor|
|        sit|
|       amet|
|consectetur|
| adipiscing|
|       elit|
|        sed|
|         do|
+-----------+
only showing top 10 rows
```

Same word Count, using Spark-core RDD ?

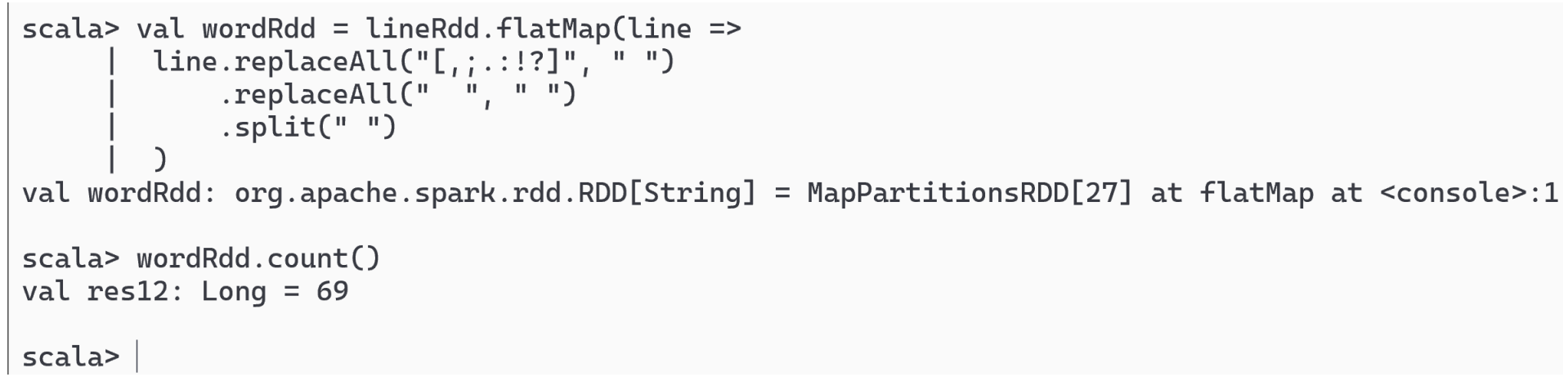
```
scala> val lineRdd = sc.textFile("c:/data/loremIpsum.txt")
val lineRdd: org.apache.spark.rdd.RDD[String] = c:/data/loremIpsum.txt MapPartitionsRDD[26] at textFile at <console>:1

scala> lineRdd.show()
               ^
       error: value show is not a member of org.apache.spark.rdd.RDD[String]

scala> lineRdd.foreach(println(_))
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
Lorem Ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

scala>
```

no "RDD.show()" method !!

not easy to print, even "RDD[String]"

wordCount using RDD
!!! you might never use RDD directly !!

```
scala> val wordRdd = lineRdd.flatMap(line =>
     |  line.replaceAll("[,;.:!?]", " ")
     |      .replaceAll("  ", " ")
     |      .split(" ")
     |  )
val wordRdd: org.apache.spark.rdd.RDD[String] =
MapPartitionsRDD[27] at flatMap at <console>:1

scala> wordRdd.count()
val res12: Long = 69
```



RDD vs Dataset look Similar ?
But look closely

```
scala> ds
val res14: org.apache.spark.sql.Dataset[String] = [value: string]
```

```
scala> val wordDs = ds.flatMap(line =>
     |  line.replaceAll("[,;.:!?]", " ")
     |       .replaceAll("  ", " ")
     |       .split(" ")
     |  )
val wordDs: org.apache.spark.sql.Dataset[String] = [value: string]
```

using spark-sql Dataset,
transformed dataset is still a "Dataset"

```
scala> lineRdd
val res13: org.apache.spark.rdd.RDD[String] = c:/data/loremIpsum.txt MapPartitionsRDD[26] at textFile at <console>:1
```

```
scala> val wordRdd = lineRdd.flatMap(line =>
     |  line.replaceAll("[,;.:!?]", " ")
     |       .replaceAll("  ", " ")
     |       .split(" ")
     |  )
val wordRdd: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[27] at flatMap at <console>:1
```

using spark-core RDD,
transformed RDD is an instance of sub-class "MapPartitionsRDD"

Dataset vs RDD

Dataset and RDD are both lazy evaluated

BUT

RDD eval => just call the overriden method of sub-class (parent class is abstract)

Dataset action execution => need to analyze query, transform logical to physical
=> optimize => generate code => ... then finally call corresponding low-level RDD

RDD is low-level, Dataset is high-level using RDD internally

What are Dataset/RDD ?
How they work ?

RDD = ?

Resilient

Distributed

Dataset

R.D.D.

Resilient = can resist failure

…. can be recomputed on demand

Distributed = split in partitions, distributed over executors

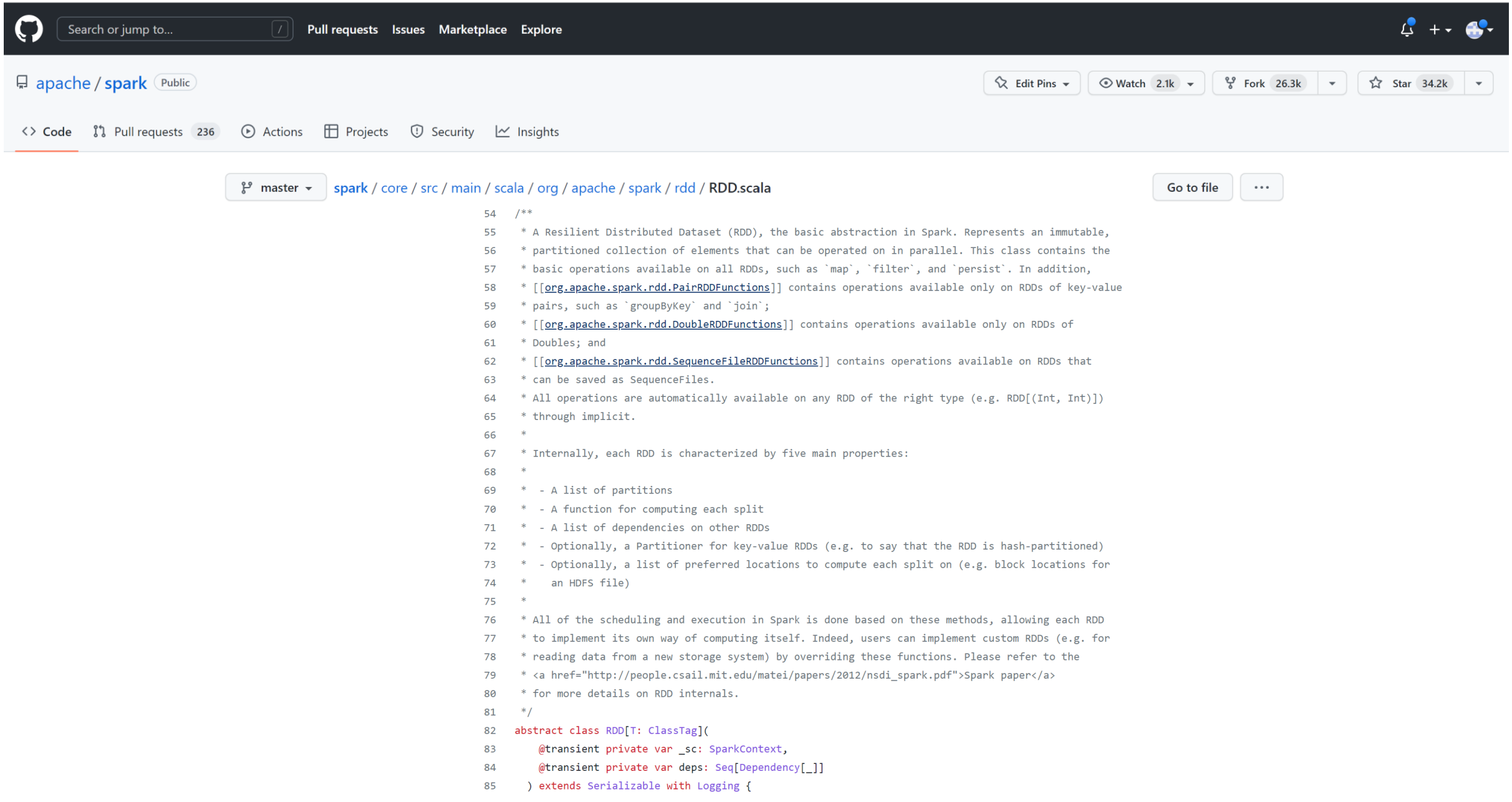
Dataset = … RDD result ends up to be a

ImmutableList<ImmutableData>

… from a function definition + inputs

List<Data> computeFunc() { … }

source code http://github.com/apache/spark




```scala
/**
 * A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable,
 * partitioned collection of elements that can be operated on in parallel. This class contains the
 * basic operations available on all RDDs, such as `map`, `filter`, and `persist`. In addition,
 * [[org.apache.spark.rdd.PairRDDFunctions]] contains operations available only on RDDs of key-value
 * pairs, such as `groupByKey` and `join`;
 * [[org.apache.spark.rdd.DoubleRDDFunctions]] contains operations available only on RDDs of
 * Doubles; and
 * [[org.apache.spark.rdd.SequenceFileRDDFunctions]] contains operations available on RDDs that
 * can be saved as SequenceFiles.
 * All operations are automatically available on any RDD of the right type (e.g. RDD[(Int, Int)])
 * through implicit.
 *
 * Internally, each RDD is characterized by five main properties:
 *
 *  - A list of partitions
 *  - A function for computing each split
 *  - A list of dependencies on other RDDs
 *  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
 *  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for
 *    an HDFS file)
 *
 * All of the scheduling and execution in Spark is done based on these methods, allowing each RDD
 * to implement its own way of computing itself. Indeed, users can implement custom RDDs (e.g. for
 * reading data from a new storage system) by overriding these functions. Please refer to the
 * <a href="http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf">Spark paper</a>
 * for more details on RDD internals.
 */
abstract class RDD[T: ClassTag](
    @transient private var _sc: SparkContext,
    @transient private var deps: Seq[Dependency[_]]
  ) extends Serializable with Logging {
```

RDD /** javadoc... */ (1/3)

A **Resilient Distributed Dataset** (RDD),
the **basic abstraction** in Spark.

Represents
an **immutable**,
partitioned collection of elements
that can be operated on **in parallel**.

This class contains the basic operations available on all RDDs,
such as `map`, `filter`, and `persist`.
In addition, PairRDDFunctions (..) of key-value pairs,
(..contains) `groupByKey` and `join` (..)

RDD /** javadoc... */ (2/3)

Internally, each RDD is characterized by :

- A **list of partitions**

- A **function for computing** each split

- A list of **dependencies on other RDDs**

- Optionally, a **Partitioner**
- Optionally, a list of **preferred locations**

RDD /** javadoc... */ (3/3)

All (..) in Spark is done based on these methods,
allowing each RDD **to implement** its own **way of computing** itself.

Indeed, users **can implement** custom RDDs
(e.g. for reading data from a new storage system)
by **overriding** these functions.

Please refer to the
Spark paper
for more details on RDD internals.

RDD Paper

A Fault-Tolerant Abstraction For In-Memory Cluster computing

To achieve fault tolerance efficiently,
RDDs provide a restricted form of shared memory,
based on coarse-grained transformations

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.,* between two MapReduce jobs) is to write it to an external stable storage system, *e.g.,* a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.,* looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.,* to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.,* cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.,* map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1] Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

RDD Abstract methods

```scala
// ======================================================================
// Methods that should be implemented by subclasses of RDD
// ======================================================================

/**
 * :: DeveloperApi ::
 * Implemented by subclasses to compute a given partition.
 */
@DeveloperApi
def compute(split: Partition, context: TaskContext): Iterator[T]

/**
 * Implemented by subclasses to return the set of partitions in this RDD. This method will only
 * be called once, so it is safe to implement a time-consuming computation in it.
 *
 * The partitions in this array must satisfy the following property:
 *   `rdd.partitions.zipWithIndex.forall { case (partition, index) => partition.index == index }`
 */
protected def getPartitions: Array[Partition]

/**
 * Implemented by subclasses to return how this RDD depends on parent RDDs. This method will only
 * be called once, so it is safe to implement a time-consuming computation in it.
 */
protected def getDependencies: Seq[Dependency[_]] = deps

/**
 * Optionally overridden by subclasses to specify placement preferences.
 */
protected def getPreferredLocations(split: Partition): Seq[String] = Nil

/** Optionally overridden by subclasses to specify how they are partitioned. */
@transient val partitioner: Option[Partitioner] = None
```

abstract class org.apache.spark.rdd.RDD

```scala
/**
 * A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable,
 * partitioned collection of elements that can be operated on in parallel. This class contains the
 * basic operations available on all RDDs, such as `map`, `filter`, and `persist`. In addition,
 * [[org.apache.spark.rdd.PairRDDFunctions]] contains operations available only on RDDs of key-value
 * pairs, such as `groupByKey` and `join`;
 * [[org.apache.spark.rdd.DoubleRDDFunctions]] contains operations available only on RDDs of
 * Doubles; and
 * [[org.apache.spark.rdd.SequenceFileRDDFunctions]] contains operations available on RDDs that
 * can be saved as SequenceFiles.
 * All operations are automatically available on any RDD of the right type (e.g. RDD[(Int, Int)])
 * through implicit.
 *
 * Internally, each RDD is characterized by five main properties:
 *
 *  - A list of partitions
 *  - A function for computing each split
 *  - A list of dependencies on other RDDs
 *  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
 *  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for
 *    an HDFS file)
 *
 * All of the scheduling and execution in Spark is done based on these methods, allowing each RDD
 * to implement its own way of computing itself. Indeed, users can implement custom RDDs (e.g. for
 * reading data from a new storage system) by overriding these functions. Please refer to the
 * <a href="http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf">Spa
 * for more details on RDD internals.
 */
abstract class RDD[T: ClassTag](
    @transient private var _sc: SparkContext,
    @transient private var deps: Seq[Dependency[_]]
  ) extends Serializable with Logging {
```

```scala
  // ========================================================================
  // Methods that should be implemented by subclasses of RDD
  // ========================================================================

  /**
   * :: DeveloperApi ::
   * Implemented by subclasses to compute a given partition.
   */
  @DeveloperApi
  def compute(split: Partition, context: TaskContext): Iterator[T]
```

RDD

- conf() : SparkConf
- A compute(Partition, TaskContext) : Iterator<T
- A getPartitions() : Partition[]
- getDependencies() : Seq<Dependency<?>
- getPreferredLocations(Partition) : Seq<Strin
- partitioner() : Option<Partitioner>
- sparkContext() : SparkContext
- id() : int
- name() : String

- F dependencies() : Seq<Dependency<?>>
- F partitions() : Partition[]
- F getNumPartitions() : int
- F preferredLocations(Partition) : Seq<String>
- F iterator(Partition, TaskContext) : Iterator<T>
- getNarrowAncestors() : Seq<RDD<?>>
- computeOrReadCheckpoint(Partition, TaskContext)
- getOrCompute(Partition, TaskContext) : Iterator<T>

- isEmpty() : boolean
- collectPartitions() : Object[]
- checkpoint() : void
- localCheckpoint() : RDD<T>
- barrier() : RDDBarrier<T>
- doCheckpoint() : void
- markCheckpointed() : void
- clearDependencies() : void
- toJavaRDD() : JavaRDD<T>
- isBarrier() : boolean
- isCheckpointed() : boolean
- isCheckpointedAndMaterialized() : boolean
- isLocallyCheckpointed() : boolean
- isReliablyCheckpointed() : boolean
- getCheckpointFile() : Option<String>
- creationSite() : CallSite
- scope() : Option<RDDOperationScope>
- getCreationSite() : String
- elementClassTag() : ClassTag<T>
- checkpointData() : Option<RDDCheckpointData<T>>
- checkpointData_$eq(Option<RDDCheckpointData<T>>) :
- firstParent(ClassTag<U>) <U> : RDD<U>
- parent(int, ClassTag<U>) <U> : RDD<U>
- context() : SparkContext
- retag(Class<T>) : RDD<T>
- retag(ClassTag<T>) : RDD<T>

- persist() : RDD<T>
- cache() : RDD<T>
- persist(StorageLevel) : RDD<T>
- unpersist(boolean) : RDD<T>
- unpersist$default$1() : boolean
- getStorageLevel() : StorageLevel

- withScope(Function0<U>) <U> : U
- map(Function1<T, U>, ClassTag<U>) <U> : RDD<U>
- flatMap(Function1<T, TraversableOnce<U>>, ClassTag<
- filter(Function1<T, Object>) : RDD<T>
- distinct(int, Ordering<T>) : RDD<T>
- distinct$default$2(int) : Ordering<T>
- repartition(int, Ordering<T>) : RDD<T>
- repartition$default$2(int) : Ordering<T>
- coalesce(int, boolean, Option<PartitionCoalescer>, Orde
- coalesce$default$2() : boolean
- coalesce$default$3() : Option<PartitionCoalescer>
- coalesce$default$4(int, boolean, Option<PartitionCoales
- sample(boolean, double, long) : RDD<T>
- sample$default$3() : long
- randomSplit(double[], long) : RDD<T>[]
- randomSplit$default$2() : long
- randomSampleWithRange(double, double, long) : RDD<
- takeSample(boolean, int, long) : Object
- union(RDD<T>) : RDD<T>
- $plus$plus(RDD<T>) : RDD<T>
- sortBy(Function1<T, K>, boolean, int, Ordering<K>, Clas
- sortBy$default$2() <K> : boolean
- sortBy$default$3() <K> : int
- intersection(RDD<T>) : RDD<T>
- intersection(RDD<T>, Partitioner, Ordering<T>) : RDD<T
- intersection(RDD<T>, int) : RDD<T>
- intersection$default$3(RDD<T>, Partitioner) : Ordering<
- cartesian(RDD<U>, ClassTag<U>) <U> : RDD<Tuple2<T
- groupBy(Function1<T, K>, ClassTag<K>) <K> : RDD<Tu
- groupBy(Function1<T, K>, int, ClassTag<K>) <K> : RDD<
- groupBy(Function1<T, K>, Partitioner, ClassTag<K>, Ord
- groupBy$default$4(Function1<T, K>, Partitioner) <K> :
- pipe(String) : RDD<String>
- pipe(String, Map<String, String>) : RDD<String>
- pipe(Seq<String>, Map<String, String>, Function1<Func
- mapPartitions$default$2() <U> : boolean
- mapPartitionsWithIndexInternal(Function2<Object, Iterator
- mapPartitionsInternal(Function1<Iterator<T>, Iterator<U>
- mapPartitionsInternal$default$2() <U> : boolean
- mapPartitionsWithIndex(Function2<Object, Iterator<T>, Ite
- mapPartitionsWithIndex(Function2<Object, Iterator<T>, Ite
- mapPartitionsWithIndexInternal$default$2() <U> : boolean
- mapPartitionsWithIndexInternal$default$3() <U> : boolean
- mapPartitionsWithIndex$default$2() <U> : boolean
- zip(RDD<U>, ClassTag<U>) <U> : RDD<Tuple2<T, U>>
- zipPartitions(RDD<B>, boolean, Function2<Iterator<T>, Ite
- zipPartitions(RDD<B>, Function2<Iterator<T>, Iterator<B>
- zipPartitions(RDD<B>, RDD<C>, boolean, Function3<Itera
- zipPartitions(RDD<B>, RDD<C>, Function3<Iterator<T>, It
- zipPartitions(RDD<B>, RDD<C>, RDD<D>, boolean, Funct
- zipPartitions(RDD<B>, RDD<C>, RDD<D>, Function4<Iter
- foreach(Function1<T, BoxedUnit>) : void
- foreachPartition(Function1<Iterator<T>, BoxedUnit>) : voi
- toLocalIterator() : Iterator<T>
- collect(PartialFunction<T, U>, ClassTag<U>) <U> : RDD<U
- subtract(RDD<T>) : RDD<T>
- subtract(RDD<T>, int) : RDD<T>
- subtract(RDD<T>, Partitioner, Ordering<T>) : RDD<T>
- subtract$default$3(RDD<T>, Partitioner) : Ordering<T>
- reduce(Function2<T, T, T>) : T
- treeReduce(Function2<T, T, T>, int) : T
- treeReduce$default$2() : int
- fold(T, Function2<T, T, T>) : T
- aggregate(U, Function2<U, T, U>, Function2<U, U, U>, Cla
- treeAggregate(U, Function2<U, T, U>, Function2<U, U, U>
- treeAggregate$default$4(U) <U> : int
- countApprox(long, double) : PartialResult<BoundedDouble
- countApprox$default$2() : double
- countByValue(Ordering<T>) : Map<T, Object>
- countByValue$default$1() : Ordering<T>
- countByValueApprox(long, double, Ordering<T>) : PartialR
- countByValueApprox$default$2() : double
- countByValueApprox$default$3(long, double) : Ordering<T
- countApproxDistinct(int, int) : long
- countApproxDistinct(double) : long
- countApproxDistinct$default$1() : double
- take(int) : Object

RDD operators…

- takeSample$default$3() : long
- top(int, Ordering<T>) : Object
- takeOrdered(int, Ordering<T>) : Object
- max(Ordering<T>) : T
- min(Ordering<T>) : T
- saveAsTextFile(String) : void
- saveAsTextFile(String, Class<? extends CompressionCodec
- saveAsObjectFile(String) : void
- keyBy(Function1<T, K>) <K> : RDD<Tuple2<K, T>>
- distinct() : RDD<T>
- glom() : RDD<Object>
- collect() : Object
- count() : long
- zipWithIndex() : RDD<Tuple2<T, Object>>
- zipWithUniqueId() : RDD<Tuple2<T, Object>>
- first() : T

Sub-classes Hierarchy extending RDD

- RDD<T>
  - BaseRRDD<T, U>
    - PairwiseRRDD<T>
    - RRDD<T>
    - StringRRDD<T>
  - BlockRDD<T>
  - CartesianRDD<T, U>
  - CheckpointRDD<T>
    - LocalCheckpointRDD<T>
    - ReliableCheckpointRDD<T>
  - CoalescedRDD<T>
  - CoGroupedRDD<K>
  - ContinuousCoalesceRDD
  - ContinuousDataSourceRDD
  - ContinuousShuffleReadRDD
  - ContinuousWriteRDD
  - DataSourceRDD
  - EmptyRDD<T>
  - EmptyRDDWithPartitions
  - FileScanRDD
  - HadoopMapPartitionsWithSplitRDD<U, T>
  - HadoopRDD<K, V>
  - JdbcRDD<T>
  - JDBCRDD
  - MapPartitionsRDD<U, T>
  - NewHadoopMapPartitionsWithSplitRDD<U, T>
  - NewHadoopRDD<K, V>
    - BinaryFileRDD<T>
    - WholeTextFileRDD
  - PairwiseRDD
  - ParallelCollectionRDD<T>
  - PartitionerAwareUnionRDD<T>
  - PartitionPruningRDD<T>
  - PartitionwiseSampledRDD<T, U>
  - PipedRDD<T>
  - PythonRDD
  - ShuffledRDD<K, V, C>
  - ShuffledRowRDD
  - SQLExecutionRDD
  - StateStoreRDD<T, U>
  - SubtractedRDD<K, V, W>
  - UnionRDD<T>
  - ZippedPartitionsBaseRDD<V>
    - StateStoreAwareZipPartitionsRDD<A, B, V>
    - ZippedPartitionsRDD2<A, B, V>
    - ZippedPartitionsRDD3<A, B, C, V>
    - ZippedPartitionsRDD4<A, B, C, D, V>
  - ZippedWithIndexRDD<T>

1 algorithm / transformation => 1 RDD Sub-class

Example: rdd.map(func) or rdd.flatMap(func)

```scala
/**
 * Return a new RDD by applying a function to all elements of this RDD.
 */
def map[U: ClassTag](f: T => U): RDD[U] = withScope {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[U, T]( prev = this, (_, _, iter) => iter.map(cleanF))
}

/**
 *  Return a new RDD by first applying a function to all elements of this
 *  RDD, and then flattening the results.
 */
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U] = withScope {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[U, T]( prev = this, (_, _, iter) => iter.flatMap(cleanF))
}
```

```scala
/**
 * An RDD that applies the provided function to every partition of the parent RDD.
 *
 * @param prev the parent RDD.
 * @param f The function used to map a tuple of (TaskContext, partition index, input iterator) to
 *          an output iterator.
 * @param preservesPartitioning Whether the input function preserves the partitioner, which should
 *                              be `false` unless `prev` is a pair RDD and the input function
 *                              doesn't modify the keys.
 * @param isFromBarrier Indicates whether this RDD is transformed from an RDDBarrier, a stage
 *                      containing at least one RDDBarrier shall be turned into a barrier stage.
 * @param isOrderSensitive whether or not the function is order-sensitive. If it's order
 *                         sensitive, it may return totally different result when the input order
 *                         is changed. Mostly stateful functions are order-sensitive.
 */
private[spark] class MapPartitionsRDD[U: ClassTag, T: ClassTag](
    var prev: RDD[T],
    f: (TaskContext, Int, Iterator[T]) => Iterator[U],  // (TaskContext, partition index, iterator)
    preservesPartitioning: Boolean = false,
    isFromBarrier: Boolean = false,
    isOrderSensitive: Boolean = false)
  extends RDD[U](prev) {
```

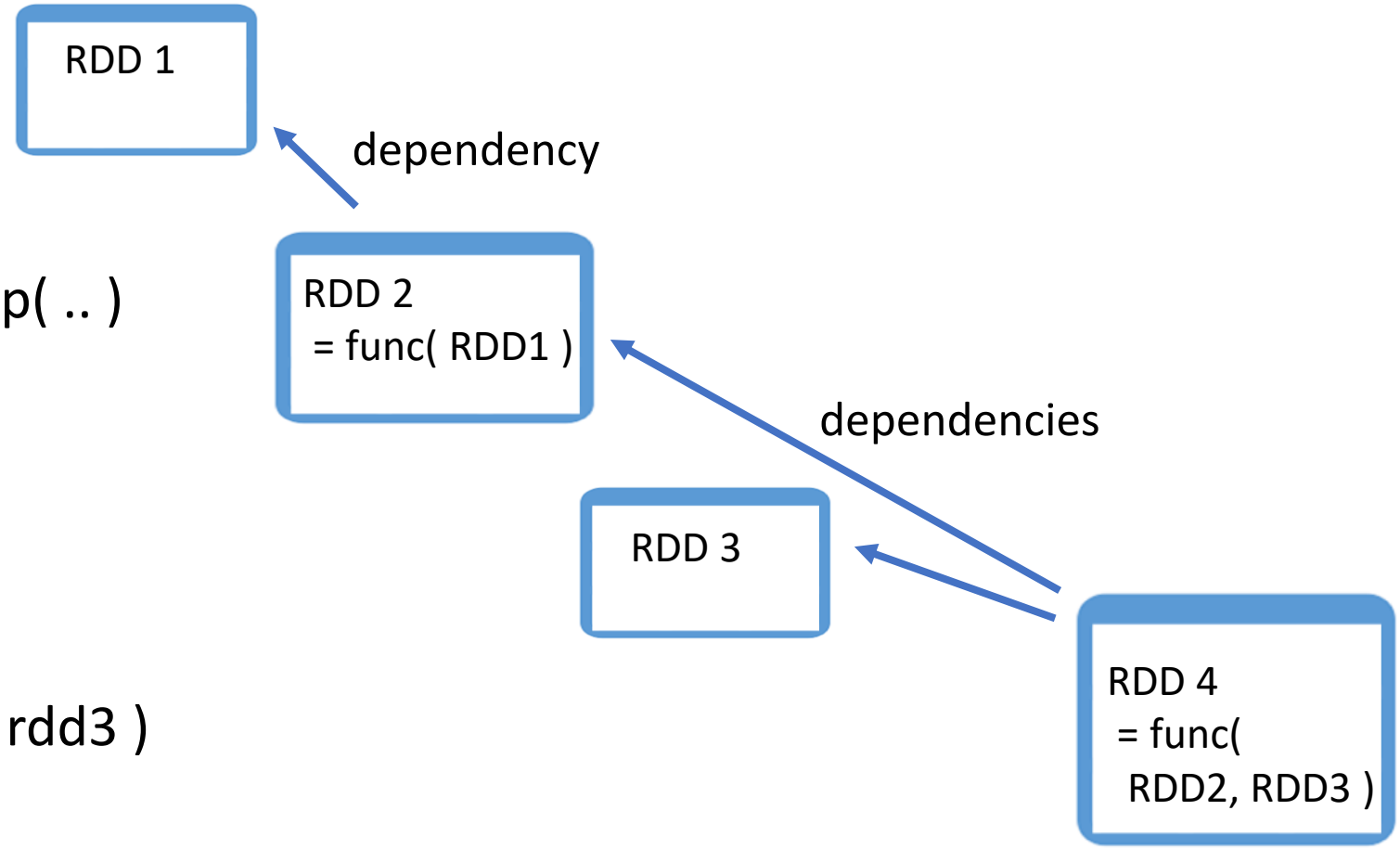
Call transform function => Create new RDD (linked)

RDD rdd1 = … read

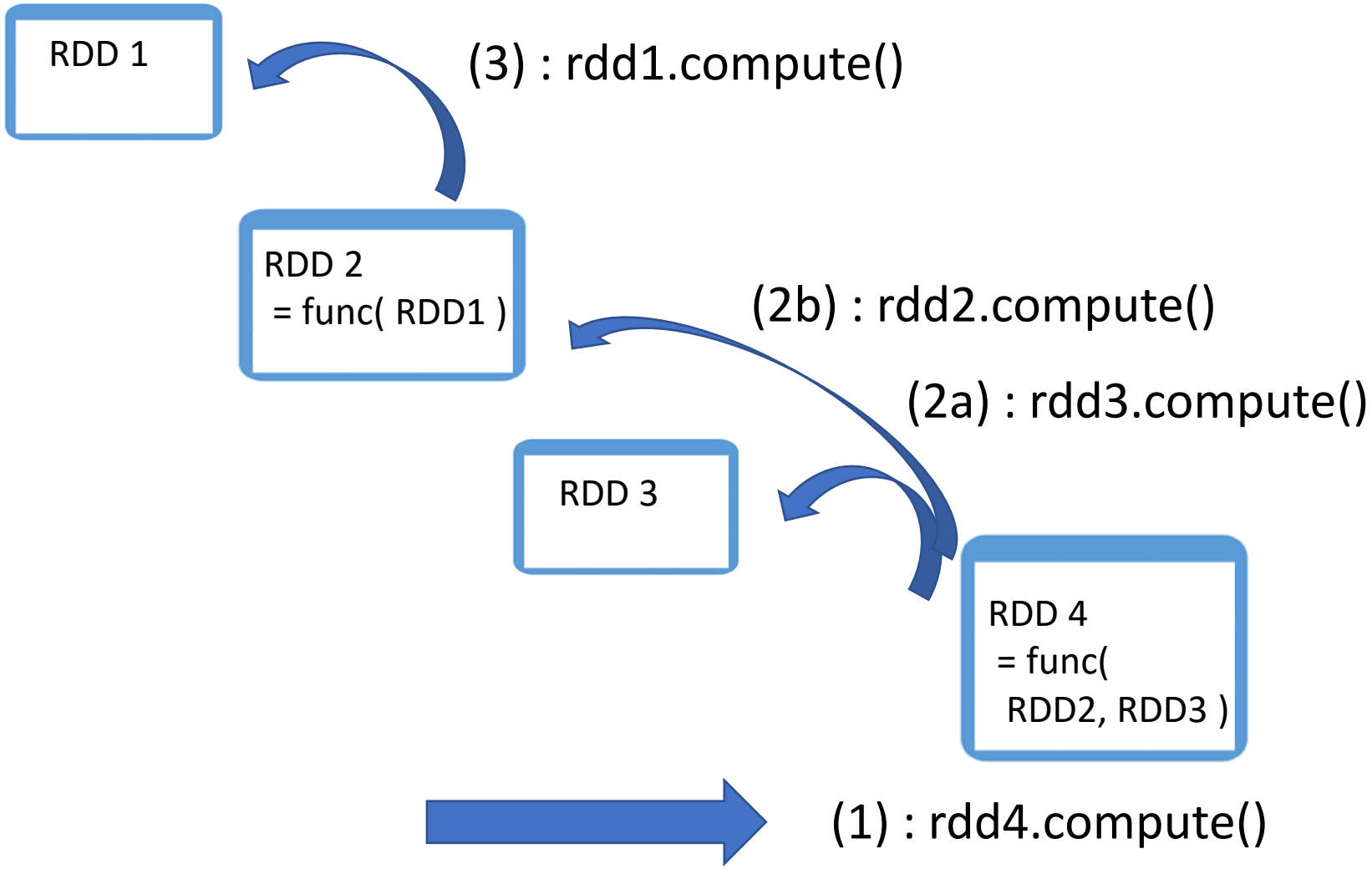
RDD rdd2 = rdd1 .map(..)

RDD rdd3 = .. read

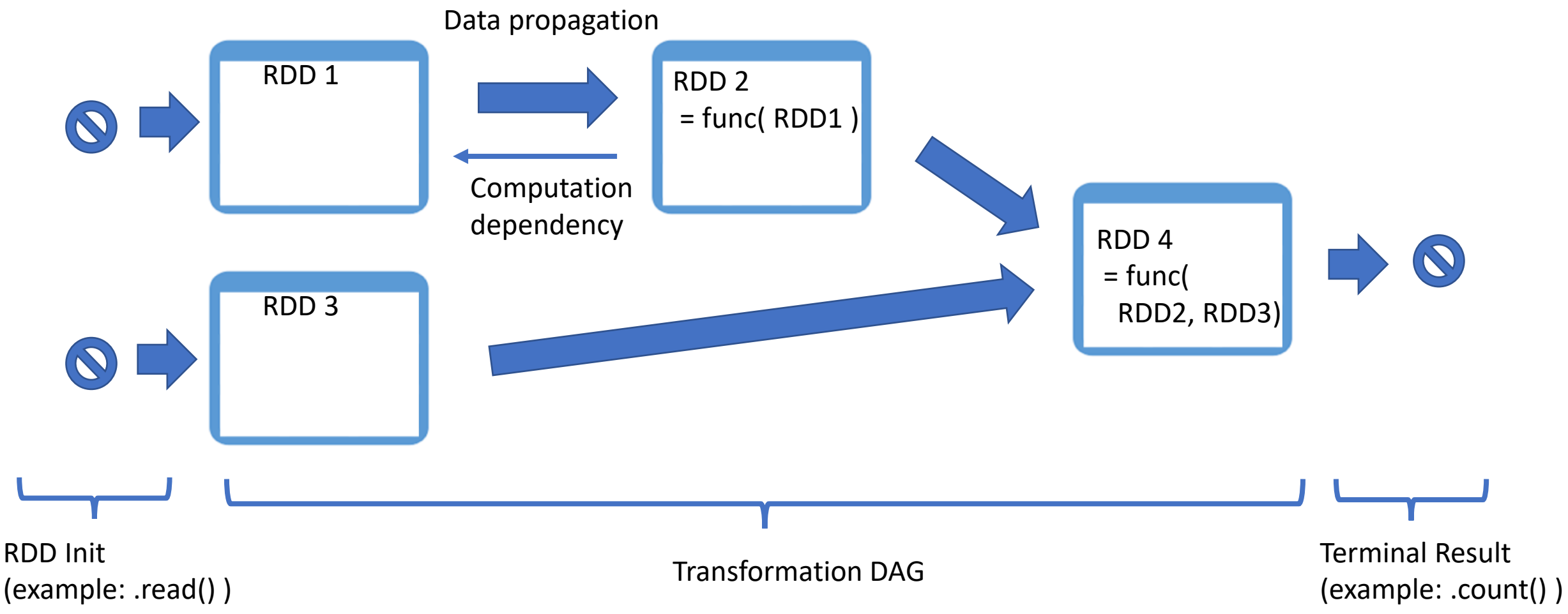
RDD rdd4 = rdd2 .join(rdd3)



Call compute() => … dependency.compute()



RDD Dependencies : DAG (Directed Acyclic Graph)

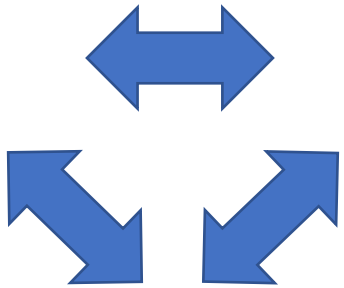


3 equivalent formalisms:
SSA create Api, Expression Algebra, DAG

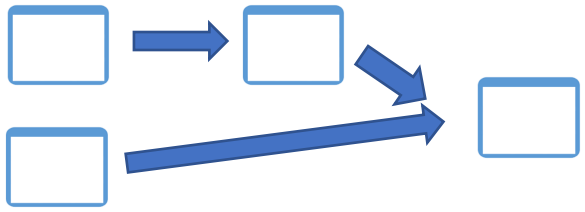
SSA = Single State Assignments

RDD API

```
RDD rdd1 = … read
RDD rdd2   =  rdd1 .map( .. )
RDD rdd3 = .. read
RDD rdd4 = rdd2 .join( rdd3 )
```



DAG

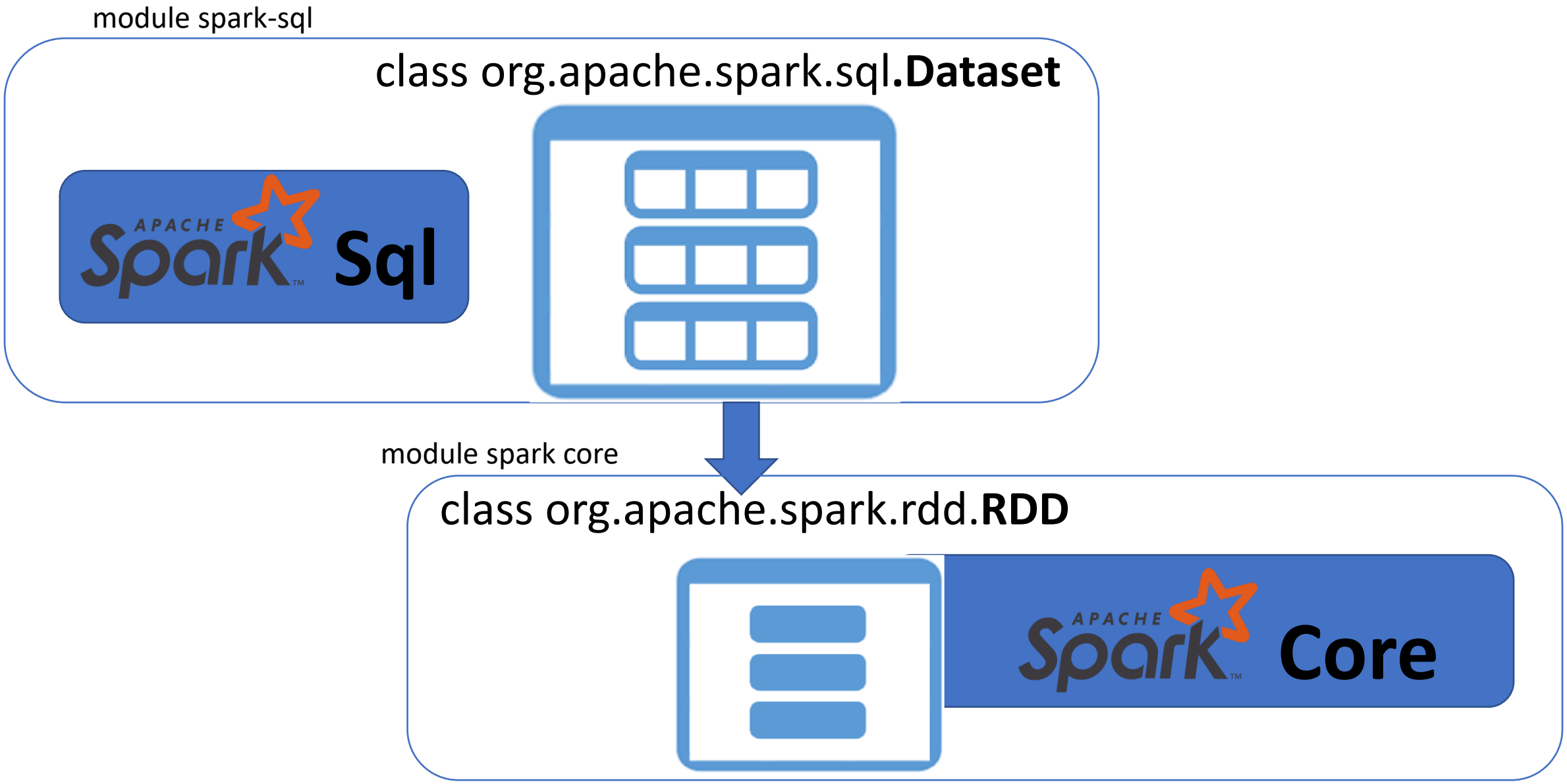


Expression Algebra, Sql

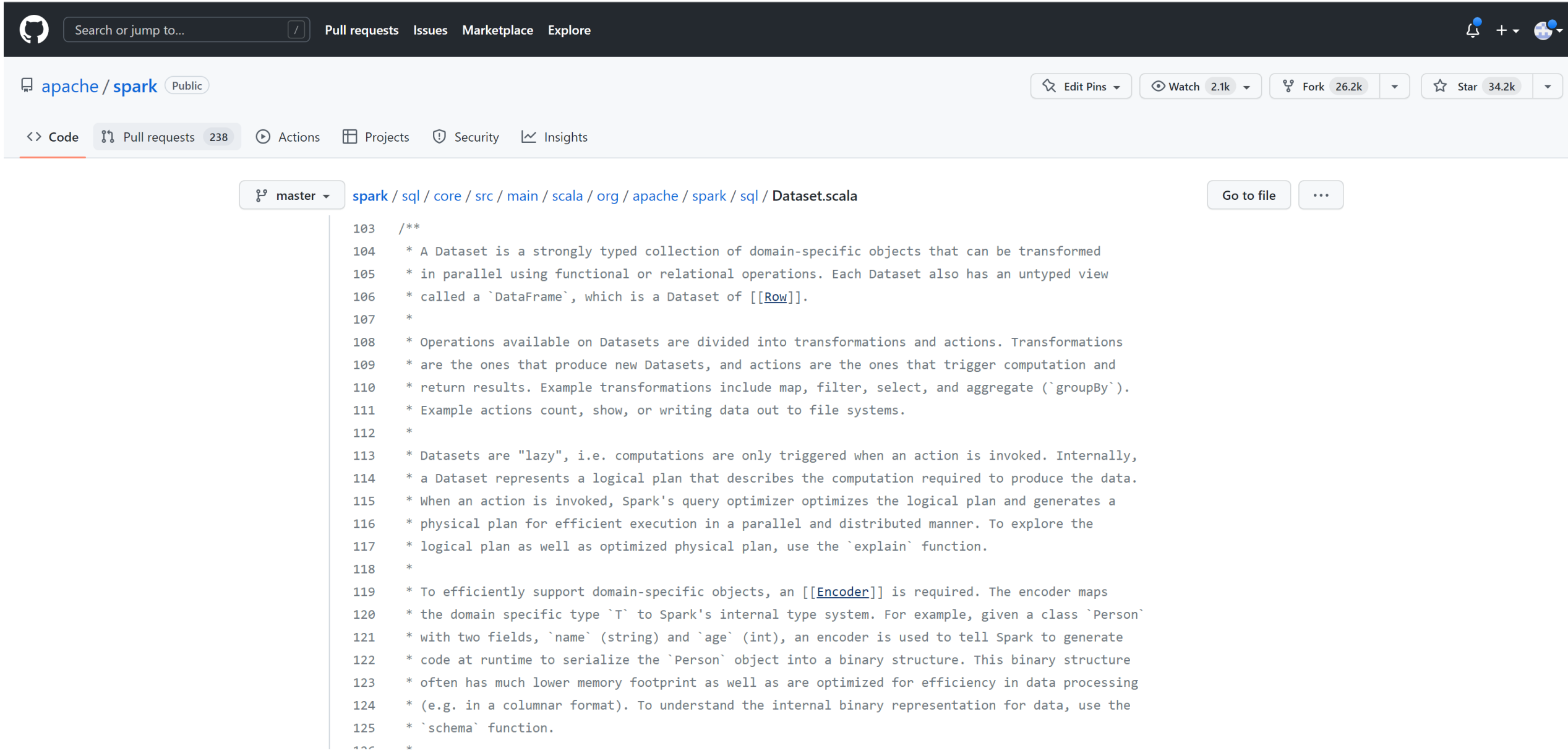
```
SELECT  map(t1) FROM Table1 t1
JOIN Table2 t2 on ..

new JoinRDD(
   new MapRDD( readRDD(table1) ),
   readRDD(table2)
   )
```

RDD … OK
so What is a Dataset ?

DataSet = sql wrapper for RDD, in module spark-sql



Dataset source doc




```
103  /**
104   * A Dataset is a strongly typed collection of domain-specific objects that can be transformed
105   * in parallel using functional or relational operations. Each Dataset also has an untyped view
106   * called a `DataFrame`, which is a Dataset of [[Row]].
107   *
108   * Operations available on Datasets are divided into transformations and actions. Transformations
109   * are the ones that produce new Datasets, and actions are the ones that trigger computation and
110   * return results. Example transformations include map, filter, select, and aggregate (`groupBy`).
111   * Example actions count, show, or writing data out to file systems.
112   *
113   * Datasets are "lazy", i.e. computations are only triggered when an action is invoked. Internally,
114   * a Dataset represents a logical plan that describes the computation required to produce the data.
115   * When an action is invoked, Spark's query optimizer optimizes the logical plan and generates a
116   * physical plan for efficient execution in a parallel and distributed manner. To explore the
117   * logical plan as well as optimized physical plan, use the `explain` function.
118   *
119   * To efficiently support domain-specific objects, an [[Encoder]] is required. The encoder maps
120   * the domain specific type `T` to Spark's internal type system. For example, given a class `Person`
121   * with two fields, `name` (string) and `age` (int), an encoder is used to tell Spark to generate
122   * code at runtime to serialize the `Person` object into a binary structure. This binary structure
123   * often has much lower memory footprint as well as are optimized for efficiency in data processing
124   * (e.g. in a columnar format). To understand the internal binary representation for data, use the
125   * `schema` function.
```

class Dataset /** javadoc.. */ (1/4)

A strongly typed **collection of domain-specific objects**

that can be **transformed in parallel**

using **functional or relational operations**

Associated untyped view: DataFrame = Dataset<Row>

Dataset /** javadoc.. */ (2/4)

Operations on Datasets:

Transformations = **produce new Datasets**

Actions = **trigger computation and return results**.

Example transformations : map, filter, select, groupBy ...
Example actions : count, show, write ...

Datasets are "lazy",
i.e. computations are only triggered **when an action is invoked**.

Transformation… « produce » new Dataset
NO UPDATE method
Dataset are computable / « **immutable** »

dataset. <noSetter> ();

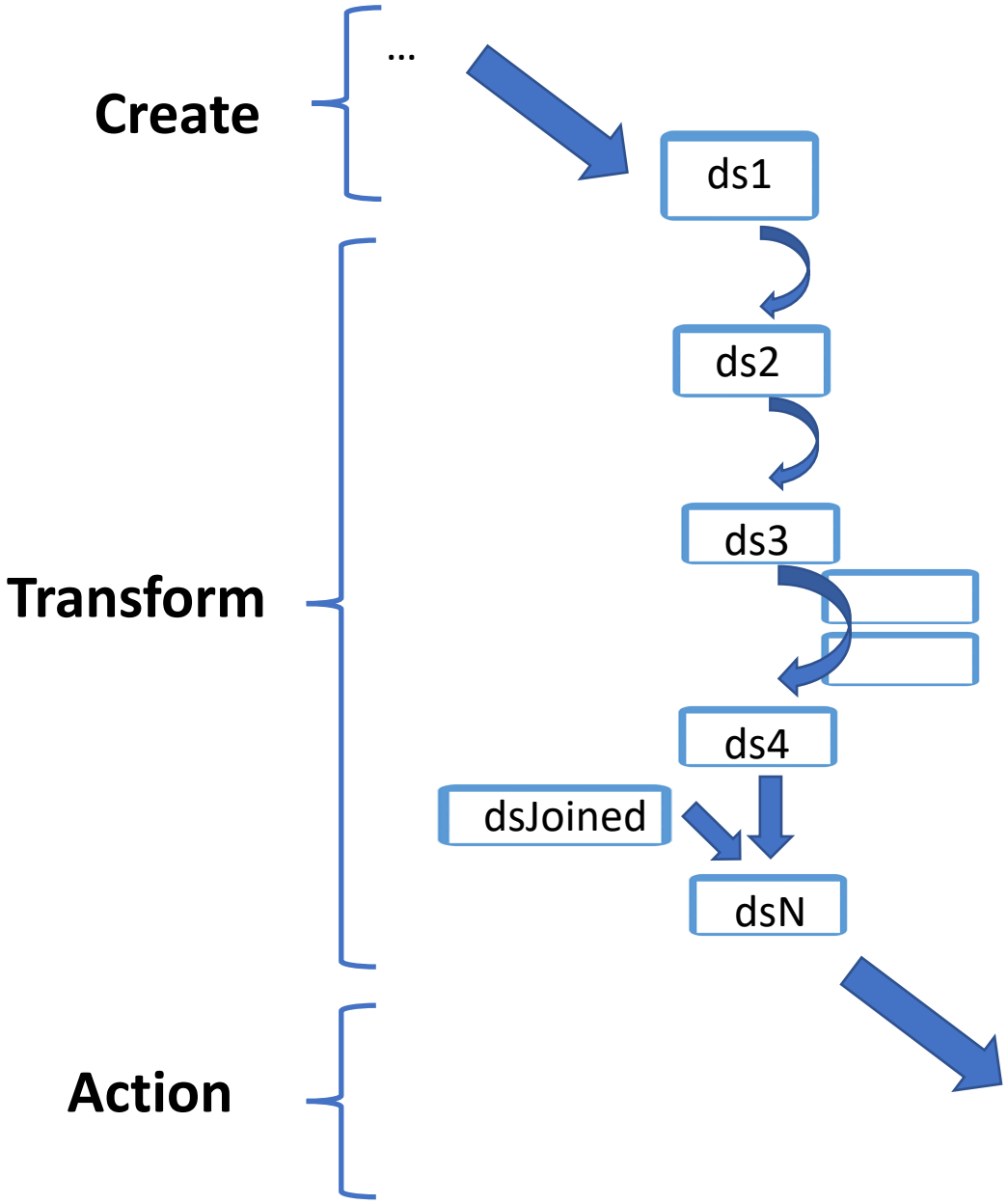
Transform

Dataset newDataset = dataset . <createNewWithTransform> (…)
Example: chain method
dataset = ds .filter(..) .filter(..) .map(..) .groupBy(..),

Action

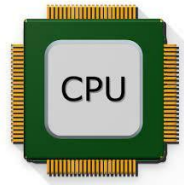
<Result> = dataset. <action> ();
example: long result = dataset. count ();

Dataset operations...




```
Dataset<T1>   ds1 =  spark.read ….

Dataset<T2>    ds2 =  ds1.map(   x -> f(x)  );

Dataset<Row>  ds3 =  ds2.toDF();

Dataset<Row>  ds4 =  ds3 .filter(  « col >= value »)
                         .filter( x -> g(x) )
                         .map( y -> h(y) );

…
Dataset<Row>  dsN =   ds4  .   join( dsJoined)
```

ds . **show**(); // => TRIGGER COMPUTE !!



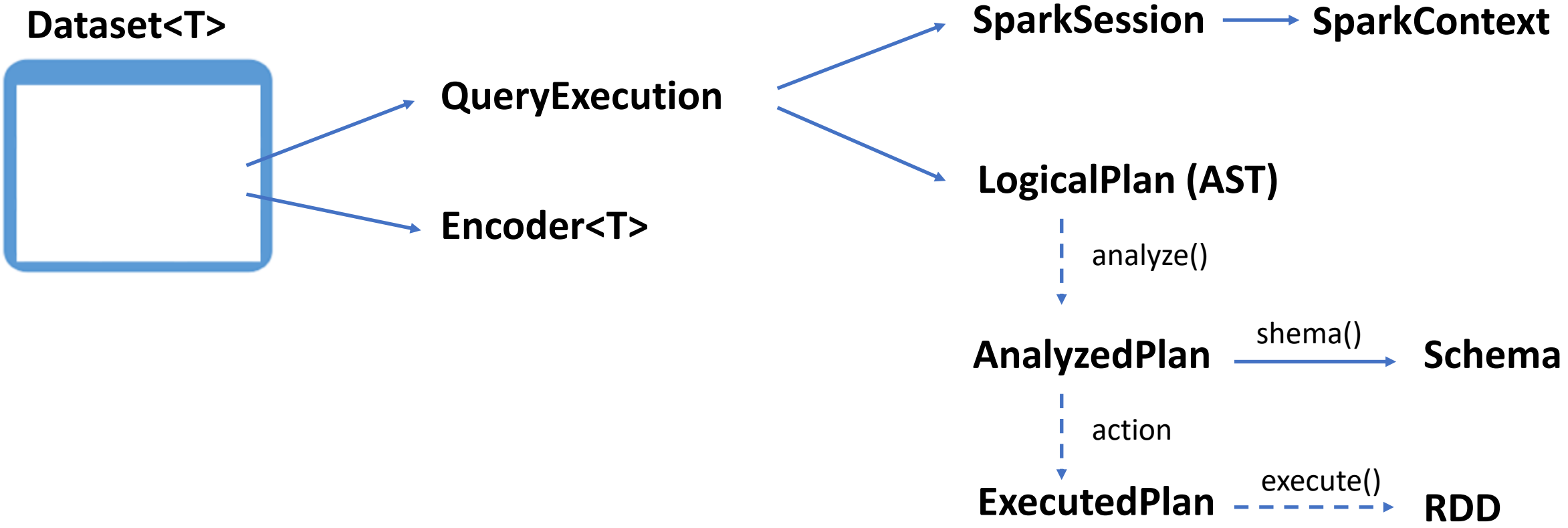
Dataset /** javadoc.. */ (3/4)

Internally, a Dataset represents a logical plan
that **describes** the computation required **to produce the data**.

When an action is invoked,
Spark's query optimizer optimizes the **logical plan**
and generates a **physical plan**
for efficient execution in a parallel and distributed manner.

To explore.. Use ‘explain plan’

Internally…


Dataset<T>
QueryExecution
Encoder<T>
SparkSession
SparkContext
LogicalPlan (AST)
analyze()
AnalyzedPlan
shema()
Schema
action
ExecutedPlan
execute()
RDD

Dataset /** javadoc.. */ (4/4)

To efficiently support domain-specific objects, an ‘Encoder’ is required.

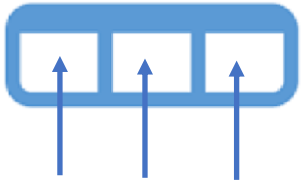
The encoder maps the domain specific type `T` to Spark's internal type system.
(…) to generate code at runtime
to serialize object into a much efficient binary structure
(..) with lower memory footprint
(..) optimized for processing (e.g. in a columnar format).

To explore representation of data, use `schema` function.

Encoder<T>

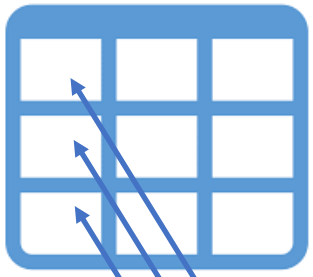
Encoder = { **objectSerializer**, **objectDeserializer**: Expression, **class**}

Row storage
Dataset<Row>

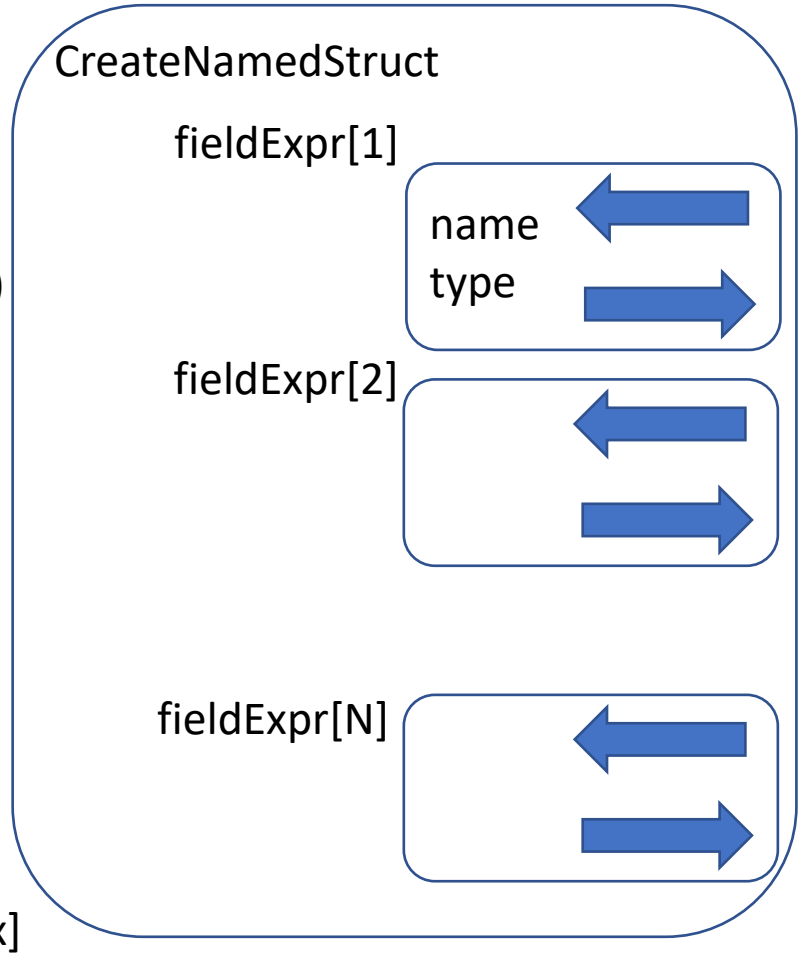


row.getAt(colIndex)
.setAt(colIndex, value)

Columnar storage



colValues[rowIndex]



Convert
To Spark
« Type System »

Getter Methods

Object

Setter Methods

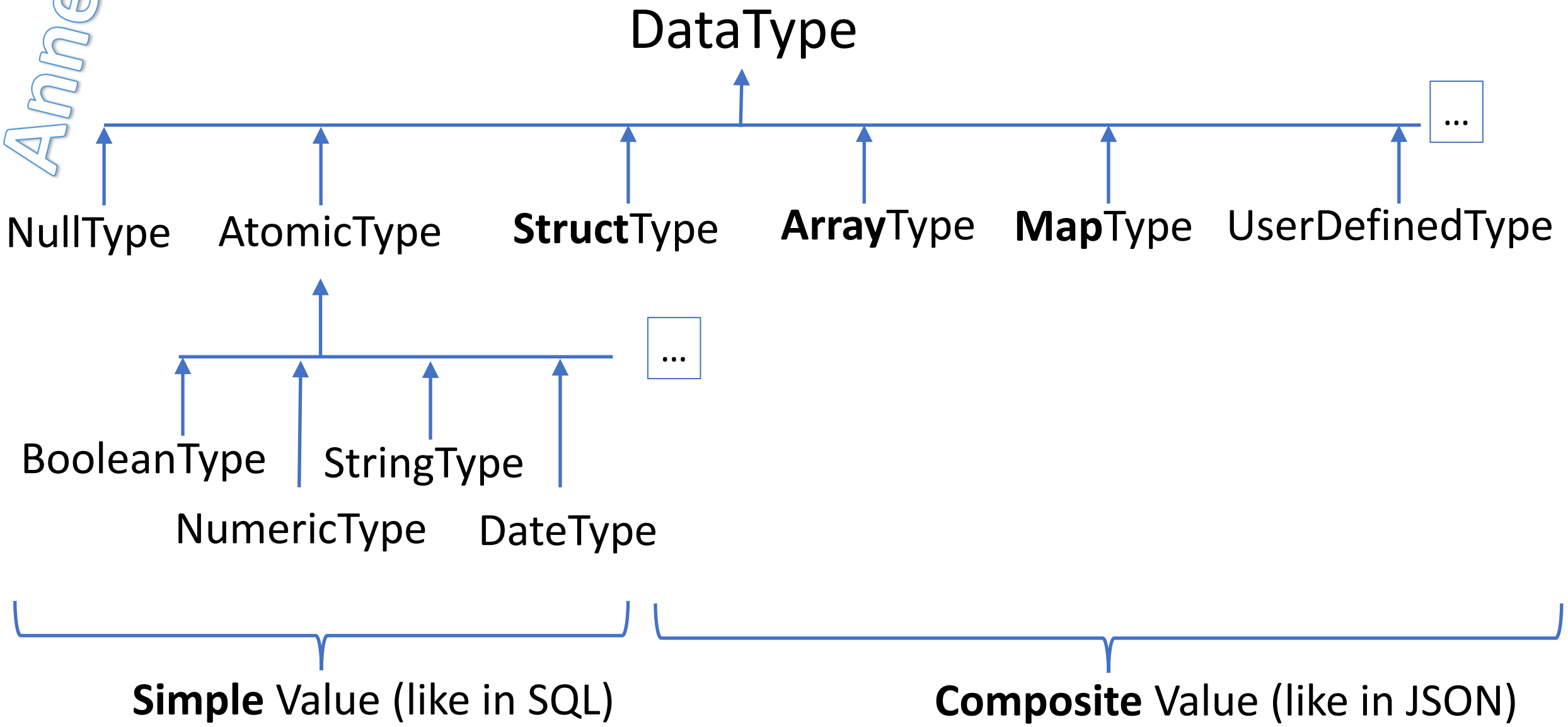
field1, field2, … fieldN

Annexe

Spark « Type System »
DataTypes …

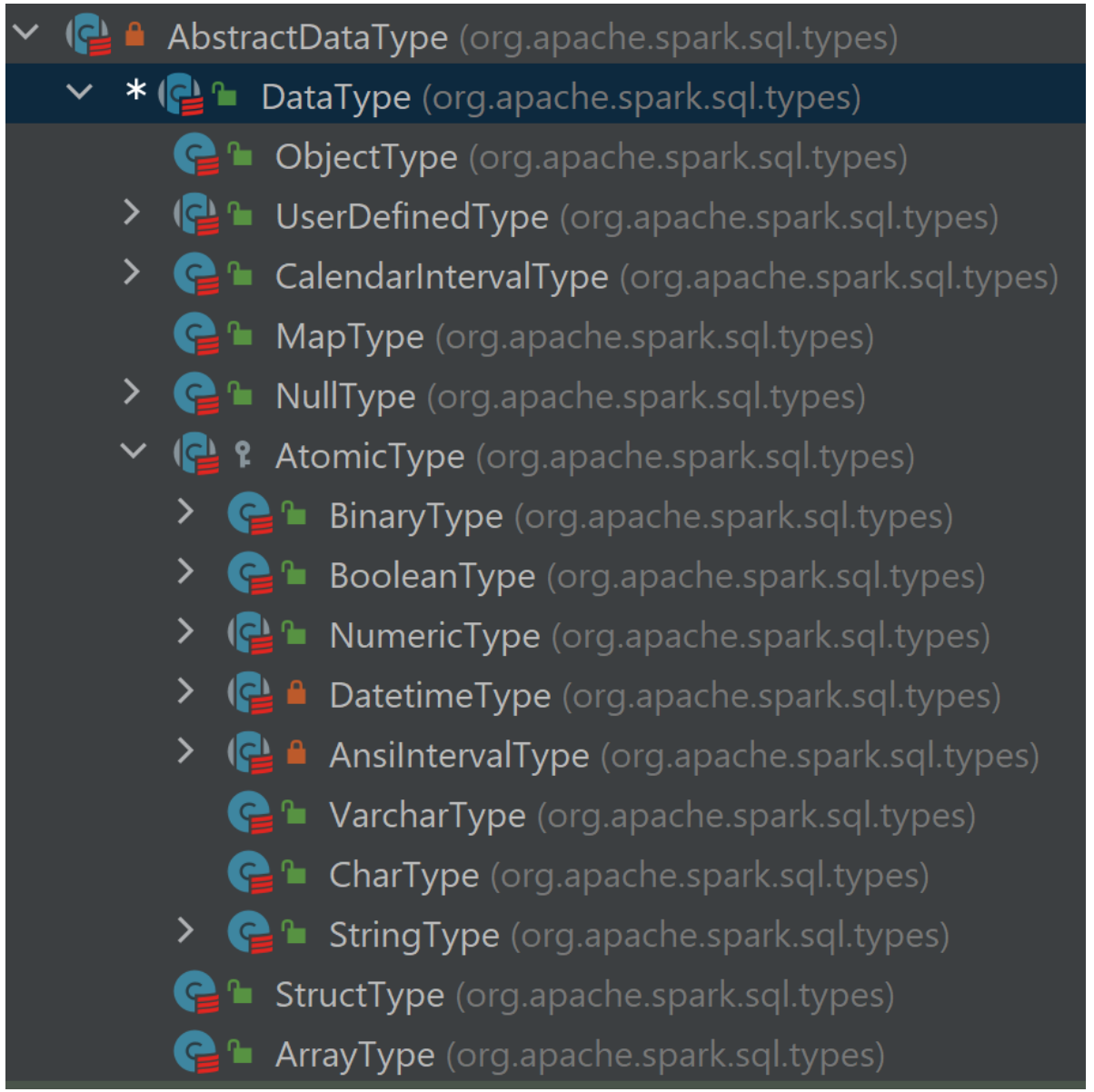
Annexe

Internal Spark « Type System »



Annexe

Spark DataType



Hive - Spark SQL supports Struct, List, Map …

Example:

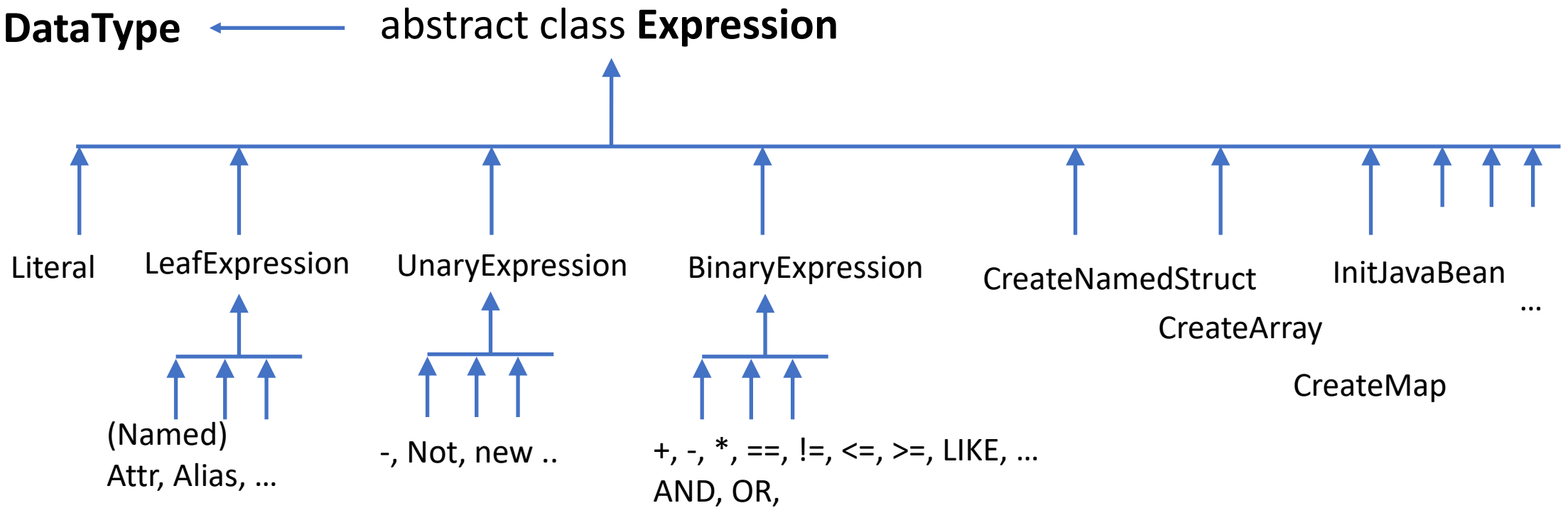
```
CREATE  EXTERNAL  TABLE  `student` (
   firstName string,  lastName string,

   practicedSports array< named_struct< name: string, numberYear: int > >,
   diploma map<string, named_struct<mention: string, obtentionDate: Date > >
)
```

class LogicalPlan ??
class PhysicalPlan ??
class Expression ??

Annexe

Encoder .. Internal Expression with DataType

Expression abstract class AST (Abstract Syntaxic Tree)
for Sql / CodeGenerator / Java Getter-Setter



Annexe

Expression ...

Complex … INTERNAL …
(not need to understand, to use Spark)

Looks like a « **SQL Compiler** »

… support **NOT ONLY Sql literal values**,
but also **NamedStruct, Array, Map, Java Objects** !!!

Used internally to **compile + generate code**

Dataset<T> OK... so what is a "DataFrame" ?

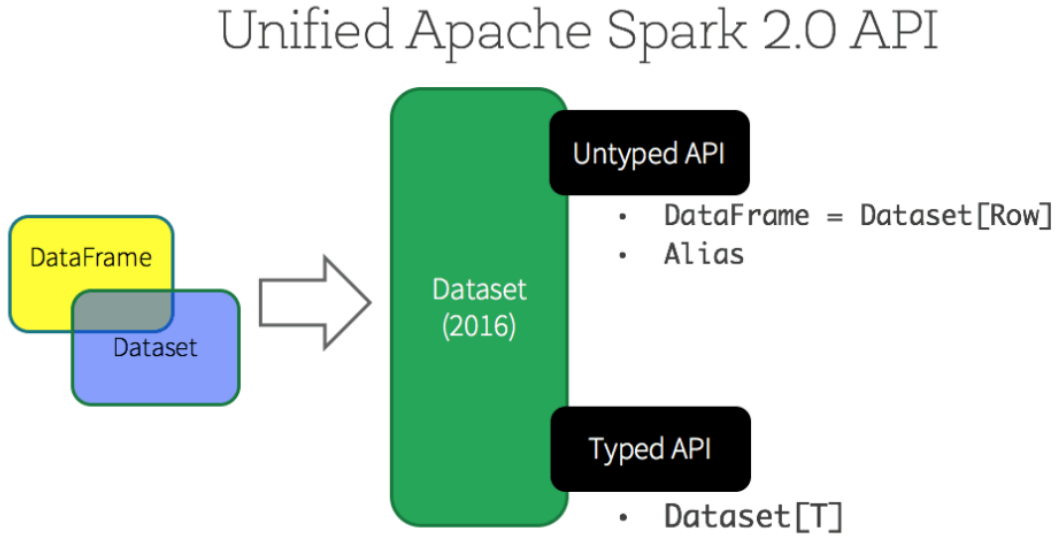
DataFrame = Dataset<Row>

Deprecated DataFrame api … now DataSet<Row>

DataFrame is deprecated since 2016

Spark still have 2 usage « kind »

- untyped (sql) … DataSet<Row> = like Jdbc ResultSet
- Typed … like ArrayList<YourClass>



interface org.apache.spark.sql.Row

Ctor + schema +
SCALA « deconstructor »

- Row
  - S empty() : Row
  - S merge(Seq<Row>) : Row
  - S fromTuple(Product) : Row
  - S fromSeq(Seq<Object>) : Row
  - S unapplySeq(Row) : Some<Seq<Object>>
  - D size() : int
  - A length() : int
  - D schema() : StructType
  - D apply(int) : Object

String + json

- D toString() : String
- A copy() : Row
- D anyNull() : boolean
- D equals(Object) : boolean
- D hashCode() : int
- D toSeq() : Seq<Object>
- D mkString() : String
- D mkString(String) : String
- D mkString(String, String, String) : String
- D getAnyValAs(int) <T> : T
- D json() : String
- D prettyJson() : String
- D jsonValue() : JValue

```
/**
 * This method can be used to extract fields from a [[Row]] object in a pattern match.
 * {{{
 * import org.apache.spark.sql._
 *
 * val pairs = sql("SELECT key, value FROM src").rdd.map {
 *   case Row(key: Int, value: String) =>
 *     key -> value
 * }
 * }}}
 */
def unapplySeq(row: Row): Some[Seq[Any]] = Some(row.toSeq)
```

Typed field getters

- A get(int) : Object
- D isNullAt(int) : boolean
- D getBoolean(int) : boolean
- D getByte(int) : byte
- D getShort(int) : short
- D getInt(int) : int
- D getLong(int) : long
- D getFloat(int) : float
- D getDouble(int) : double
- D getString(int) : String
- D getDecimal(int) : BigDecimal
- D getDate(int) : Date
- D getLocalDate(int) : LocalDate
- D getTimestamp(int) : Timestamp
- D getInstant(int) : Instant
- D getSeq(int) <T> : Seq<T>
- D getList(int) <T> : List<T>
- D getMap(int) <K, V> : Map<K, V>
- D getJavaMap(int) <K, V> : Map<K, V>
- D getStruct(int) : Row
- D getAs(int) <T> : T
- D getAs(String) <T> : T
- D fieldIndex(String) : int
- D getValuesMap(Seq<String>) <T> : Map<String

Class « Row » methods API
… you should not care ?

With SQL built-in functions
… you normally do NOT need any « map(row => { … row.someMethod(..) } »

Map code fragments are advanced usages, running on « Executors »
… You focus on « Driver » code

List<Row> ls = dataset.collect();
… probably a bad idea for in-memory fitting data on driver side

SQL -vs- Compiled Custom Class

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

SQL … « where » vs filter(lambda predicate)

```
spark.sql("""
   select p.id
   from User
   where p.firstName = ‘arnaud’
""")
```

```
Dataset<User> ds = … load(.. Encoder.bean(User.class) );
ds
   .filter( x -> x.getFirstName().equals("arnaud") )
   .map( x -> x.getId() );
```

Concise vs Maintanable …

Java : verbose langage with parenthesis …
+ additionnal JavaBean getter/setter naming conventions verbosity …
Spark use « Encoder.beans() » to enforce this

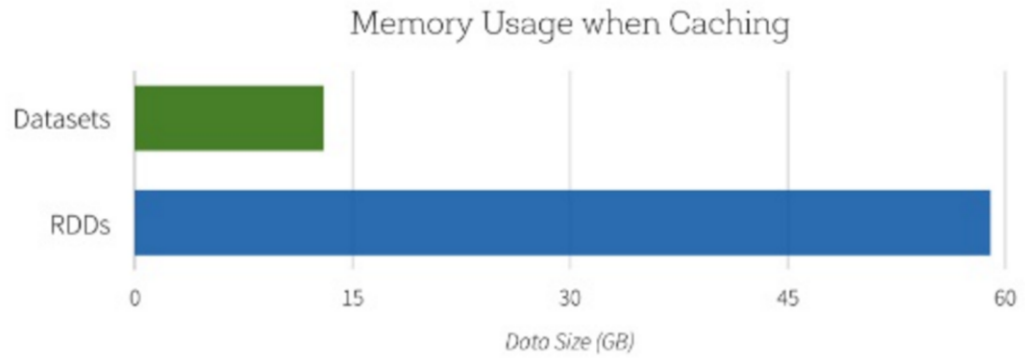
But works well + safe at compile time + easily readable & maintanable

Did you ever try to refactor > 20 000 lines of legacy SQL ?

Did you ever read > 20 000 lines of complex/idiomatic Scala code ?

Spark Tungsten …
Internal memory encoder for « Row »

Space Efficiency



Second, since **Spark as a compiler** understands your Dataset type JVM object, it maps your type-specific JVM object to Tungsten's internal memory representation using **Encoders**. As a result, **Tungsten** Encoders can efficiently serialize/deserialize JVM objects as well as generate compact bytecode that can execute at superior speeds.

OffHeap vs Managed JVM Memory
⇒ May handle Huge DATA
with « small » JVM GC memory

Efficient network
(shuffling) serialization

Efficient SpillToDisk
(swapping)