

BigData – Spark – Processing

discover Spark Core & Sql

Arnaud Nauwynck – Nov 2023

Outline

Example RAW to LAKE transformations

Explanation step-by-step

Dataset

Parallel Distribution

Reminder: Spark RAW to LAKE samples

Typical RAW to LAKE as Spark Java code

read

spark.read

.format(« csv »)

.option(« schema », « col1 type1, ... colN typeN »)

.load(« hdfs://raw/team/domain/table/date=2022-10-12 »)

transform

.as(Encoder.bean(Bean.Class))

.map(bean -> transformBean(bean))

.toDF()

write

.repartition(2, « col1 »)

.sortWithinPartition(« col1, col2, col3 »)

.write

.format(« parquet »)

.save(« hdfs://lake/team/domain/table/date=2022-10-22 »);

Typical RAW to LAKE processing with Spark as SQL code

| | | |
|-----------|---|---|
| write | { | INSERT OVERWRITE lake_team_domain.table |
| | | SELECT /* +REPARTITION(col1, 2) */ col1, col2, udf_func1(col3, col4) as col3, udf_func2(col4, col5) as col4, .. |
| transform | { | |
| read | { | FROM raw_team_domain.table |
| transform | { | JOIN lake_anotherTeam_domain.anotherTable x ON x.ID=id |
| read | { | WHERE date='2022-10-22' AND .. |
| write | { | SORT BY col1, col2, col3 -- idem sortWithinPartition |

Example of LAKE Aggregation

```
INSERT OVERWRITE
  lake_team_domain.table
SELECT * FROM (
  SELECT * FROM table1 WHERE ..
  UNION
  SELECT * FROM table2 WHERE ..
  UNION
  SELECT * FROM table3 WHERE ..
  UNION
  SELECT * FROM table4 WHERE ..
)
SORT BY col1, col2, col3    -- idem sortWithinPartition
```

Example of « latest value » cristalisation analytical query « over(partition by) »

```
INSERT OVERWRITE
  lake_team_domain.table
SELECT
  col1,col2,.... colN  -- idem * EXCEPT rank  (cf issue SPARK-33164)
FROM (
  SELECT *,
    RANK() OVER (PARTITION BY id ORDER BY update_time DESC) as rank
  FROM lake_team_domain.event_table
)
WHERE rank=1
SORT BY col1, col2, col3  -- idem sortWithinPartition
```

Step-by-Step explained

Typical RAW to LAKE as Spark Java code

spark.read

read
Step 1/4

```
.format(« csv »)  
.option(«schema », « col1 type1, ... colN typeN »)  
.load(« hdfs://raw/team/domain/table/date=2022-10-12 »)
```

transform
Step 2/4

```
.as(Encoder.bean(Bean.Class))  
.map(bean -> transformBean(bean) )  
.toDF()
```

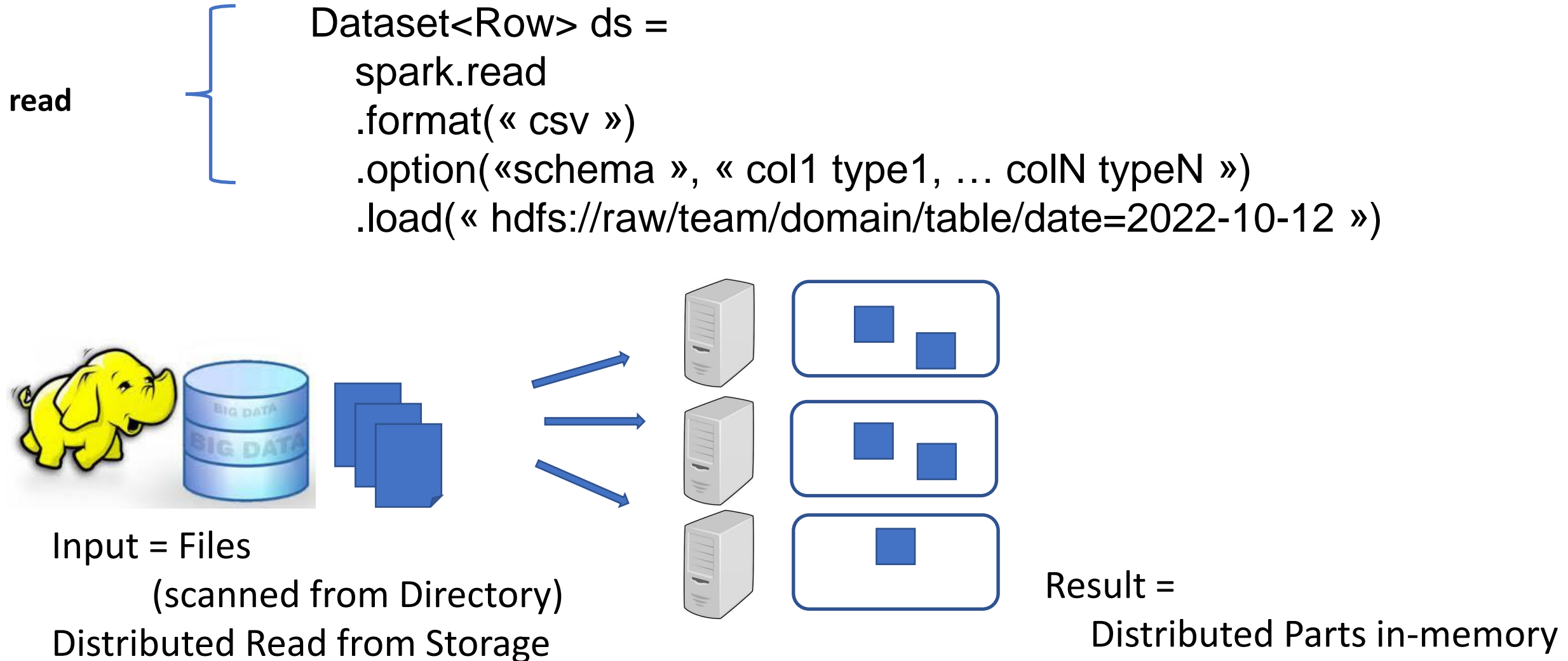
Step 3/4

```
.repartition(3, « col1 »)  
.sortWithinPartition(« col1, col2, col3 »)
```

write
Step 4/4

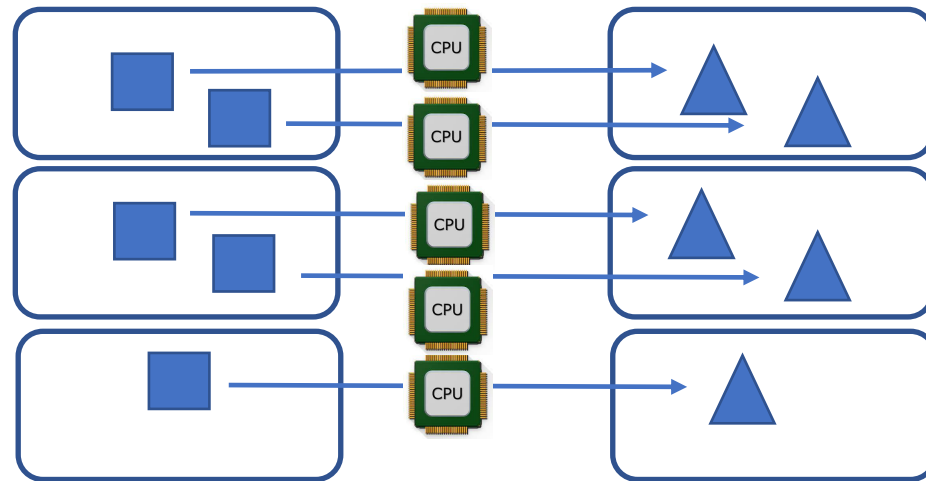
```
.write  
.format(« parquet »)  
.save(« hdfs://lake/team/domain/table/date=2022-10-22 »);
```

RAW to LAKE – Step 1/4: read to Dataset



RAW to LAKE – Step 2/4 : Transform Dataset

transform { Dataset<Row> ds2 = ds.map(row -> transformData(row))

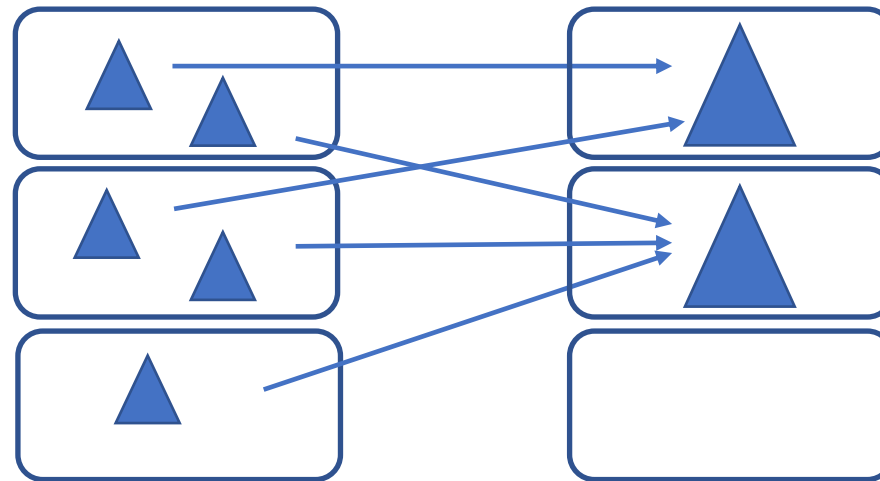


Distributed Processing to compute each new part

RAW to LAKE – Step 3/4 : Repartition Dataset

transform

```
Dataset<Row> ds3 = ds2  
  .repartition(2, « col1 »)  
  .sortWithinPartition(« col1, col2, col3 »)
```



Network Shuffle to distribute / group / sort data

RAW to LAKE – Step 4/4 : Write Dataset

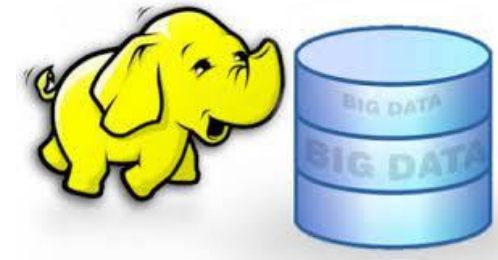
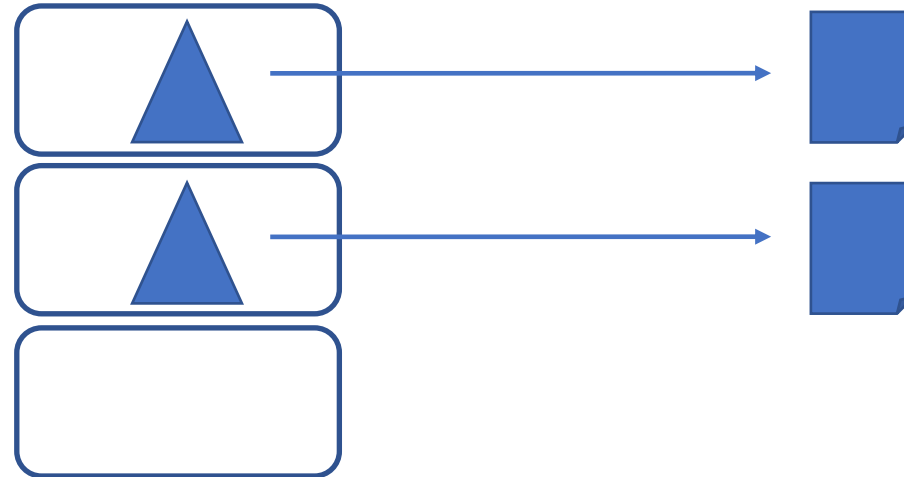
write



```
ds3.write
```

```
.format(« parquet »)
```

```
.save(« hdfs://lake/trig/domain/table »)
```



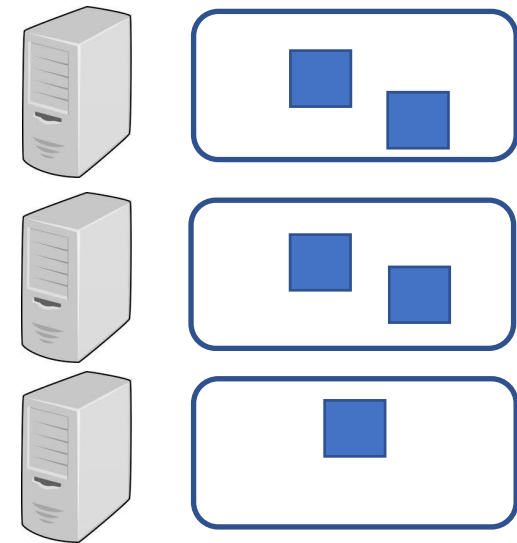
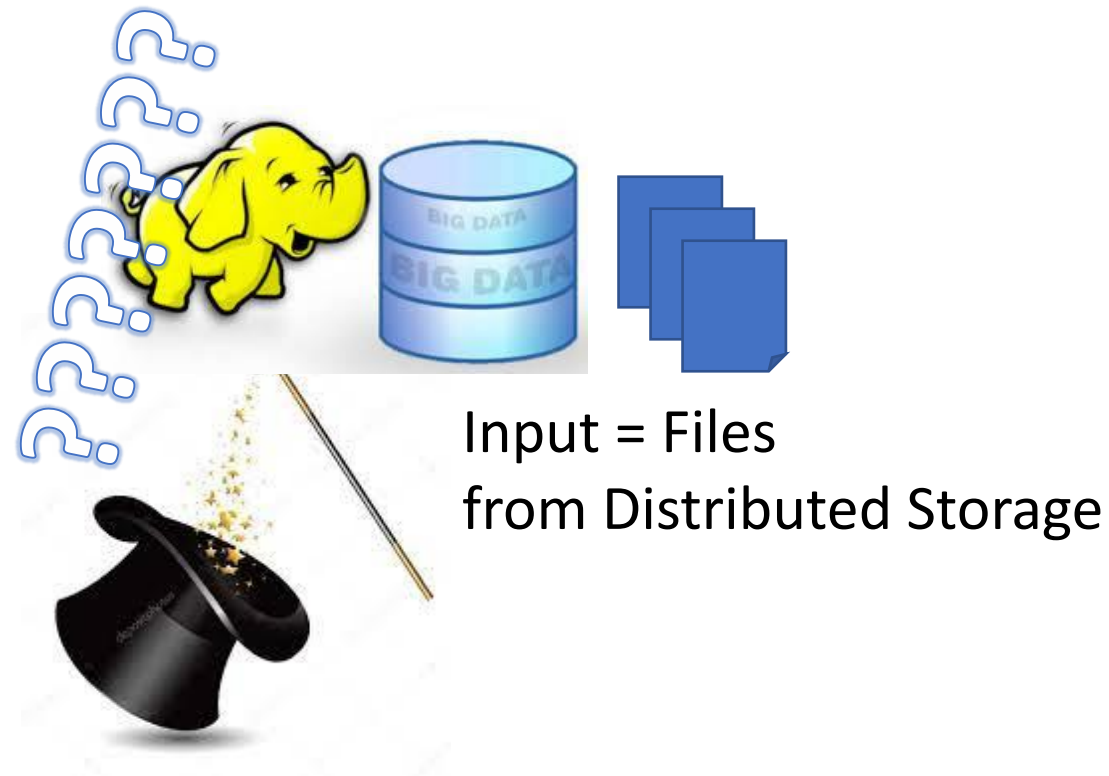
Distributed Write Dataset to Storage

How it works ?



Zooming more ..

RAW to LAKE – Step 1/4: read to Dataset



Result = Distributed Parts in-memory



How to Assign $N \times \text{Files} - P \times \text{blocks} \Leftrightarrow$ to $Q \times \text{Executors}$??
+ Retry on Error ?? + Communicate more ??

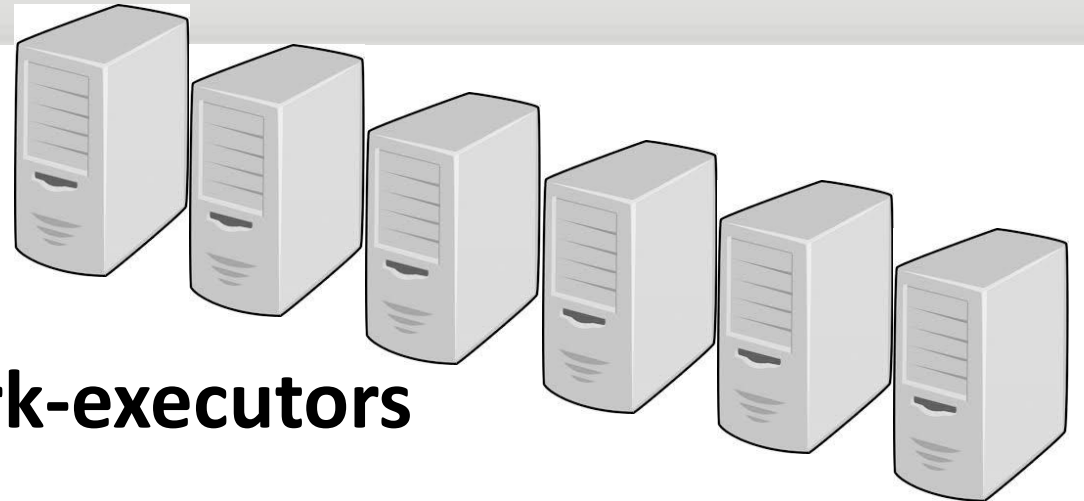
Analogy : How to play music ?
(N musicians without 1 Conductor \neq 1 Orchestra)



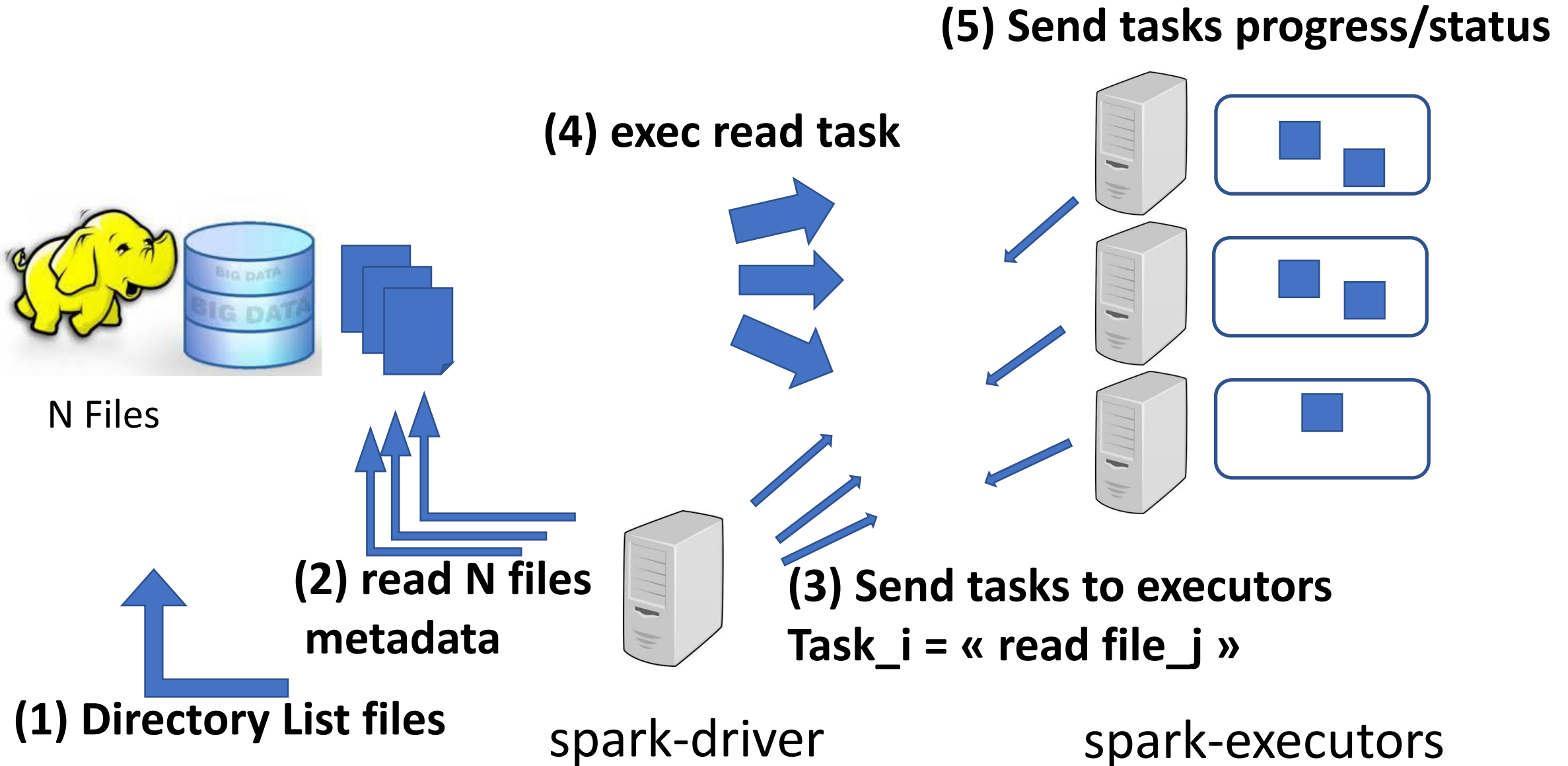
spark-driver



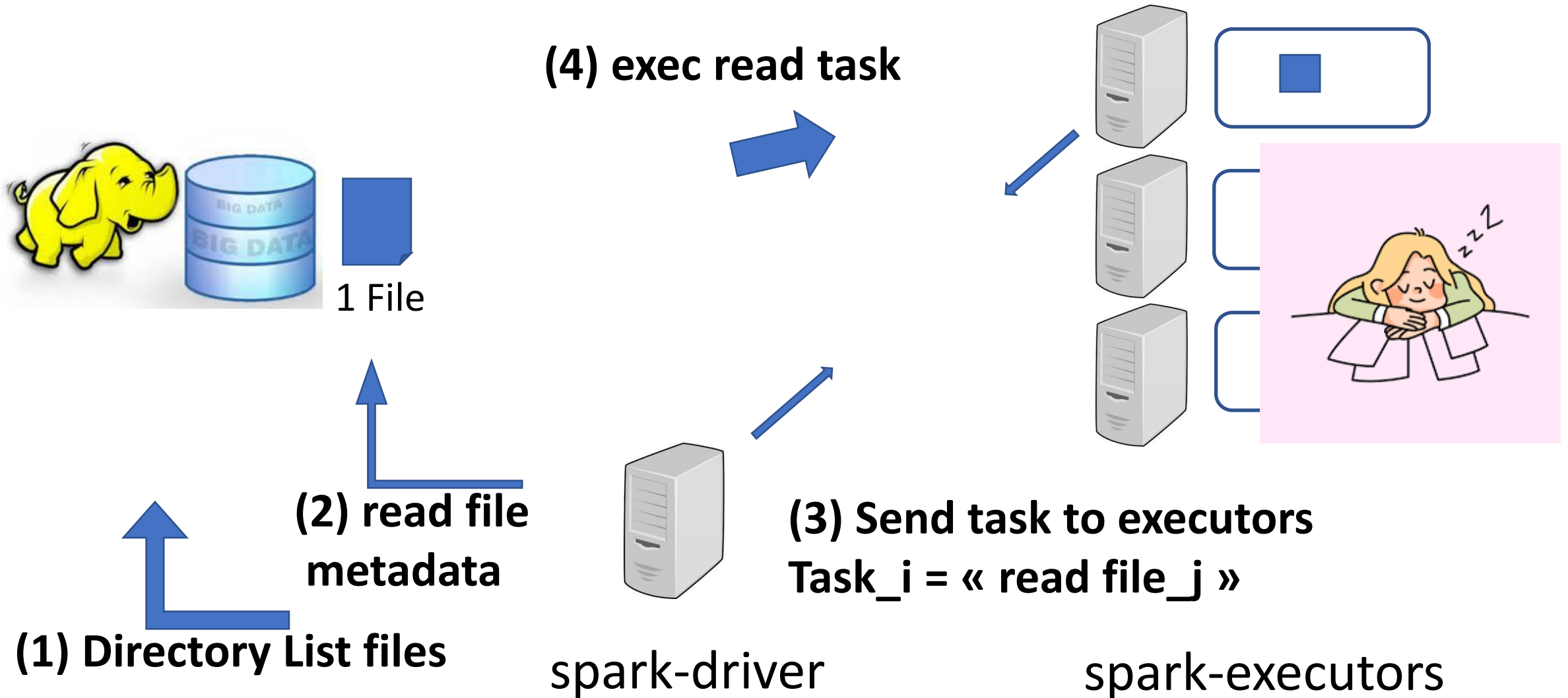
spark-executors



Read N Files – assign Tasks to Executors

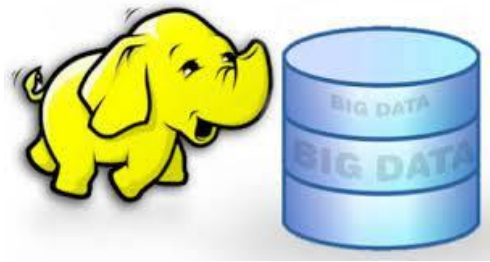


Remark [1/2] on Parallelism only 1 File -> only 1 Task

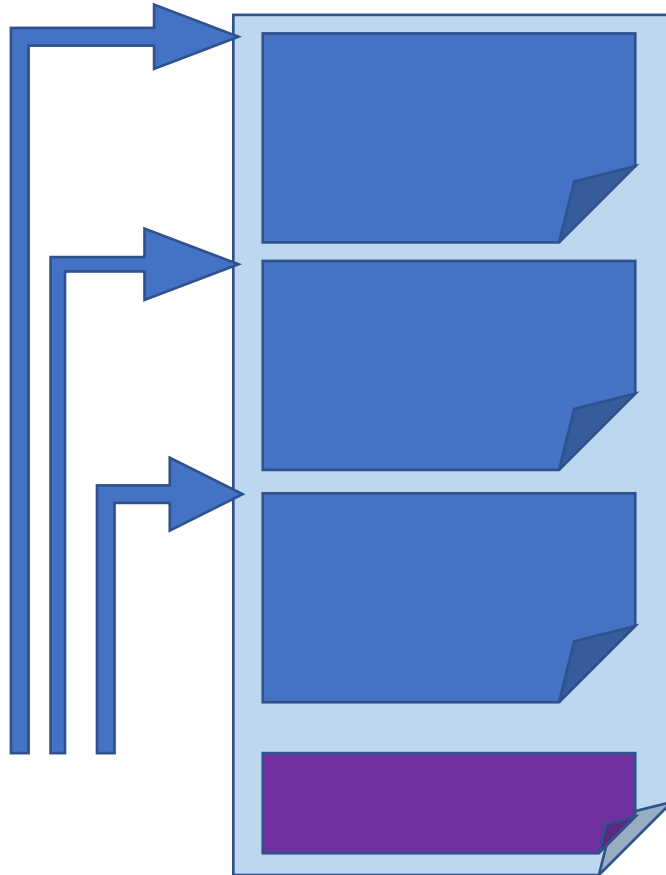


Remark [2/2] on Parallelism

Splittable File format (parquet).. Like dir



offsets
...Like dir



1 splittable file

N x Blocks
(usually 256 Mo)
independent, read at offset

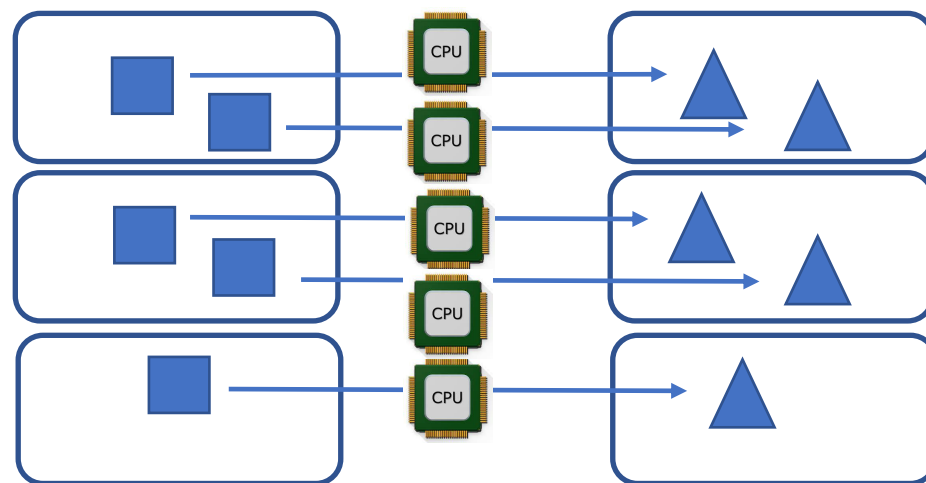
File metadata
= schema + N blocks infos
(offset + stats)

Zooming RAW to LAKE – Step 2/4 : Transform Dataset

transform



```
Dataset<Row> ds2 = ds.map(row -> transformData(row) )
```



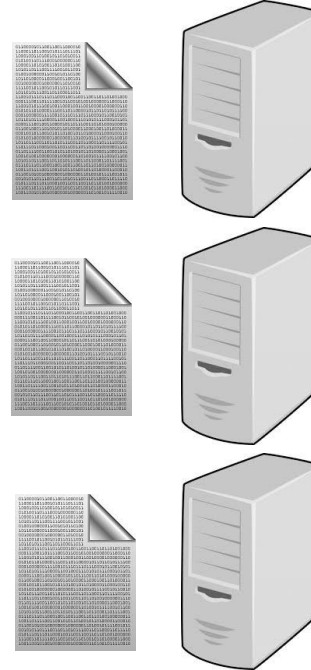
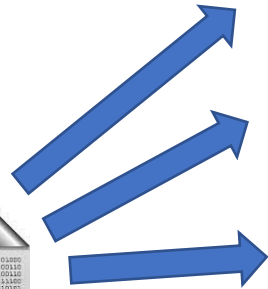
Distributed Processing to compute each new part

WholeStageCodeGen

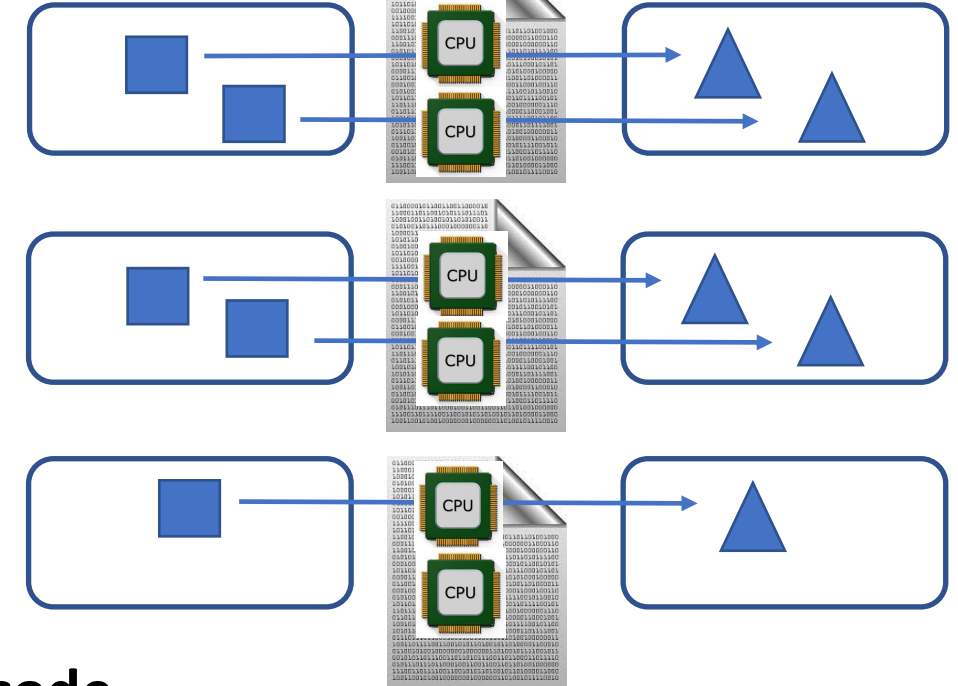
Program

= Dataset instructions

```
36 val sc = new SparkContext(args(0), "GroupBy Test",  
37   System.getenv("SPARK_HOME"), SparkContext.jarOfClass(this.getClass))  
38  
39 val pairs1 = sc.parallelize(0 until numMappers, numMappers).flatMap { p =>  
40   val ranGen = new Random  
41   val arr1 = new Array[(Int, Array[Byte])](numKVPairs)  
42   for (i <- 0 until numKVPairs) {  
43     val byteArray = new Array[Byte](valSize)  
44     ranGen.nextBytes(byteArray)  
45     arr1(i) = (ranGen.nextInt(Int.MaxValue), byteArray)  
46   }  
47   arr1
```



(3) Send task + bytecode
to spark-executors



(1) Generate java code

(RDD Spark sub-class « WholeStageCodeGen\$ »)

(2) Compile Bytecode

(4) Execute tasks

Advanced Transform ...

using Row -> Java -> map()-> Java -> Row

transform {

```
ds.as( Encoders.bean(InputBean.class) )  
.toDF()
```

```
class InputBean {  
  ...  
}
```



```
class OutputBean {  
  ...  
}
```

```
OutputBean transformBean(InputBean b) {  
  // complex transform in java  
  return new OutputBean(... );  
}
```

Explained as().map().toDF()

```
Dataset<Row> ds = ...
```

```
// convert Row->Bean
```

```
Dataset<InputBean> dsInputBean =
```

```
ds.as(Encoder.bean(InputBean.class))
```

```
// map
```

```
Dataset<OutputBean> dsOut =
```

```
dsInputBean.map(bean -> transformBean(bean) )
```

```
// convert OutputBean -> Row
```

```
Dataset<Row> df = dsOut.toDF();
```

```
ds.as( Encoders.bean(InputBean.Class) )  
  .map(bean -> transformBean(bean) )  
  .toDF()
```



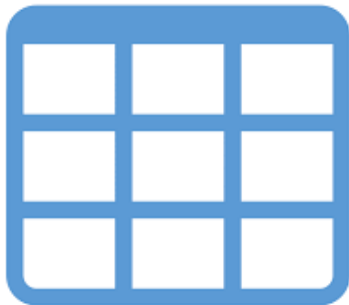
Converting Tabular SQL Row to Java Beans

```
CREATE TABLE MyTable (  
  field1 Int,  
  field2 String  
)
```



```
public class MyBean {  
  public int field1;  
  public String field2;  
}
```

Dataset<Row> df = ...



encoder = Encoders.bean(MyBean.class)

df.as(encoder)



ds.toDF()

Dataset<MyBean> ds = ...

Object[0] →



Object[1] →



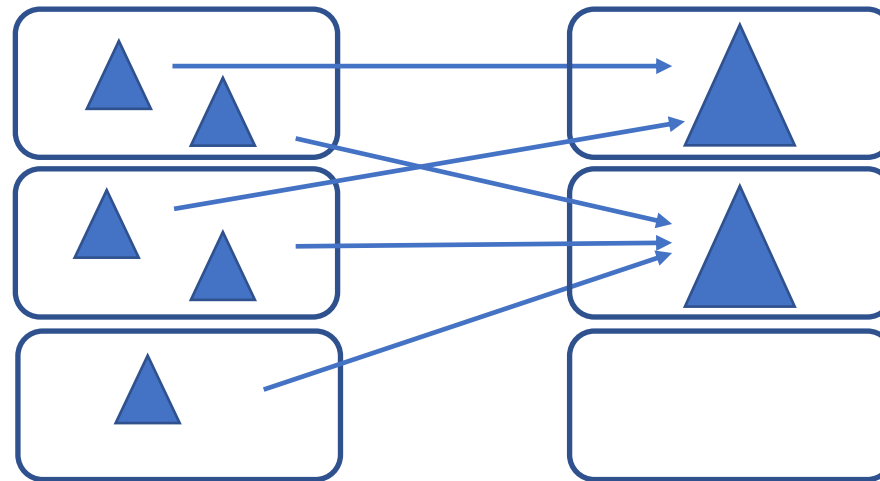
Object[2] →



RAW to LAKE – Step 3/4 : Repartition Dataset

transform

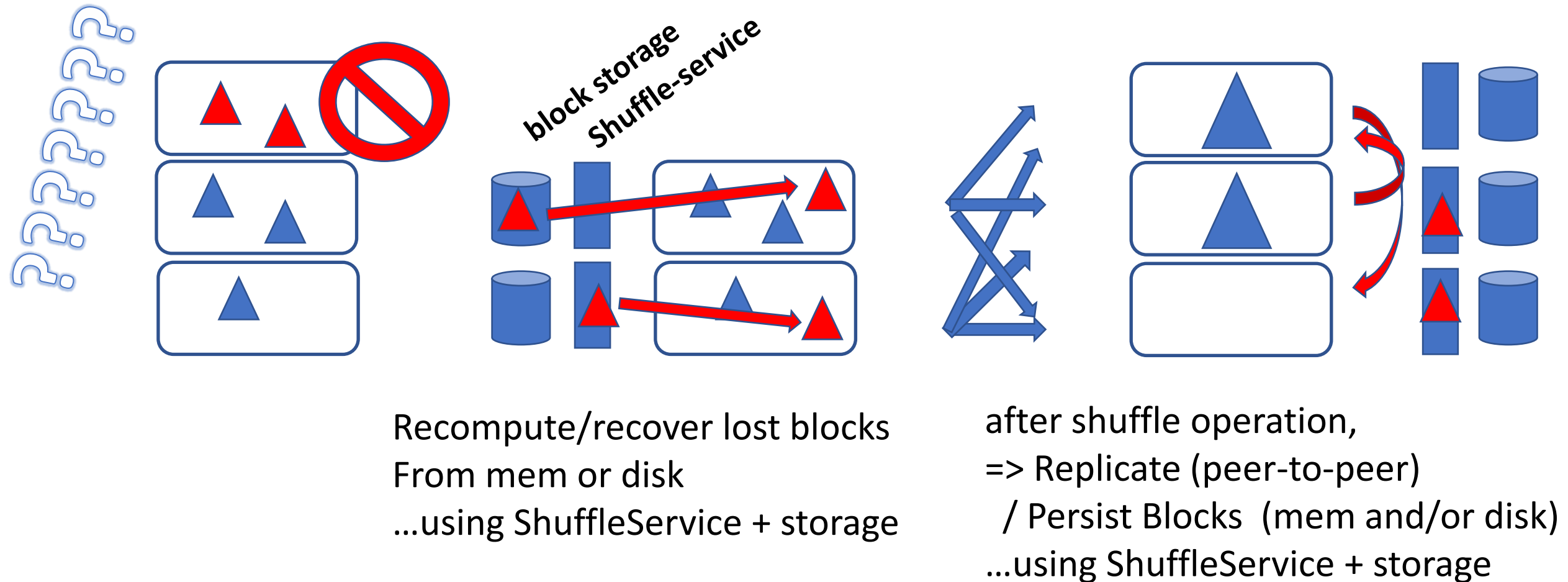
```
Dataset<Row> ds3 = ds2  
  .repartition(2, « col1 »)  
  .sortWithinPartition(« col1, col2, col3 »)
```



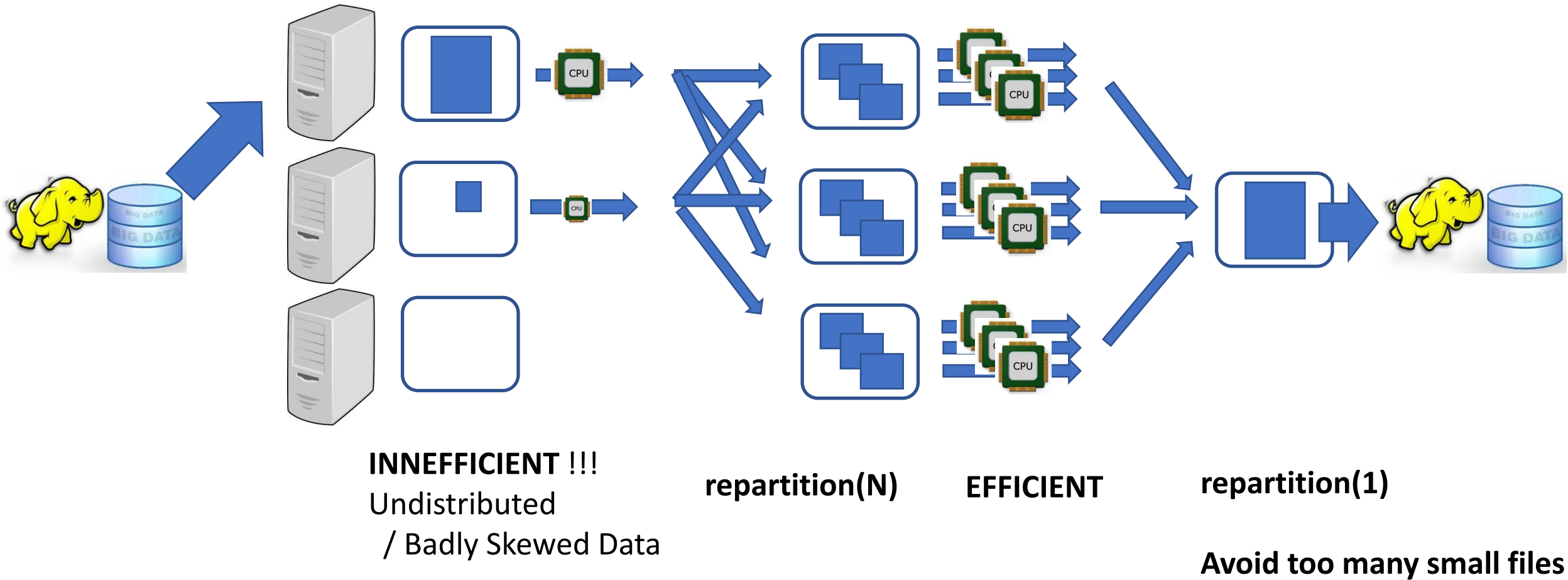
Network Shuffle to distribute / group / sort data



What if a Spark-Executor crash / Node stop ?



Example usage:
`repartition(N).map(..).repartition(1)`

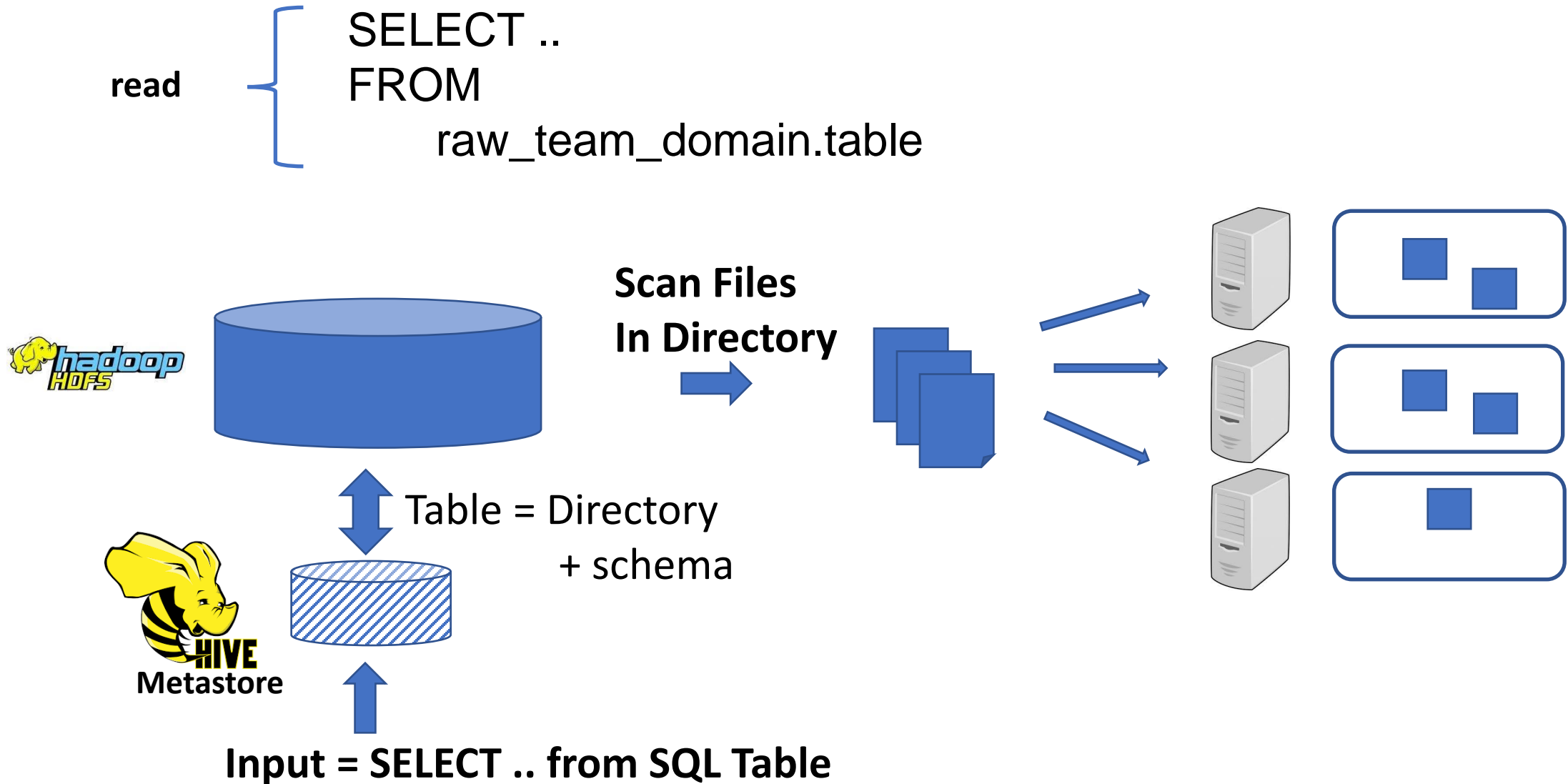


Example transformation ... in SQL

Typical RAW to LAKE as Spark SQL

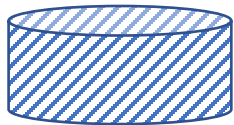
| | | |
|-----------|---|---|
| write | { | INSERT OVERWRITE lake_team_domain.table SELECT /* +REPARTITION(col1, 2) */ col1, col2, udf_func1(col3, col4) as col3, udf_func2(col4, col5) as col4, .. |
| transform | { | |
| read | { | FROM raw_team_domain.table |
| transform | { | JOIN lake_anotherTeam_domain.anotherTable x ON x.ID=id |
| read | { | WHERE date='2022-10-22' AND .. |
| write | { | SORT BY col1, col2, col3 -- idem sortWithinPartition |

Explained ... SQL (-> Files) -> Dataset



(Hive) MetaStore

Store ONLY metadatas (DDL + partitions)



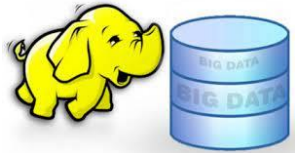
DDL:

```
CREATE EXTERNAL TABLE students (  
  socialSeculd: Int,  
  firstName string, lastName string,  
  birth: Date, ...  
) PARTITIONED BY (promo: Int)  
STORED AS parquet  
LOCATION 'hdfs://lake/students'
```

Mapping SQL – Dirs+Files



Location Dir + Partitions



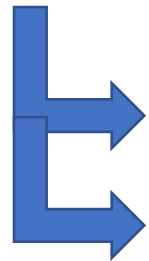
« / » = root hdfs://host/



« /lake » (dir)

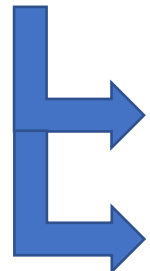


« /students » (table storage dir)



« /promo=2021 » (partition dir)

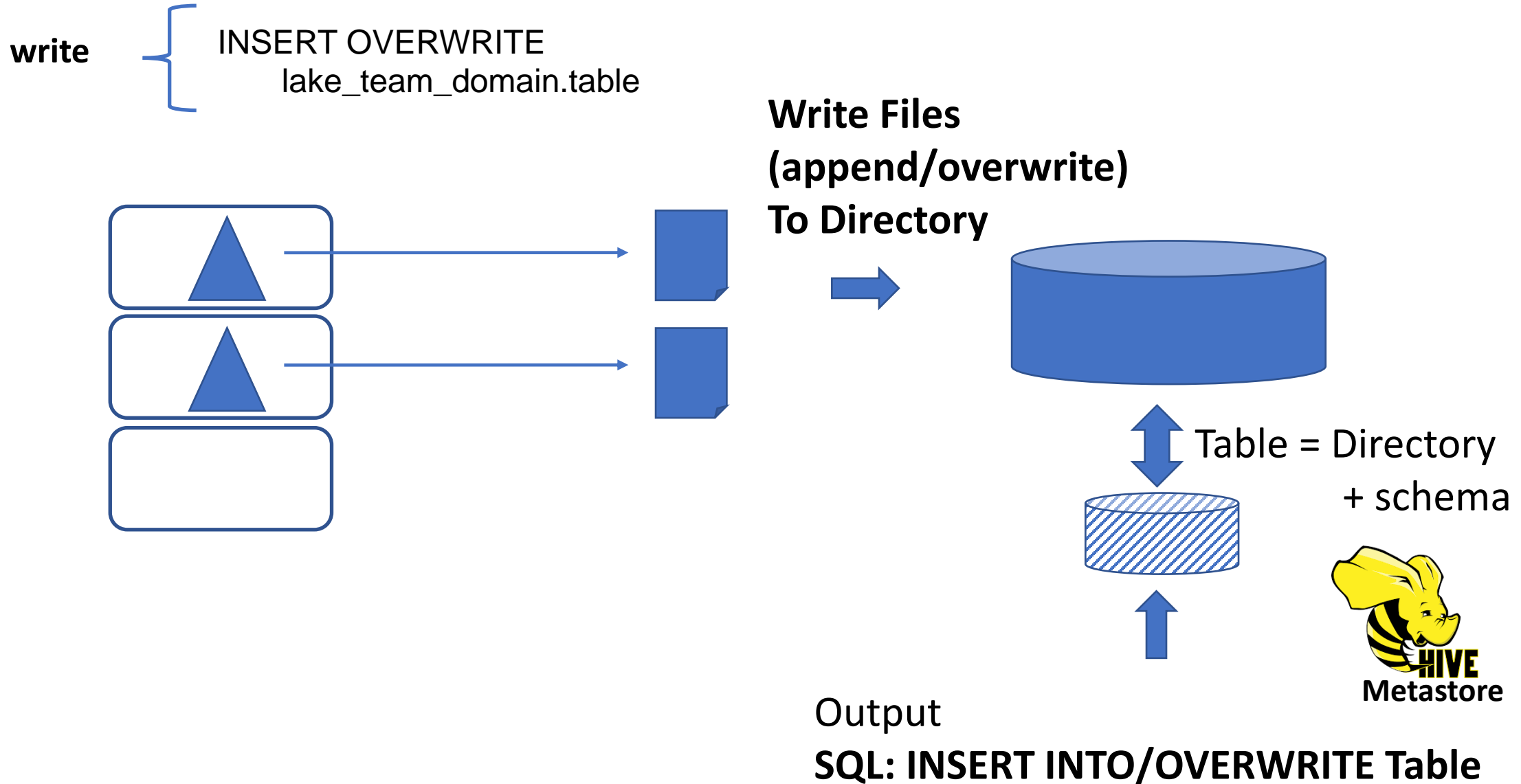
« /promo=2022 » (partition dir)



« file1.parquet » (data file)

« file2.parquet » (data file)

Dataset -> INSERT SQL Table (-> Files)



More Java <-> Sql Interactions

Executing SQL from Java



```
for( int i = 0; i < 10; i++) {  
    String sql = « SELECT * from db.table » + i;  
  
    Dataset[Row] ds = spark.sql(sql);  
  
    ..  
}
```

NO imperative in SQL (cf PL/Sql extensions)
=> OK in java code : if, for(), ...

Java DataSet as SQL View



```
Dataset[Row] ds = ..
```



```
ds.createTemporaryView(« myview1 »)
```



```
spark.sql(« SELECT * FROM myview1 »)
```



Calling Java from SQL : User-Defined Function

transform {

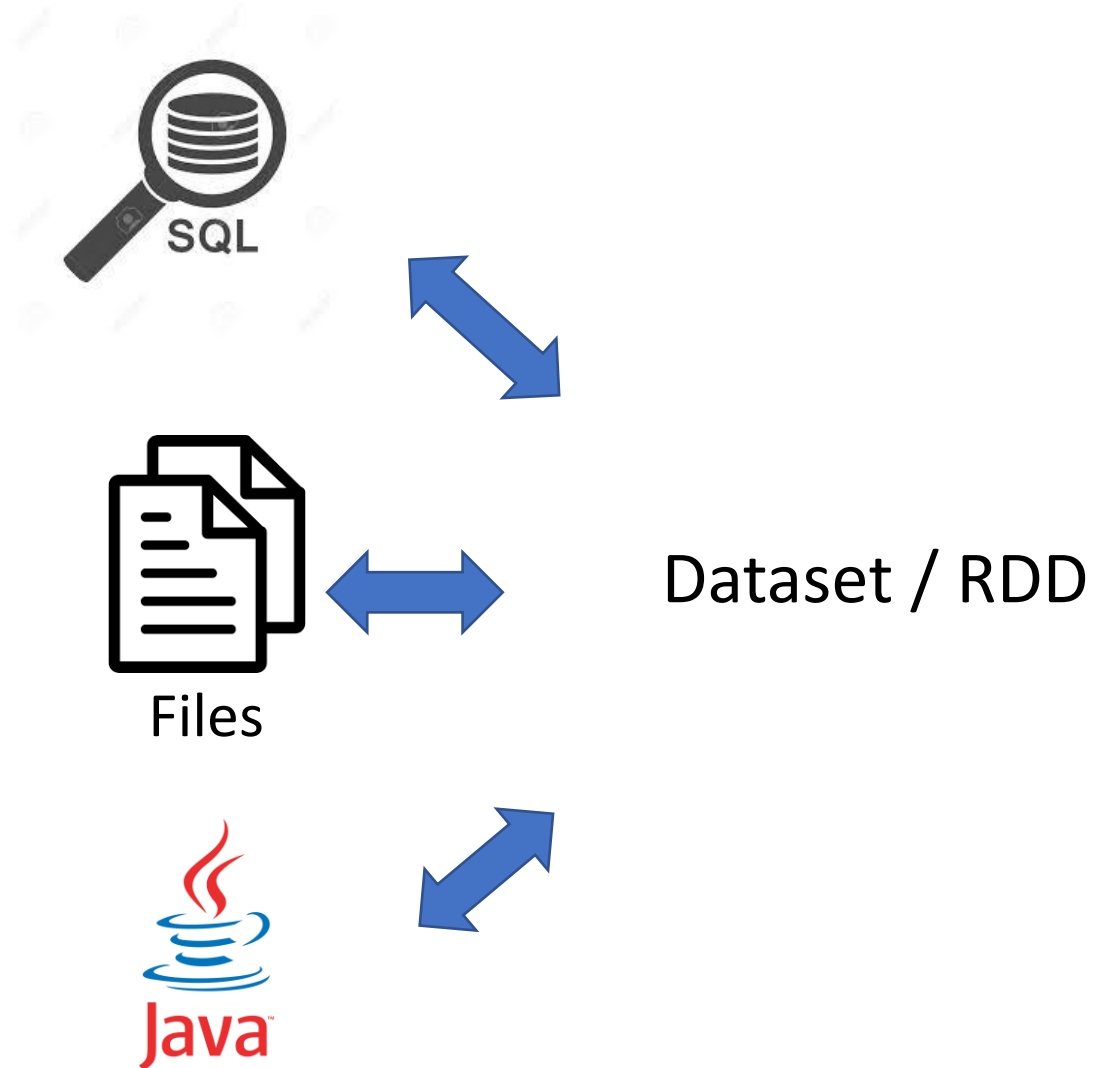


```
SELECT ..  
  udf_func1(col3, col4) as col3,  
  udf_func2(col4, col5) as col4,
```

```
int func1(int x, int y) { return x+y; }
```

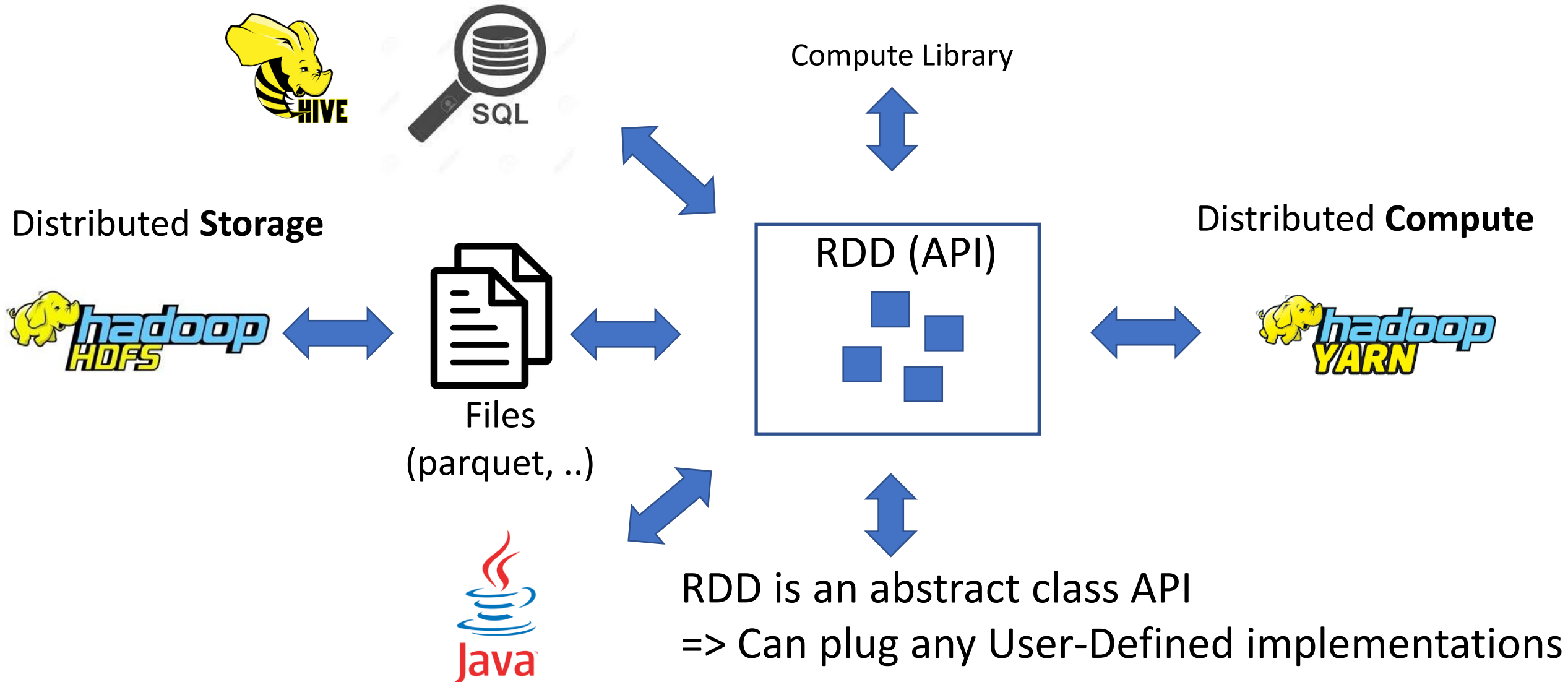
```
spark.udf().register(« udf_func1",  
  (UDF2<Integer,Integer, Integer>) ::func1,  
  DataTypes.IntegerType);
```

Spark = Unified Sql-Files-Java

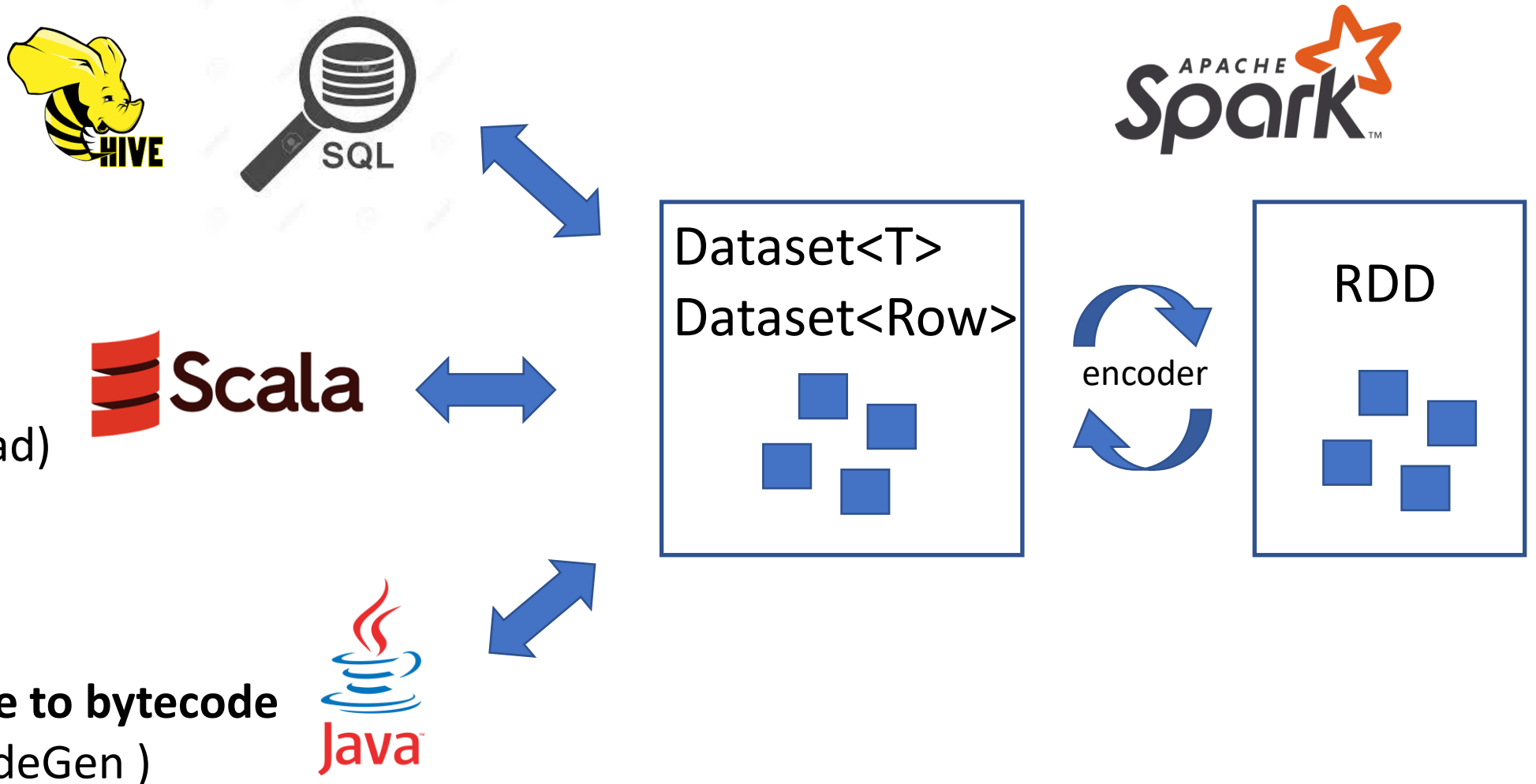


Spark : Unified Engine

(Distributed Storage, Distributed Compute)



Dataset API SQL Extensions



More Extensions: Hadoop FileSystem API



HDFS implements FileSystem



Distributed Storage API

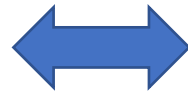


```
abstract class FileSystem {  
    ..read, write, list,  
}
```



Spark rely on API
=> Can plug any implementations

java.io.File
adapter



More adapters

....



More Extensions: Cluster Scheduler API

Cluster Manager
Scheduler API

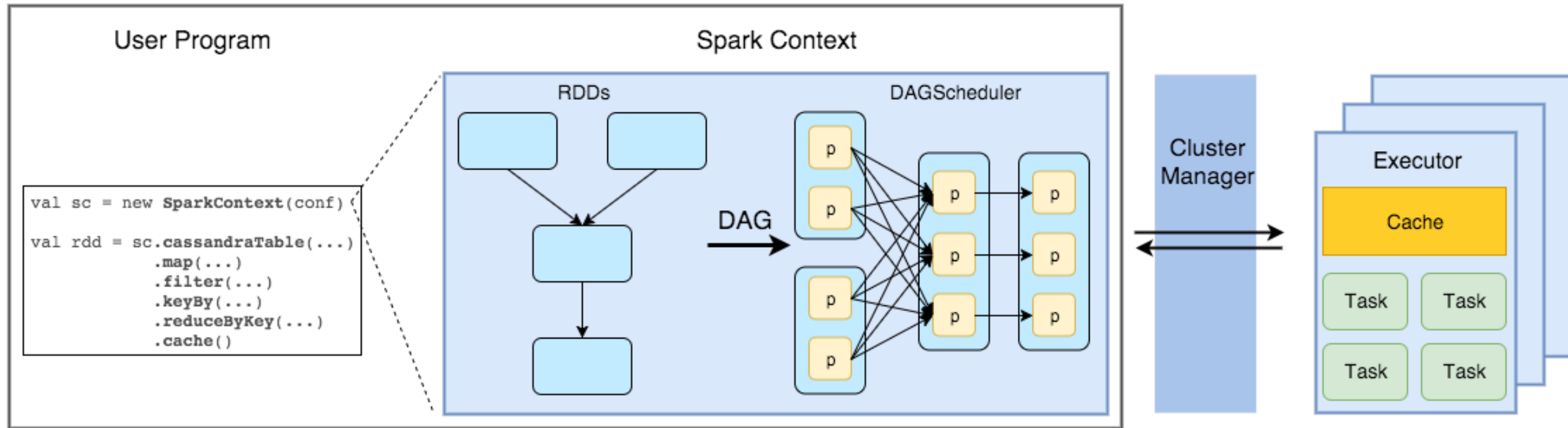


```
TaskScheduler (SPI)  
abstract class TaskScheduler {  
  ..start,stop,  
  submitTasks,cancelTasks,  
  notify Host-Executor-Task changes  
}
```



Spark Application

Workers



More « Extensions »

backport API to other Languages (Python, R)

class DataFrame:

... hundred methods ...

```
def method123 (self, x) -> Y  
    self._jdf.method(x, self.sparkSession)
```



API



API

public class Dataset {

... hundred methods ...

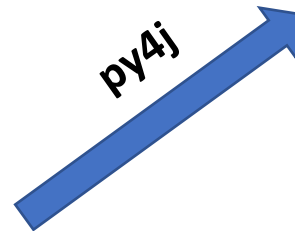
```
public Y method123 (X x) {  
    ... internals sparkContext...  
}
```



```
from py4j.java_gateway import JavaObject  
from subprocess import Popen  
from pyspark.context import SparkContext  
..  
pid = Popen(« spark-submit » ...)  
Socket(.. )
```



py4j

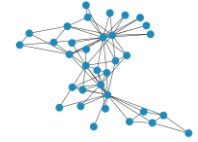
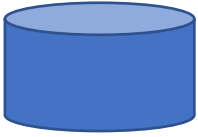


Scala

SparkContext

Spark-Core + ...

Structured
Data



Spark SQL

Spark
Streaming

Spark MLlib

Spark
GraphX



Modules



Amazon S3



Azure Data Lake Storage Gen2

DataSource Connectors
(Hadoop API)



Cluster Manager



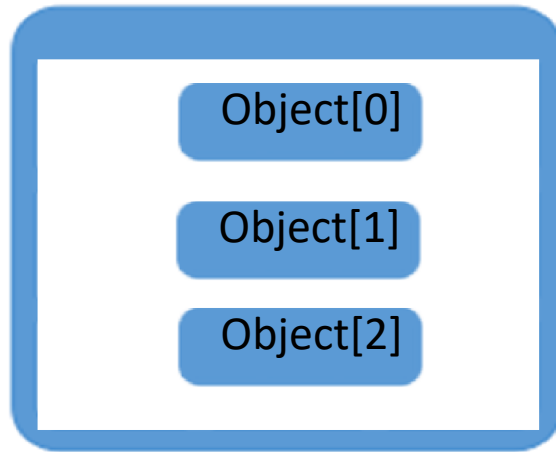
Langages Support



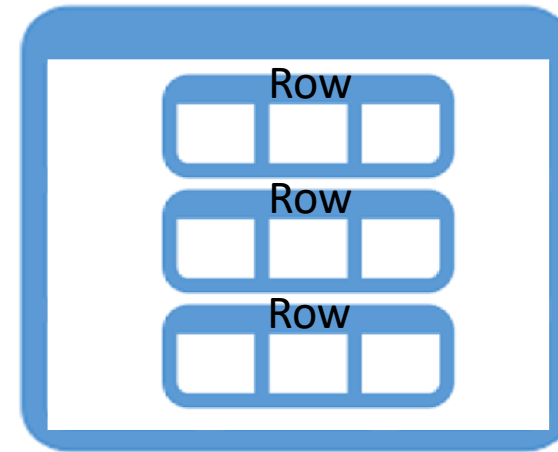
What are Dataset/RDD ?
How they work ?

DataSet ~ Set of <Data>

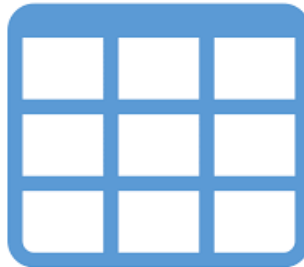
Dataset<UserDefinedClass>



Dataset<Row> = « DataFrame »



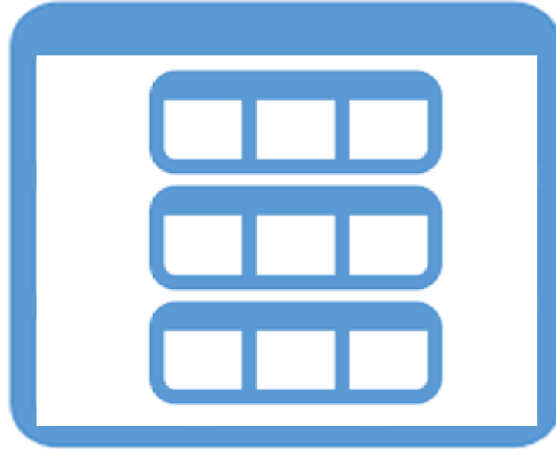
Internal RDD



DataSet = sql view for RDD

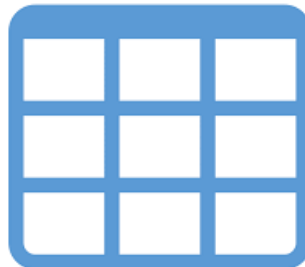
module spark-sql

class org.apache.spark.sql.**Dataset**



module spark core

class org.apache.spark.rdd.**RDD**



RDD = ?

Resilient


Resist to failure

Distributed


Horizontal scale

Dataset

RDD source doc

 Search or jump to... /

Pull requests Issues Marketplace Explore

 **apache / spark** Public

Edit Pins Watch 2.1k Fork 26.3k Star 34.2k

<> Code Pull requests 236 Actions Projects Security Insights

master spark / core / src / main / scala / org / apache / spark / rdd / RDD.scala

Go to file

...

```
54  /**
55   * A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable,
56   * partitioned collection of elements that can be operated on in parallel. This class contains the
57   * basic operations available on all RDDs, such as `map`, `filter`, and `persist`. In addition,
58   * \[\[org.apache.spark.rdd.PairRDDFunctions\]\] contains operations available only on RDDs of key-value
59   * pairs, such as `groupByKey` and `join`;
60   * \[\[org.apache.spark.rdd.DoubleRDDFunctions\]\] contains operations available only on RDDs of
61   * Doubles; and
62   * \[\[org.apache.spark.rdd.SequenceFileRDDFunctions\]\] contains operations available on RDDs that
63   * can be saved as SequenceFiles.
64   * All operations are automatically available on any RDD of the right type (e.g. RDD[(Int, Int)])
65   * through implicit.
66   *
67   * Internally, each RDD is characterized by five main properties:
68   *
69   * - A list of partitions
70   * - A function for computing each split
71   * - A list of dependencies on other RDDs
72   * - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
73   * - Optionally, a list of preferred locations to compute each split on (e.g. block locations for
74   *   an HDFS file)
75   *
76   * All of the scheduling and execution in Spark is done based on these methods, allowing each RDD
77   * to implement its own way of computing itself. Indeed, users can implement custom RDDs (e.g. for
78   * reading data from a new storage system) by overriding these functions. Please refer to the
79   * Spark paper
80   * for more details on RDD internals.
81   */
82  abstract class RDD[T: ClassTag](
83    @transient private var _sc: SparkContext,
84    @transient private var deps: Seq[Dependency[_]]
85  ) extends Serializable with Logging {
```

RDD Doc (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`` (..)

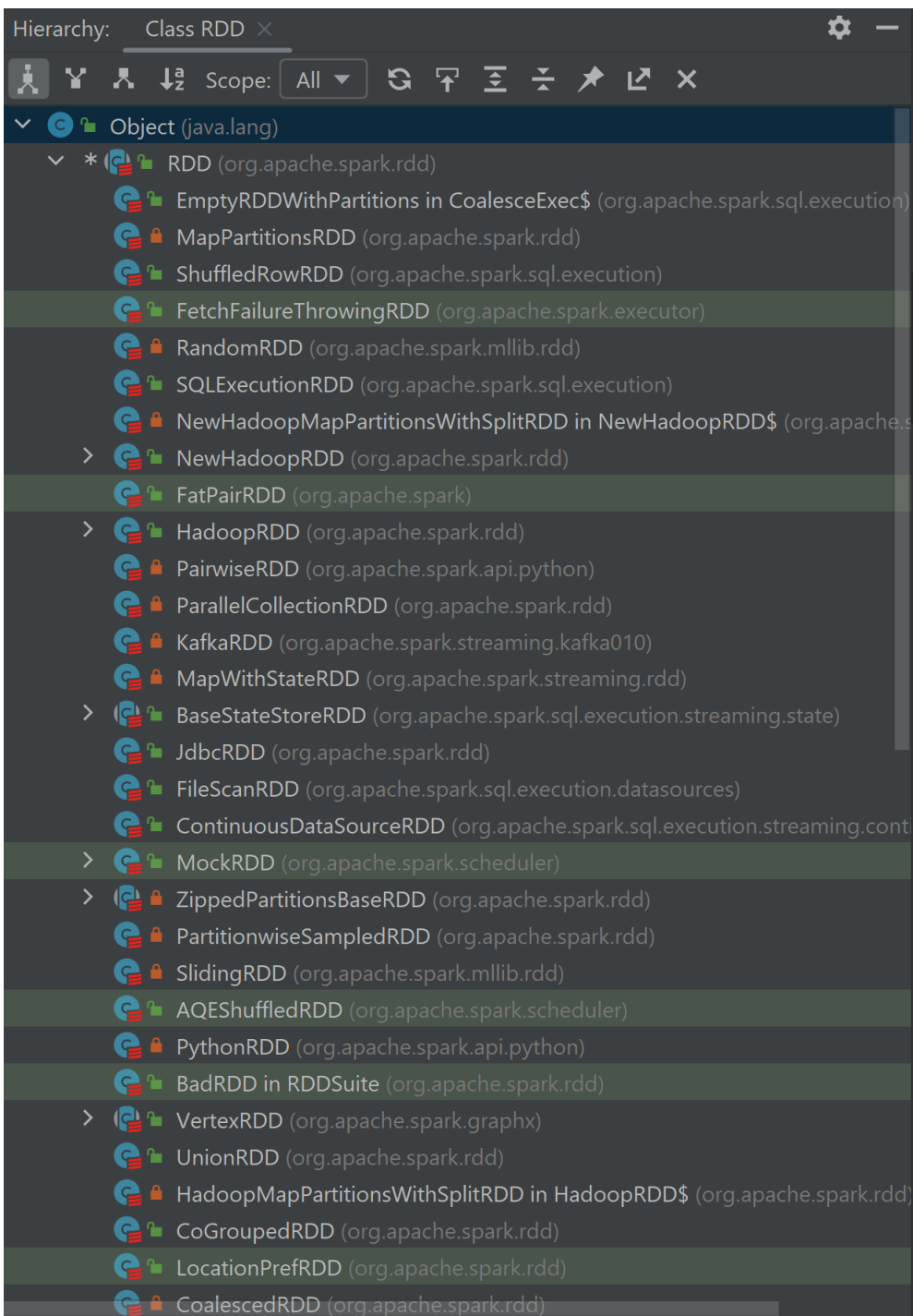
RDD Doc (2/3)

Internally, each RDD is characterized by :

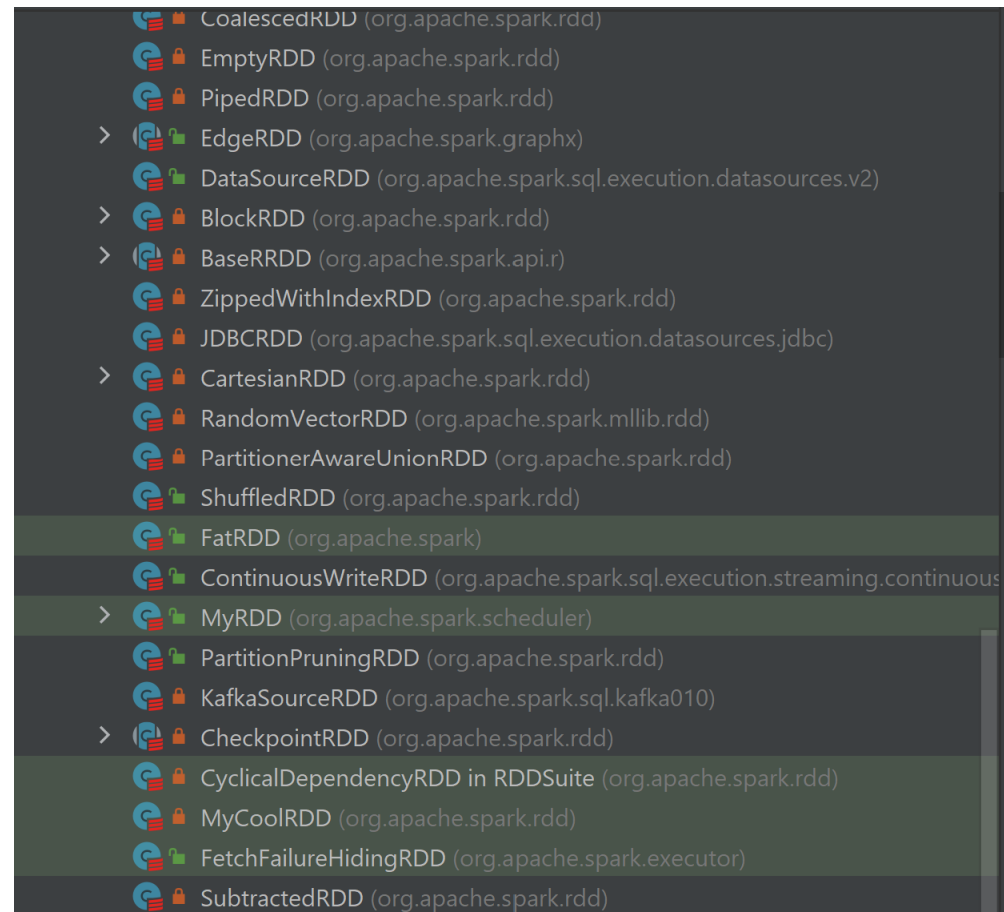
- A **list of partitions**
- A **function for computing** each split
- A list of **dependencies on other RDDs**
- Optionally, a **Partitioner**
- Optionally, a list of **preferred locations**

RDD Abstract methods

```
105 // =====
106 // Methods that should be implemented by subclasses of RDD
107 // =====
108
109 /**
110  * :: DeveloperApi ::
111  * Implemented by subclasses to compute a given partition.
112  */
113 @DeveloperApi
114 def compute(split: Partition, context: TaskContext): Iterator[T]
115
116 /**
117  * Implemented by subclasses to return the set of partitions in this RDD. This method will only
118  * be called once, so it is safe to implement a time-consuming computation in it.
119  *
120  * The partitions in this array must satisfy the following property:
121  * `rdd.partitions.zipWithIndex.forall { case (partition, index) => partition.index == index }`
122  */
123 protected def getPartitions: Array[Partition]
124
125 /**
126  * Implemented by subclasses to return how this RDD depends on parent RDDs. This method will only
127  * be called once, so it is safe to implement a time-consuming computation in it.
128  */
129 protected def getDependencies: Seq[Dependency[_]] = deps
130
131 /**
132  * Optionally overridden by subclasses to specify placement preferences.
133  */
134 protected def getPreferredLocations(split: Partition): Seq[String] = Nil
135
136 /** Optionally overridden by subclasses to specify how they are partitioned. */
137 @transient val partitioner: Option[Partitioner] = None
```



Abstract RDD class
=> (many) concrete sub-classes



1 algorithm / transformation => 1 RDD Sub-class

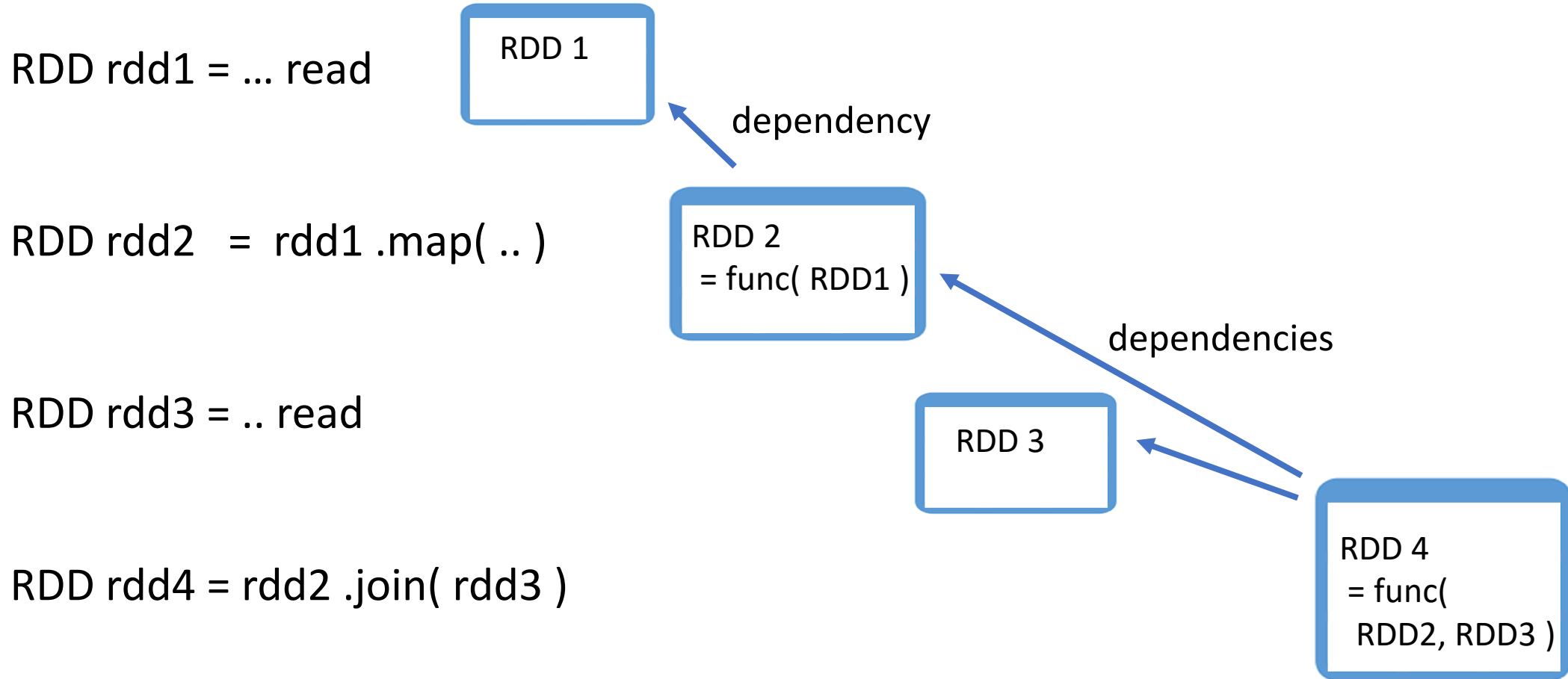
Example: `rdd.map(func)` or `rdd.flatMap(func)`

```
/**
 * 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))
}

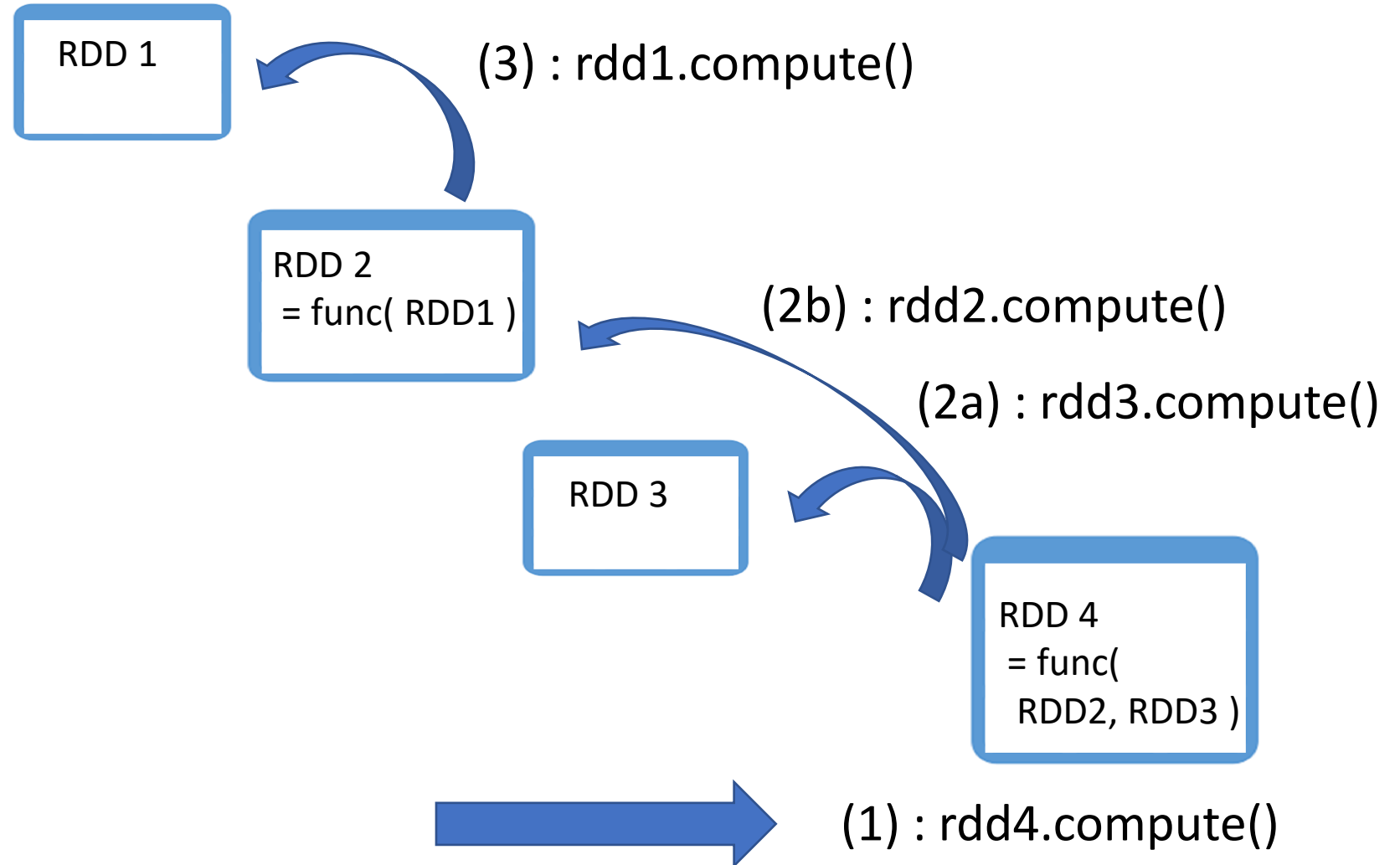
/**
 * Return a new RDD by first applying a function to all elements of this
 * RDD, and then flattening the results.
 */
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U] = withScope {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[U, T](prev = this, (_, _, iter) => iter.flatMap(cleanF))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

```
/**
 * 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) {
```

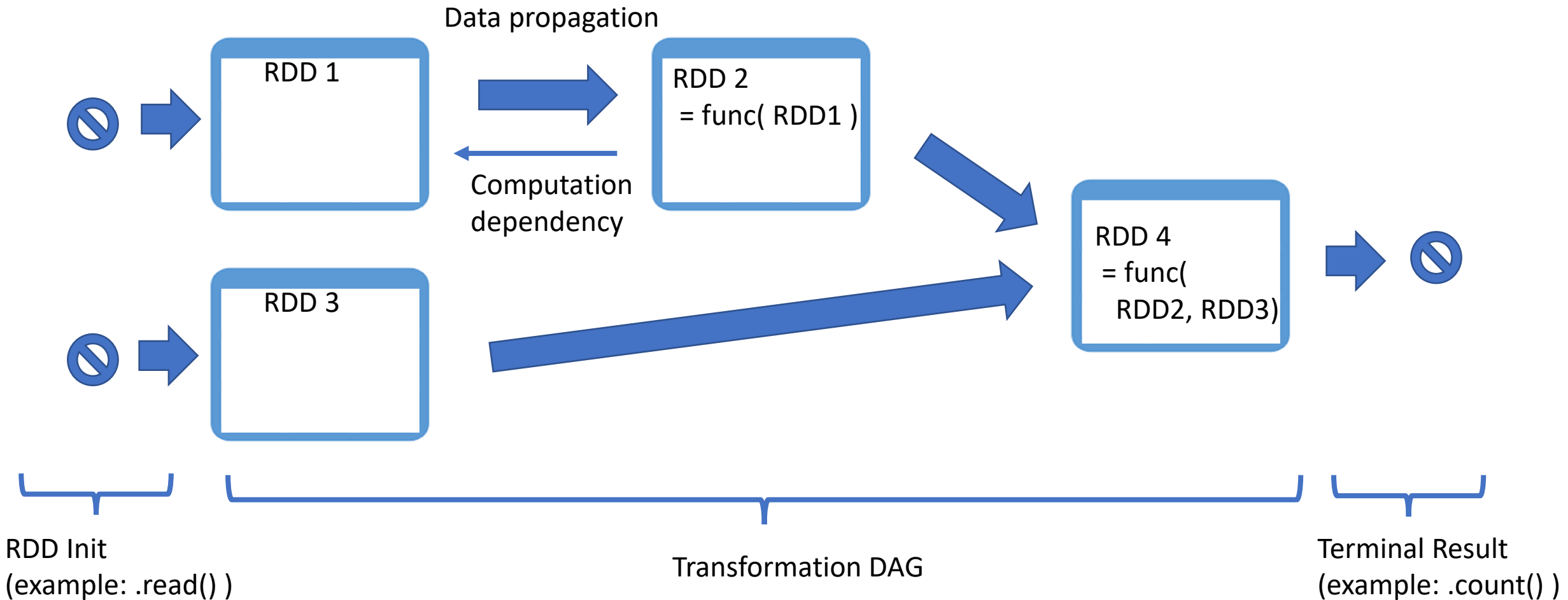

Call transform function => Create new RDD (linked)



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



RDD Dependencies : DAG (Directed Acyclic Graph)



3 equivalent formalisms:

SSA create Api, Expression Algebra, DAG

SSA = Single State Assignments

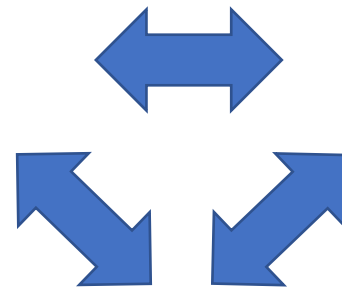
RDD API

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

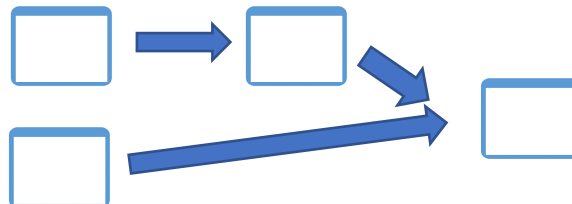
Expression Algebra, Sql

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

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



DAG



RDD Doc (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

1 / 14



67%



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 serializa-

tion, 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.

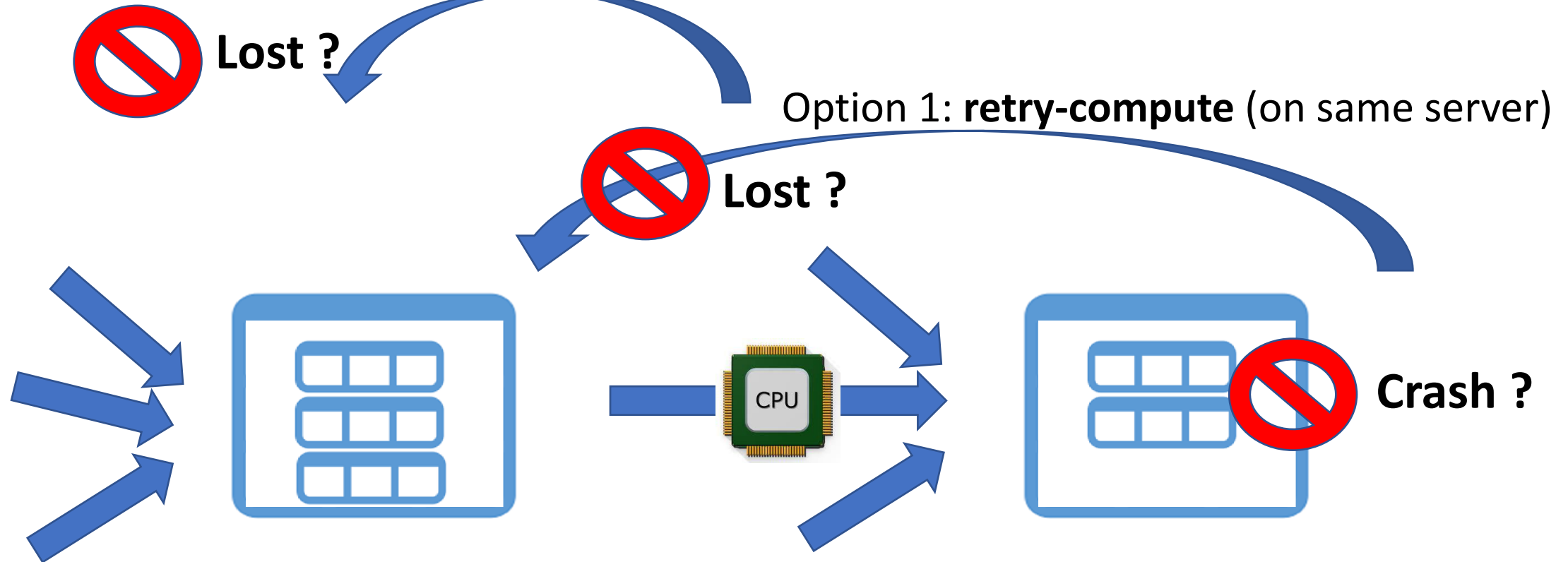
In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) 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

¹Checkpointing 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.

Fault Tolerant - Computation

Option 3: **recompute dependency** sources

Option 2: find sources dependency **backup Copy** (on different server / storage)



CoarseGrain ... Scheduler/Executor

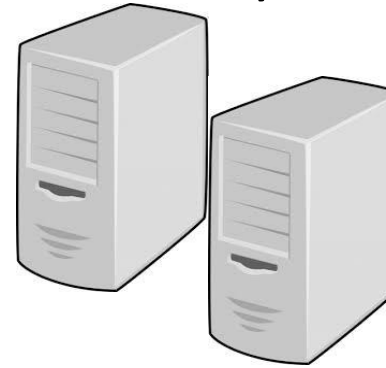
Spark-driver

Implements Fault Tolerance+Distribution
... internally called « CoarseGrainScheduler »



Spark-executor

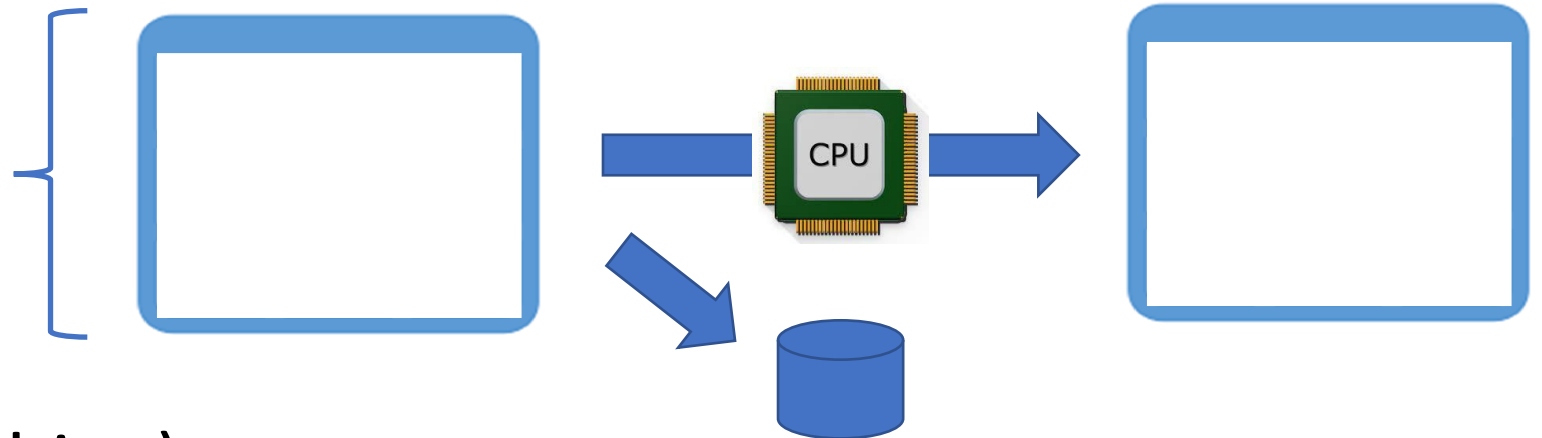
Implements task main loop
... internally called « CoarseGrainExecutor »



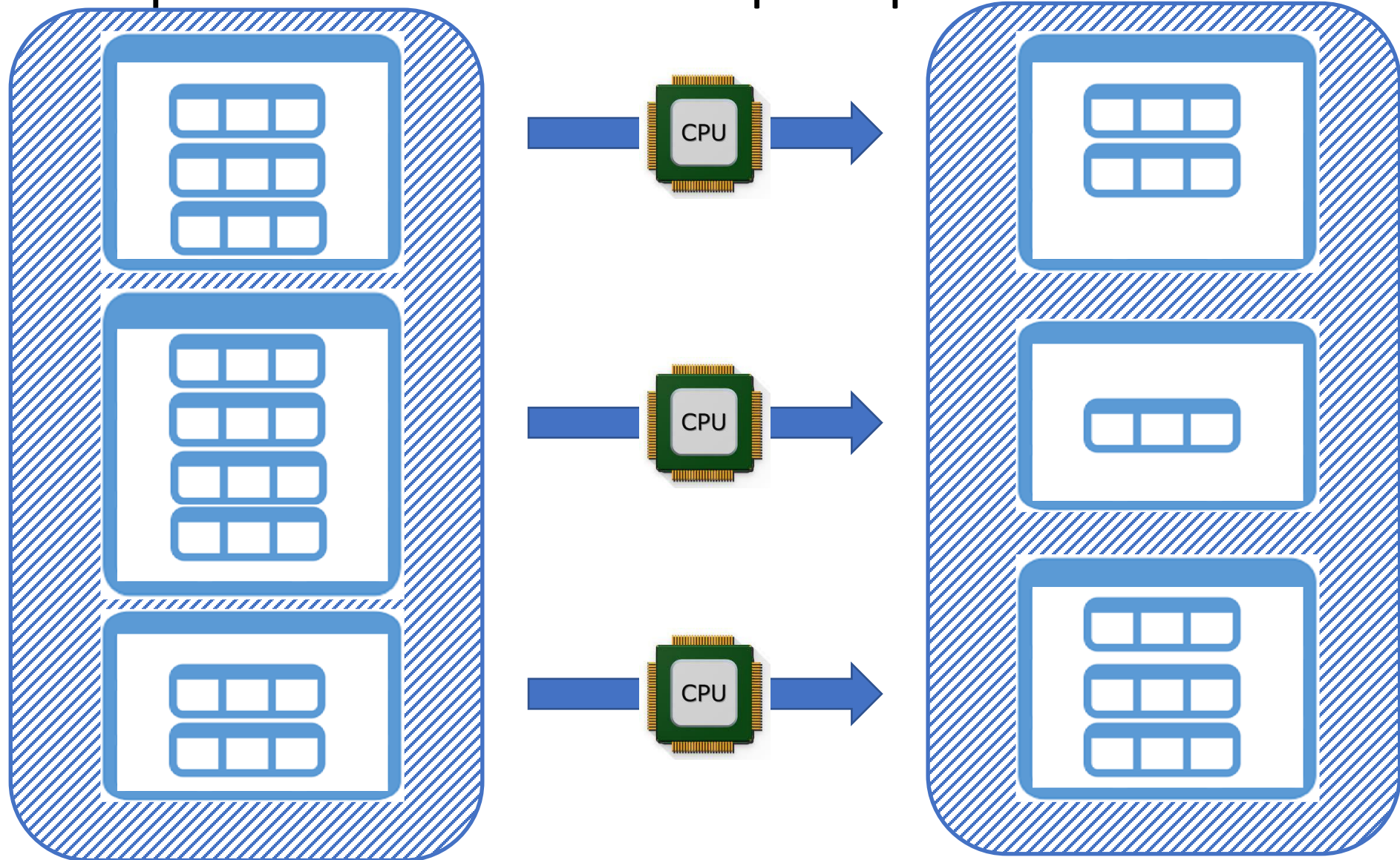
CoarseGrain

partition

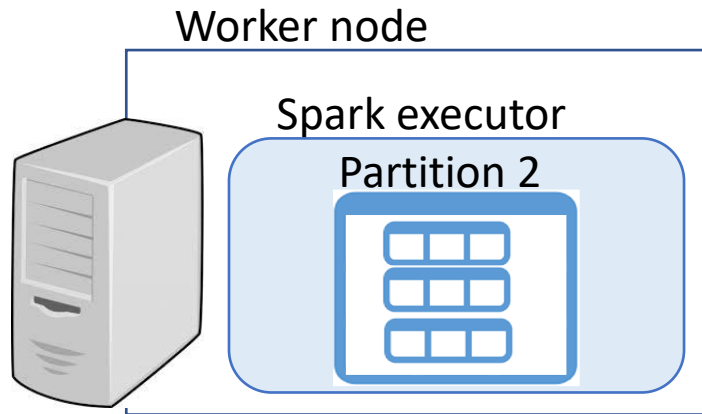
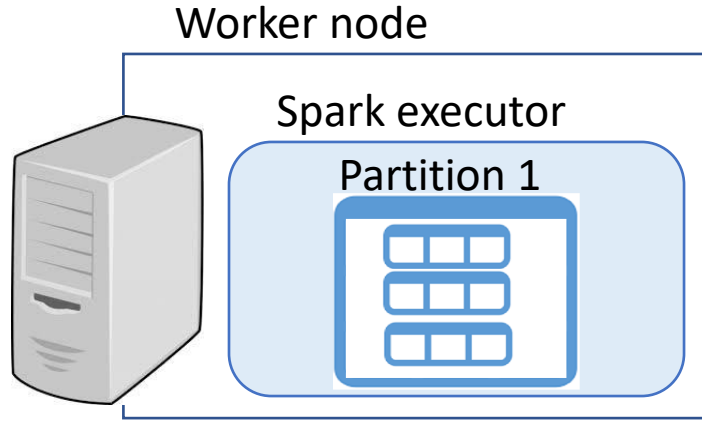
= unit of caching/
recomputation
(all elements or nothing)



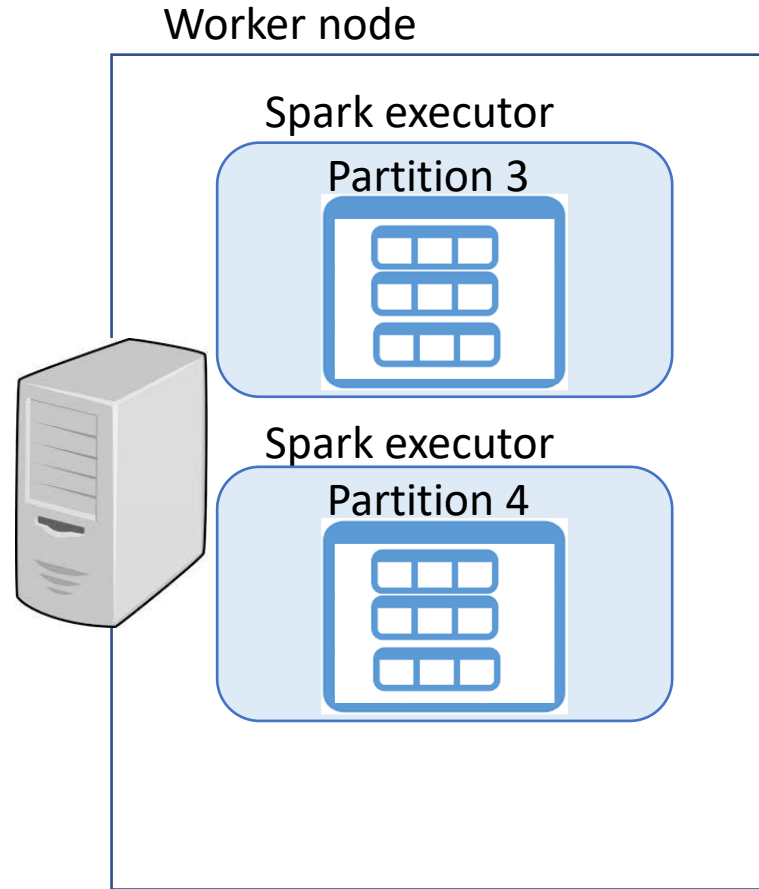
DataSet : Collection of Objects,
parallelize 1 CPU per partition



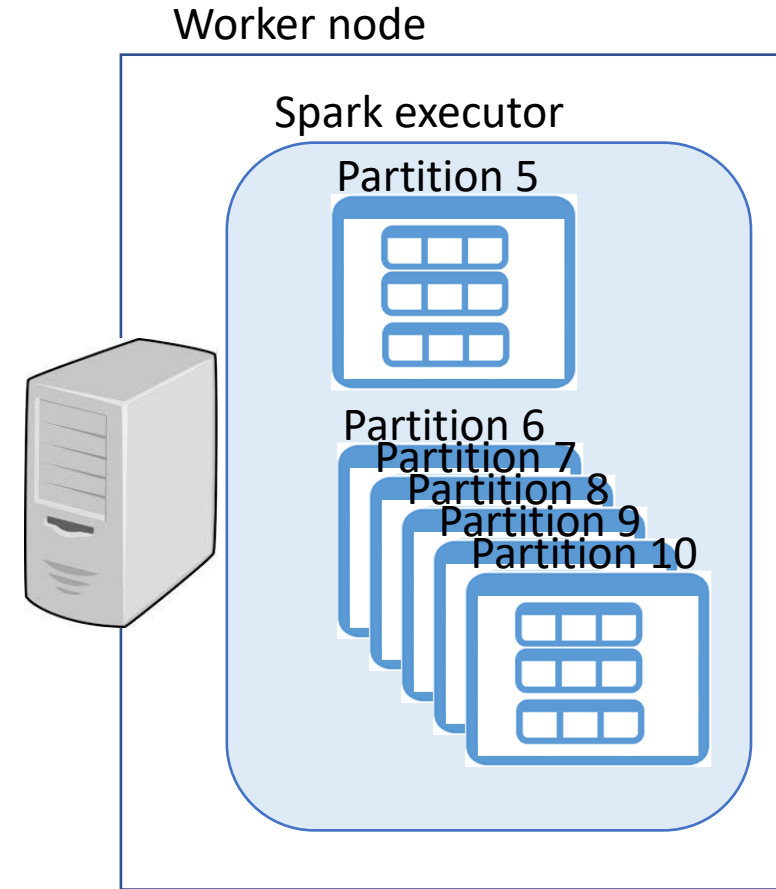
Distribution: Partition < Executor < Node



several worker nodes



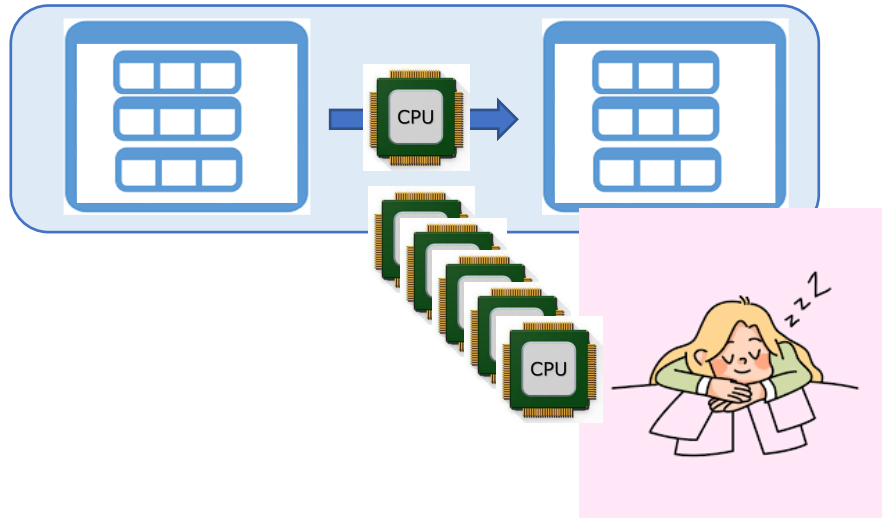
Several spark-executor
processes per nodes



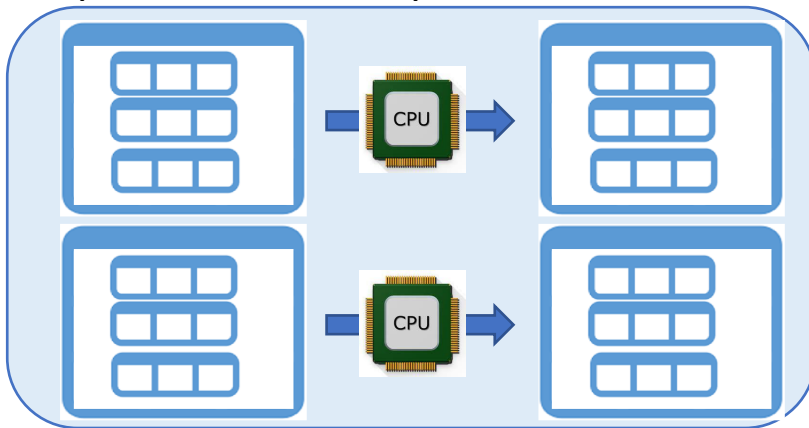
Several partitions
Per spark-executor

Optimize parallelism: Adapt partitions to number of Cores

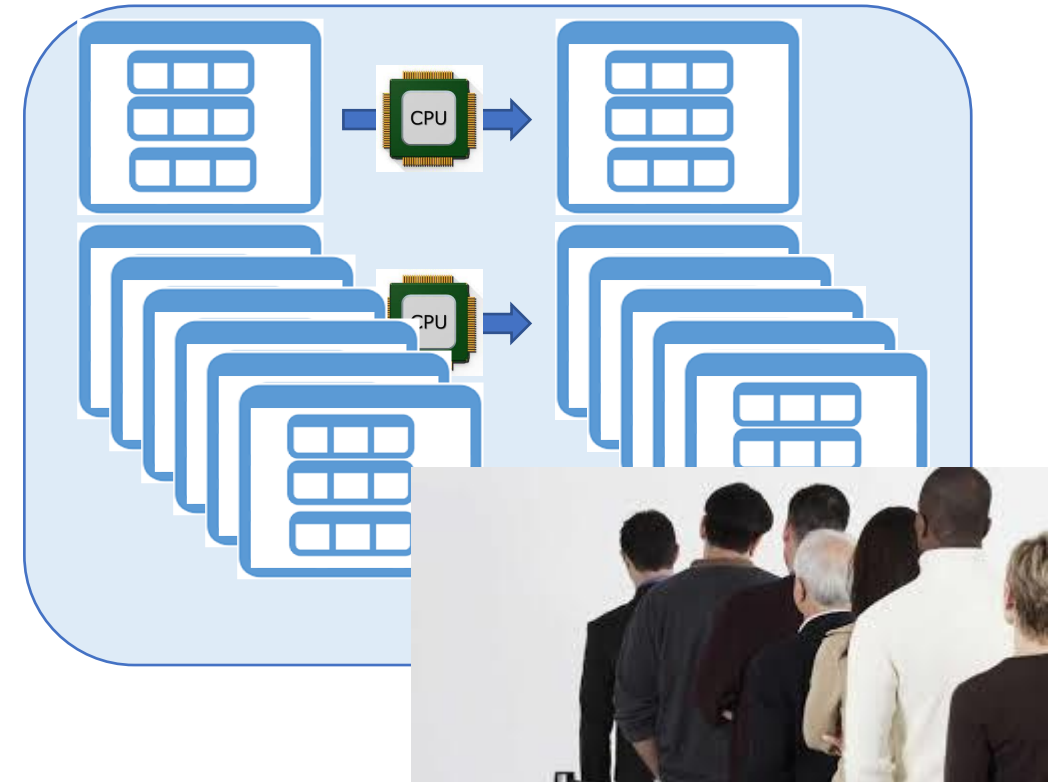
Spark-executor: 1 partition \ll N cores



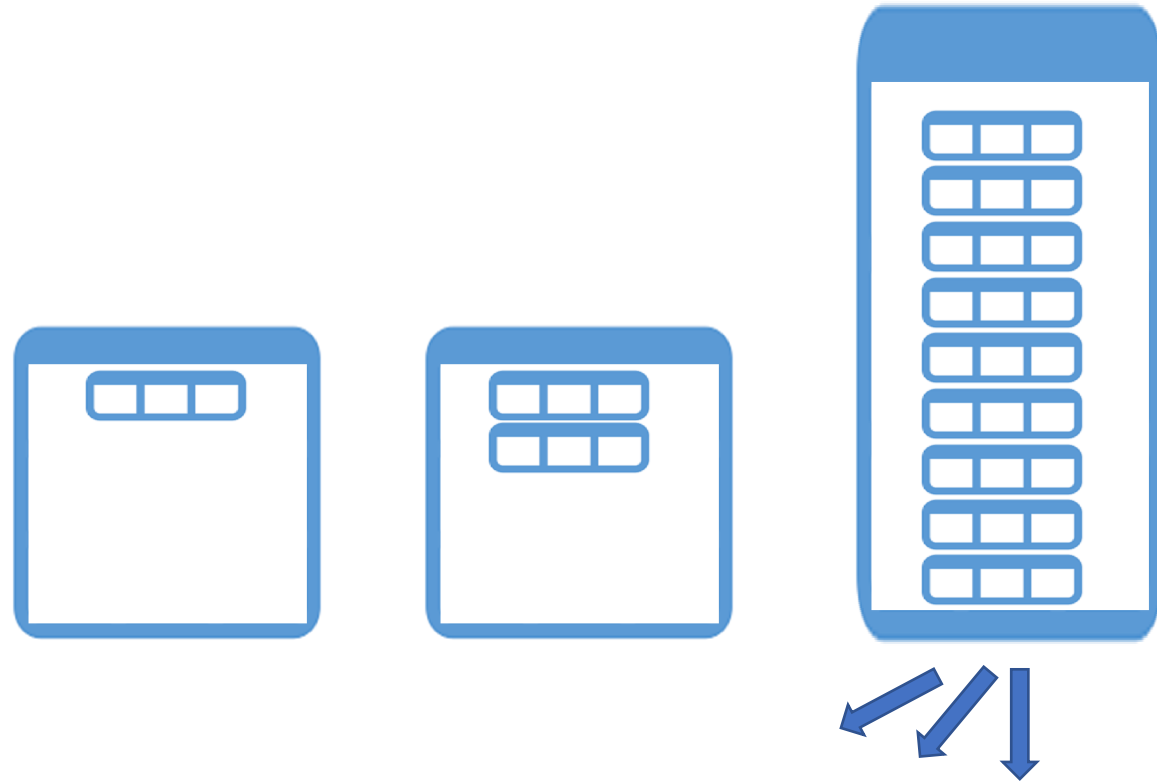
Spark executor: N partitions \sim N cores



Spark executor: N partitions \gg few cores

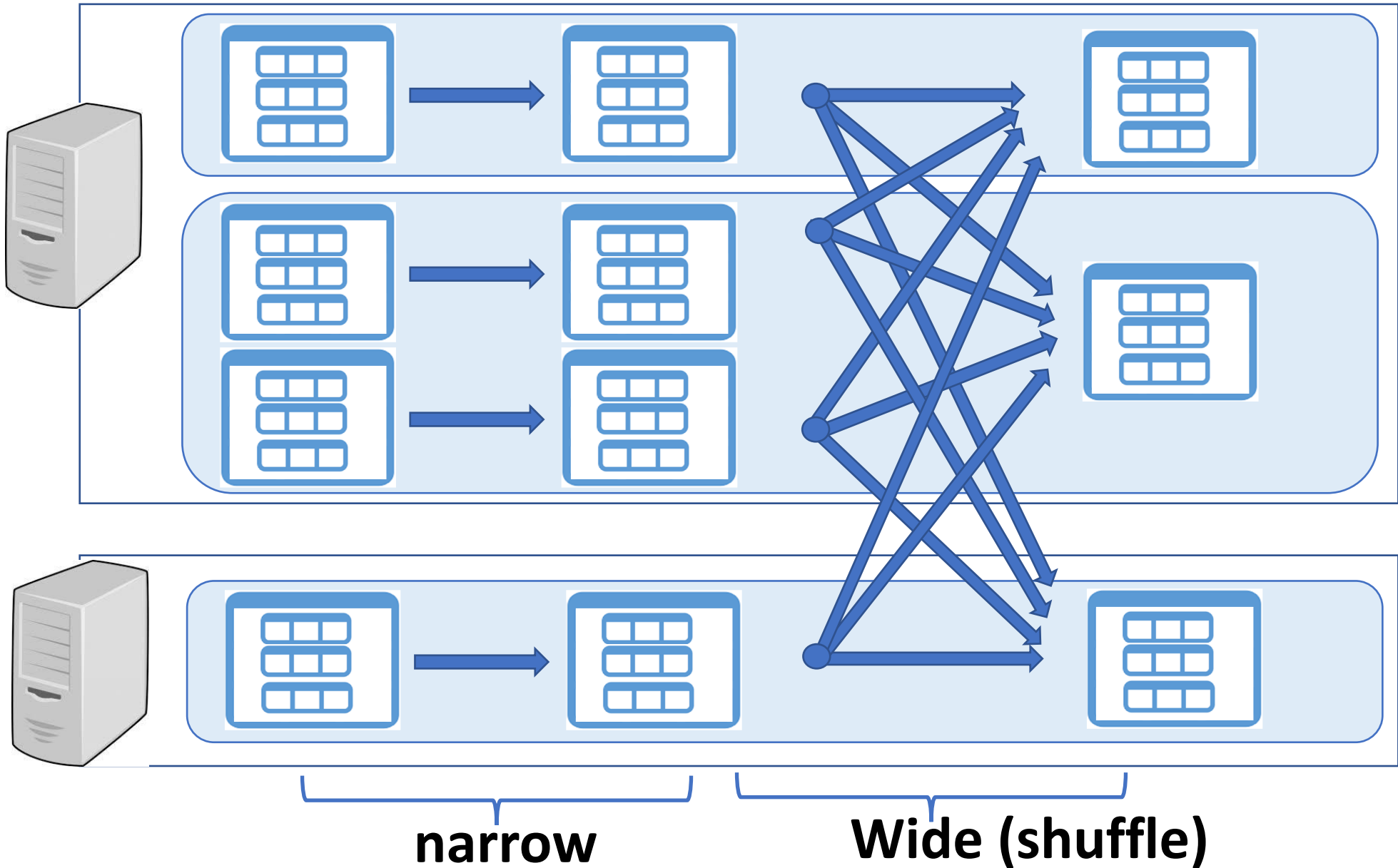


Skewed Data ... need Repartitioned equally



Target: move each row[i] to node[j] $j = \text{« rowId modulo } N \text{ »}$

« Narrow » / « Wide » Transformations



RDD ... OK
so What is a Dataset ?

Dataset source doc

[Pull requests](#)[Issues](#)[Marketplace](#)[Explore](#)[apache / spark](#)Public[Edit Pins](#)[Watch](#) 2.1k[Fork](#) 26.2k[Star](#) 34.2k[Code](#)[Pull requests](#) 238[Actions](#)[Projects](#)[Security](#)[Insights](#)[master](#)[spark / sql / core / src / main / scala / org / apache / spark / sql / Dataset.scala](#)[Go to file](#)[...](#)

```
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.
126   *
```

Dataset Doc (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 Doc (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

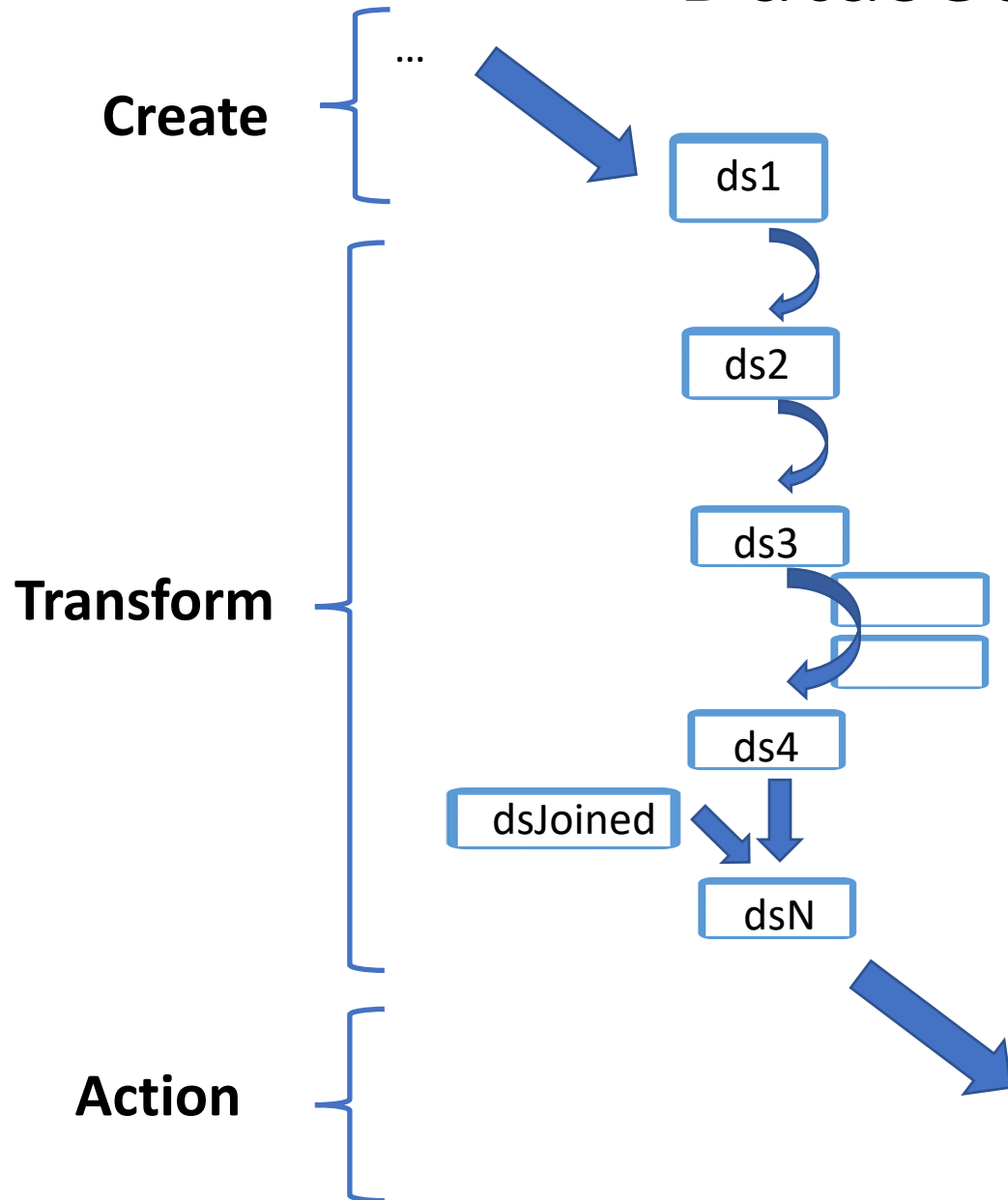
dataset = ds .filter(..) .filter(..) .map(..) .groupBy(..),

Action

<Result> = dataset. <action> ();

example: long result = dataset. count ();

Dataset operations...



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

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

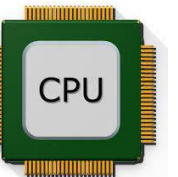
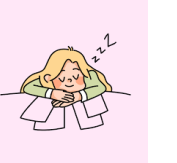
```
Dataset<Row> ds3 = ds2.toDF();
```

```
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 !!
```



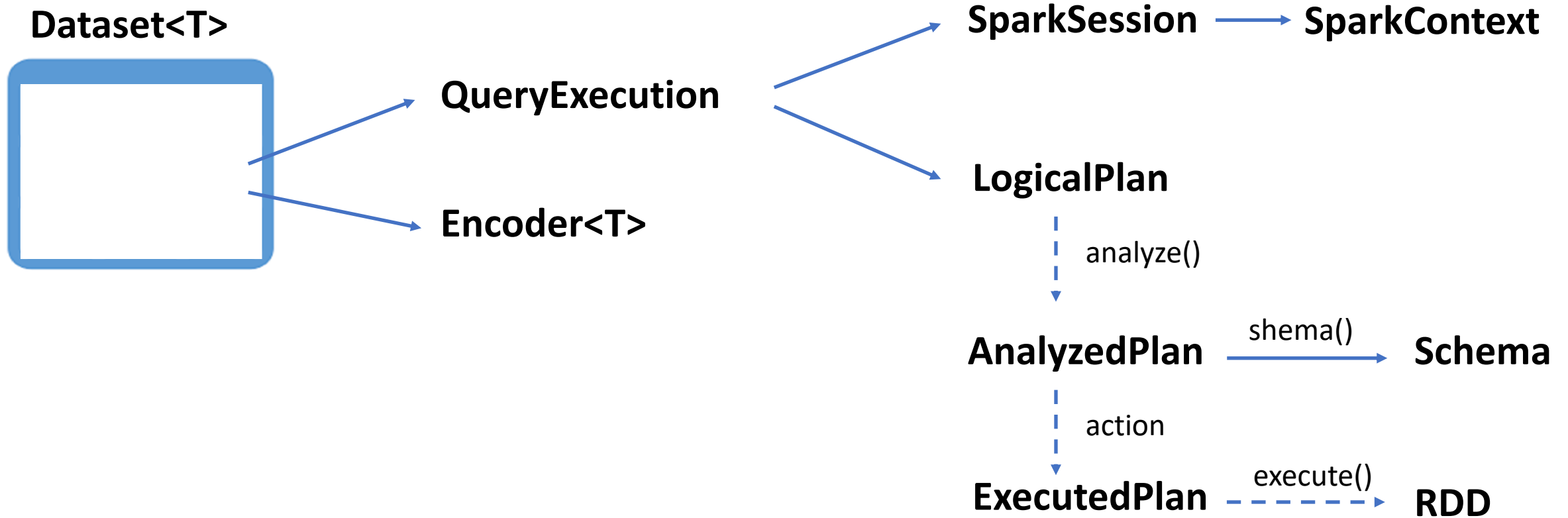
Dataset Doc (3/4)

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 Doc 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

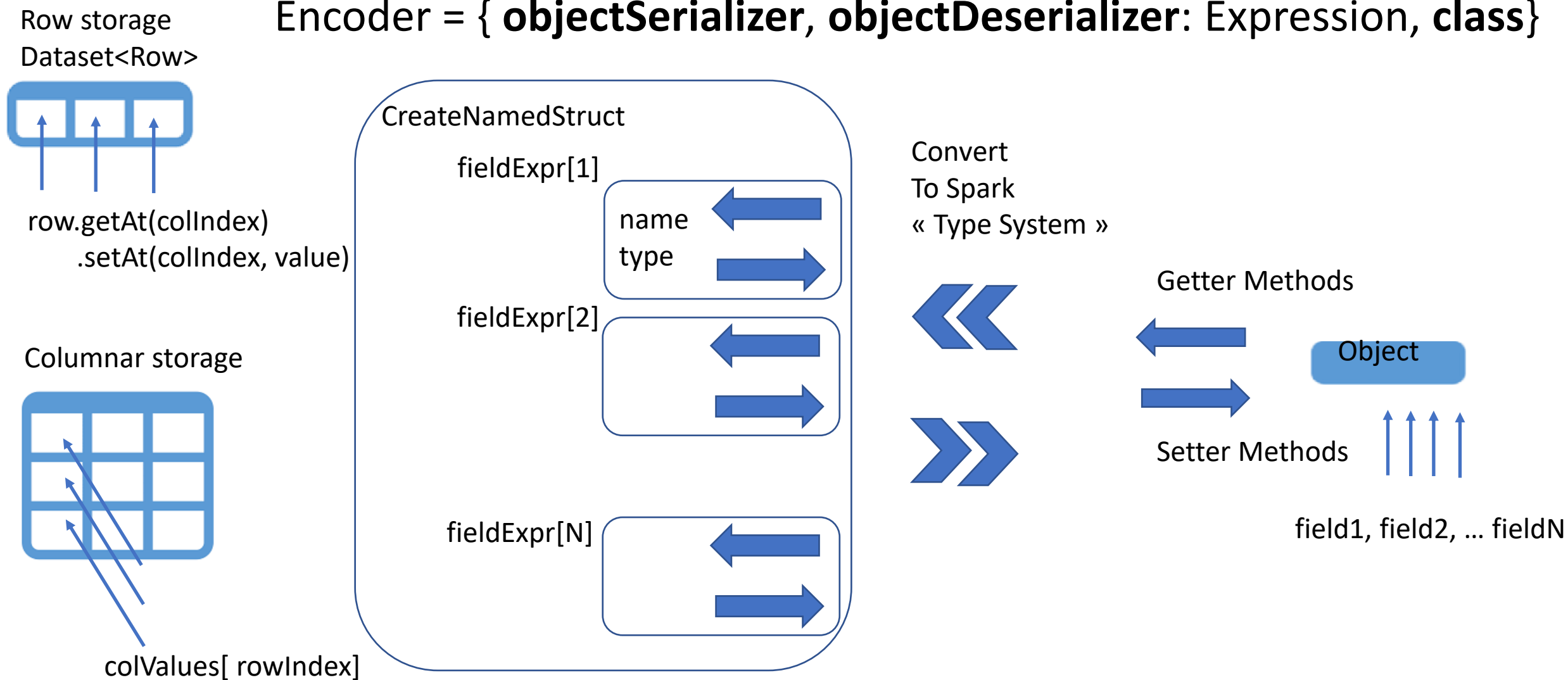
- (..) with lower memory footprint

- (..) optimized for processing (e.g. in a columnar format).

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

Encoder<T>

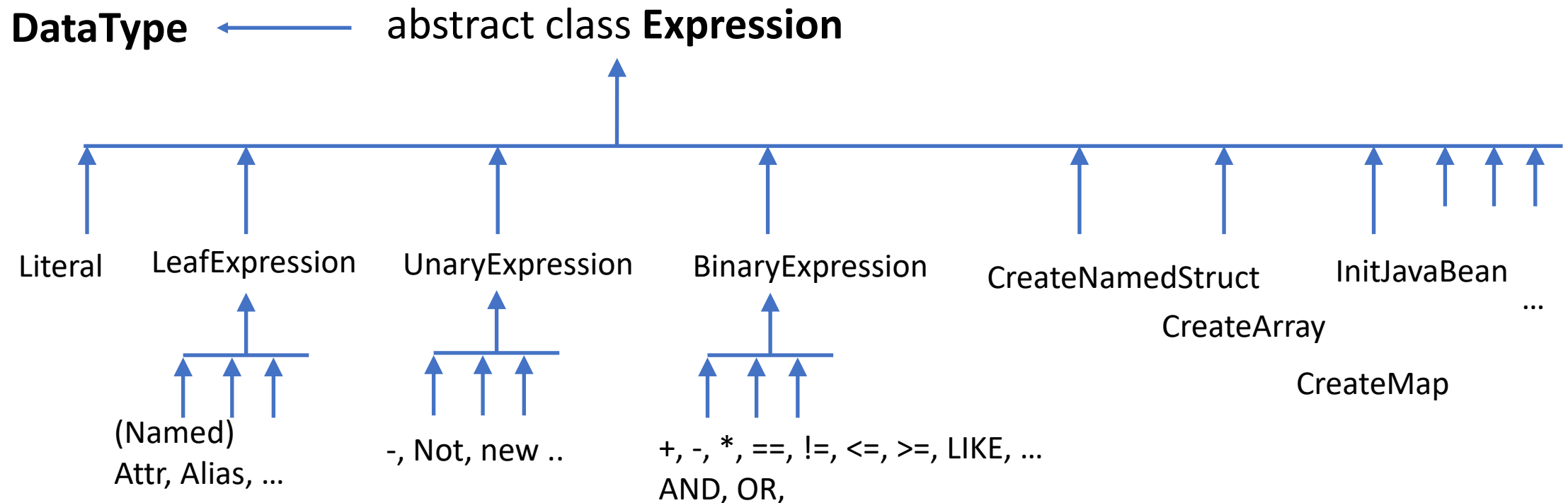
Encoder = { **objectSerializer**, **objectDeserializer**: Expression, **class** }



Spark « Type System » DataTypes ...

Encoder .. Internal Expression with DataType

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



Expression ...

Complex ... INTERNAL ...

(not need to understand, to use Spark)

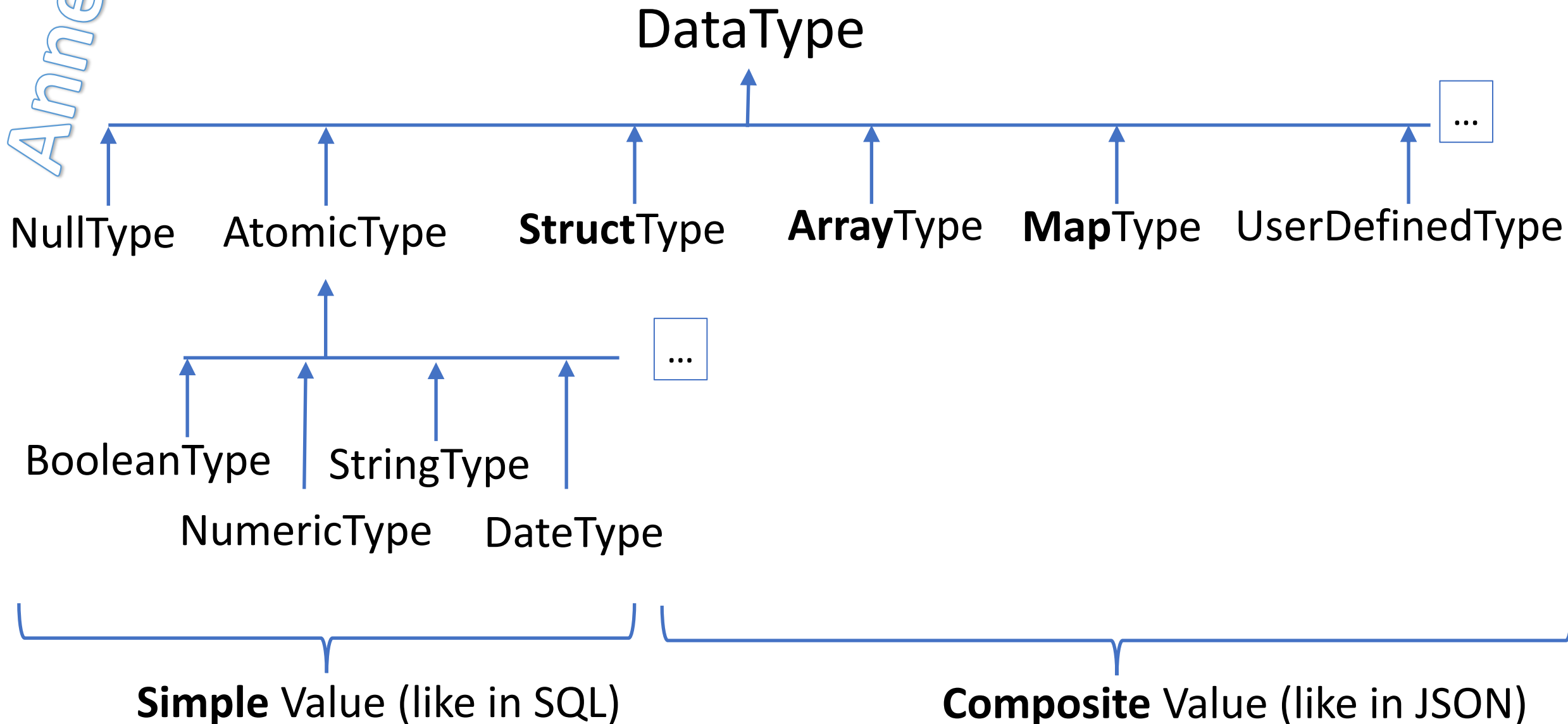
Looks like a « **SQL Compiler** »

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

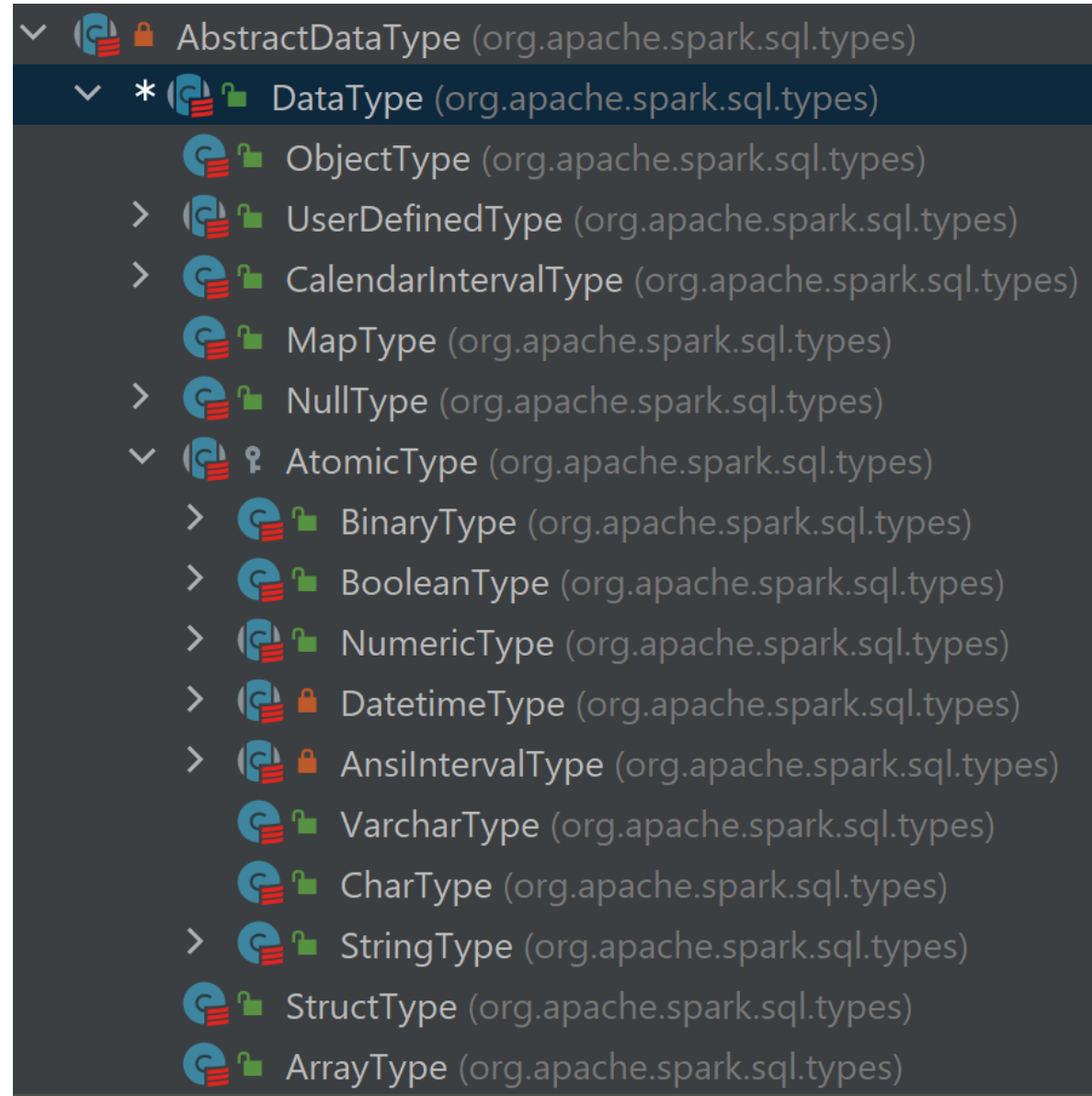
Used internally to **compile + generate code**

Internal Spark « Type System »

Annexe



Spark DataType

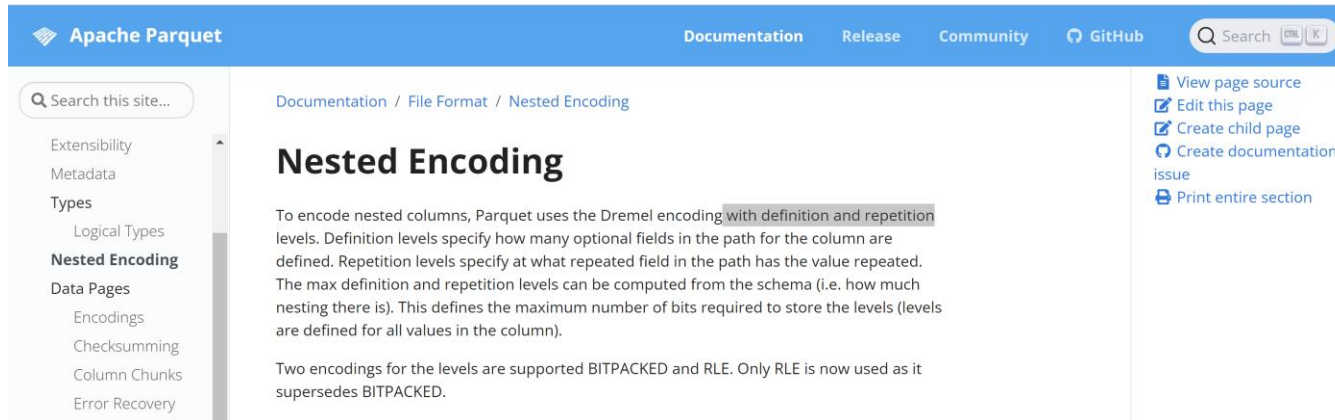


Hive - Spark SQL supports Struct, List, Map ...

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 > >  
)
```

Nested fields in File Format: Parquet / Orc / Json



Parquet DataType ~ **Spark DataType**
Nested Encoding with « definiton » + « repetition »

JSON DataType ~ **Spark DataType**
(map with string only)



Nested Fields in Spark SQL UDF

```
SELECT ename, dept_list
FROM employee
```

| ename | dept_list |
|-------|------------|
| Tom | [20] |
| Jerry | [10,20] |
| Riley | [20,30,40] |

```
SELECT ename,
       exists(dept_list, x -> x = 10) as found10
FROM employee
```

| ename | found10 |
|-------|---------|
| Tom | false |
| Jerry | true |
| Riley | false |

SQL Grammar Extension: « lateral view »

```
SELECT ename, dept_list
FROM employee
```

| ename | dept_list |
|-------|------------|
| Tom | [20] |
| Jerry | [10,20] |
| Riley | [20,30,40] |

```
SELECT ename, dept_id
FROM employee
LATERAL VIEW explode(dept_list) depts AS dept_id;
```

| ename | dept_id |
|-------|---------|
| Tom | 20 |
| Jerry | 10 |
| Jerry | 20 |
| Riley | 20 |
| Riley | 30 |
| Riley | 40 |

More SQL: collect_list(row) -> List

```
SELECT ename, dept_list
FROM employee
```

| ename | dept_list |
|-------|------------|
| Tom | [20] |
| Jerry | [10,20] |
| Riley | [20,30,40] |

```
SELECT ename, collect_list(dept_id + 1) as ls
FROM ( SELECT employee
        LATERAL VIEW explode(dept_list) depts AS dept_id )
GROUP BY ename
```

| ename | ls |
|-------|------------|
| Tom | [21] |
| Jerry | [11,21] |
| Riley | [21,31,41] |

UDF / UDAF (User Defined Aggregate Function)

Example UDF : $f(x, y) \{ \text{return } x + y \}$

Function like in Math : idempotent, side-effect less, ..

! = UDAF : Aggregate / Accumulator

like in « SELECT count(..), sum(..), average(..) FROM .. GROUP BY .. »

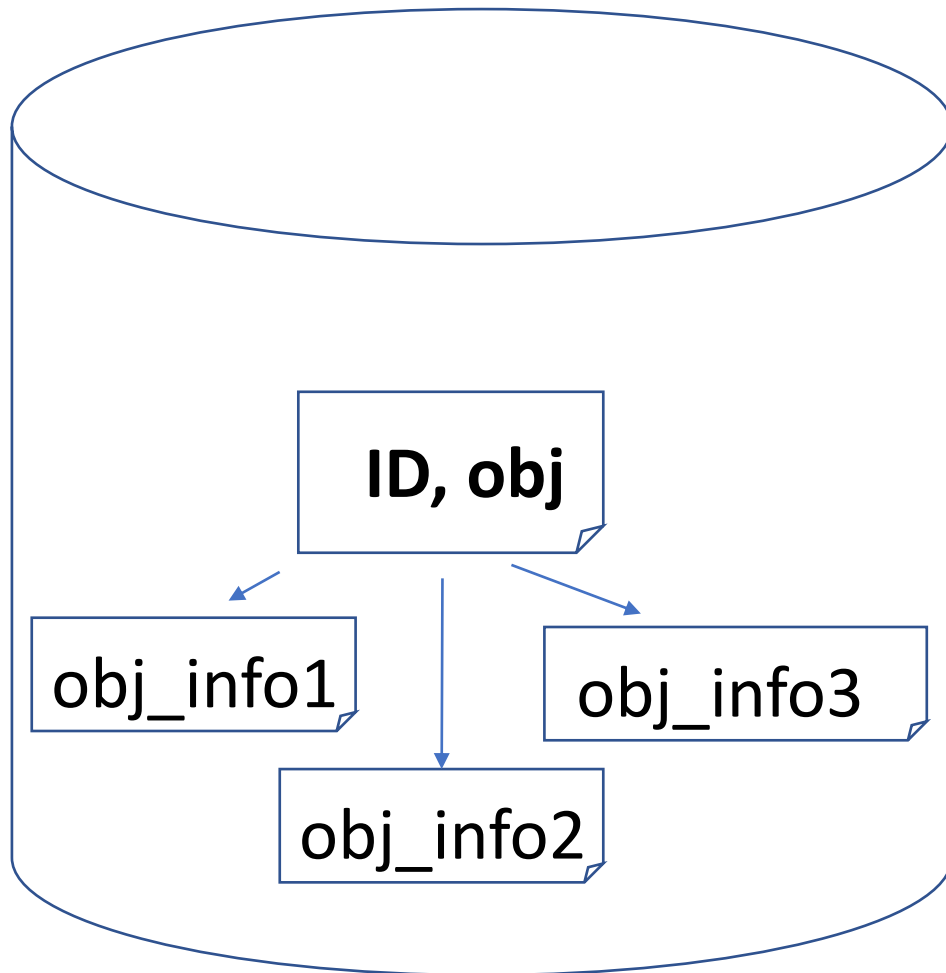
Object instance, Class with 3 methods:

- init()

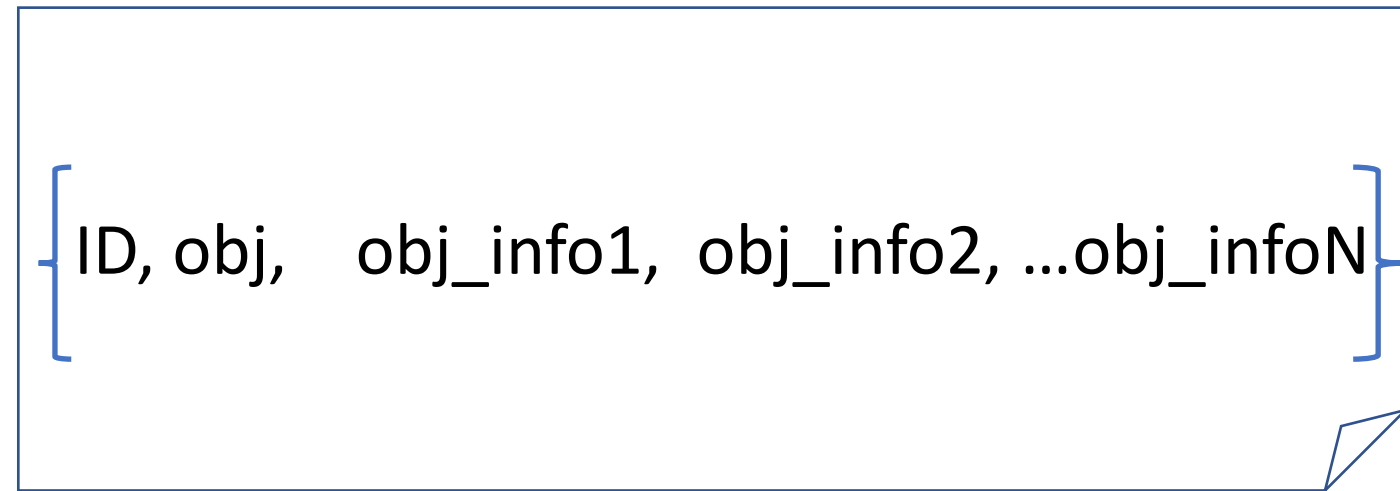
- add(value)

- Result getResult()

List, Map, Struct ... denormalize data, avoid Joins



Normalized relational database



Efficient DE-normalized analytics system

Enough for Today !

... but more to come

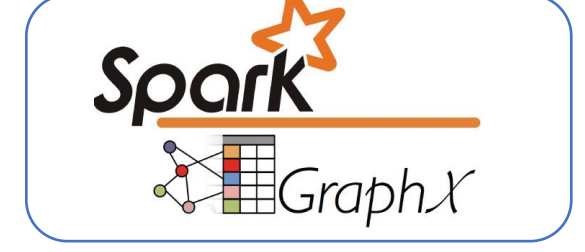
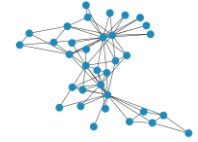
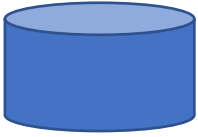
Questions are Welcome

Take Away

What Did you learn ?

Spark-Core + ...

Structured
Data



Modules



Amazon S3



Azure Data Lake Storage Gen2

DataSource Connectors
(Hadoop API)



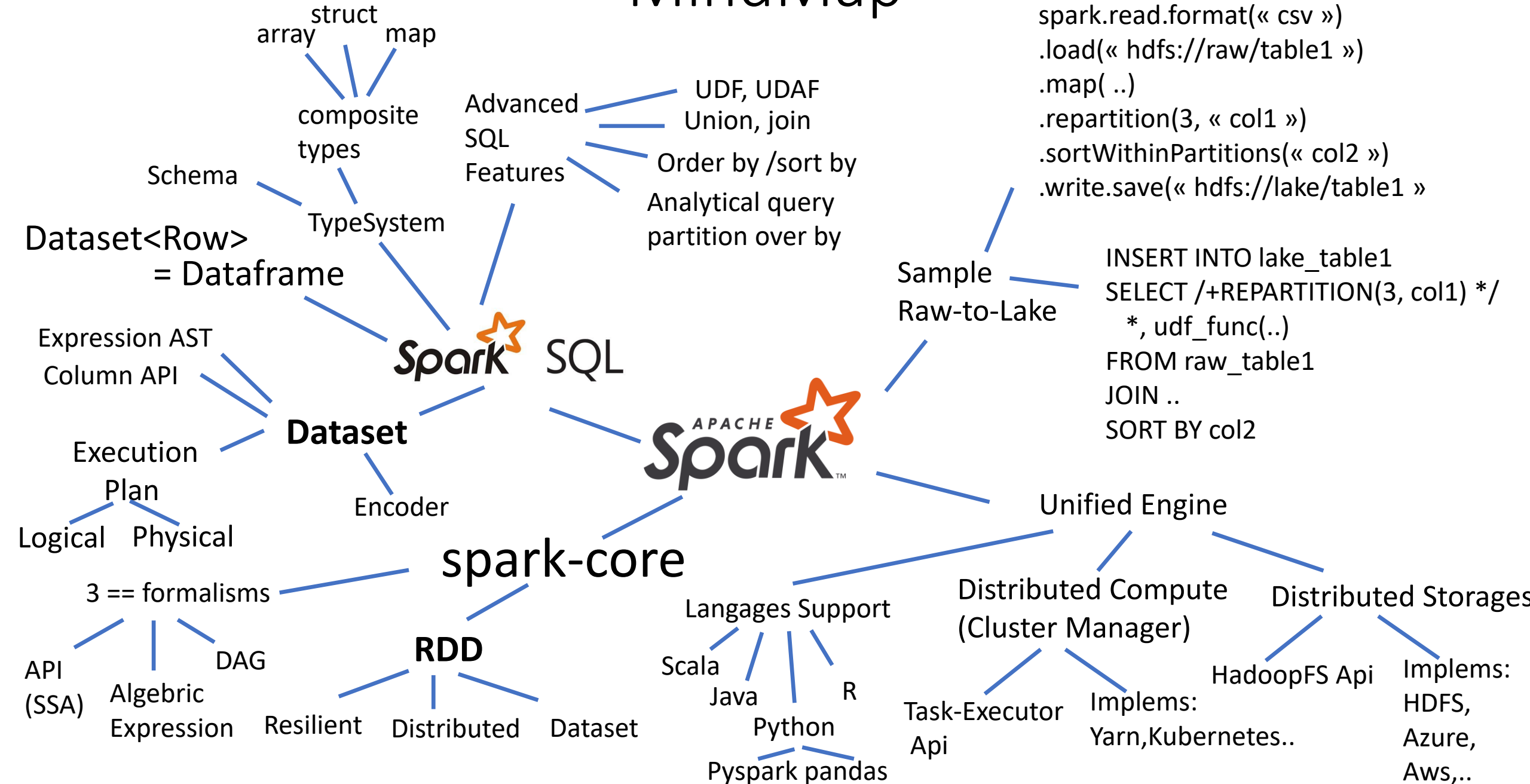
Cluster Manager



Langages Support



MindMap



Next..

Lesson 1 : introduction to BigData / Hadoop cluster / Spark



Lesson 2 : this document : spark-core / spark-sql

Lesson 3 : cluster management, tuning args, deploy-mode

Lesson 4 : Advanced Spark: UI, parquet PPD, java api, spark-streaming, ..