Spark Unified Engine Features (Sql, Dataset & Api)

arnaud.nauwynck@gmail.com

Course Esilv 2024

This document:

https://github.com/Arnaud-Nauwynck/presentations/
/pres-bigdata/10-Spark-unified-engine-features

Outline

Example RAW to LAKE transformations

Explanation step-by-step

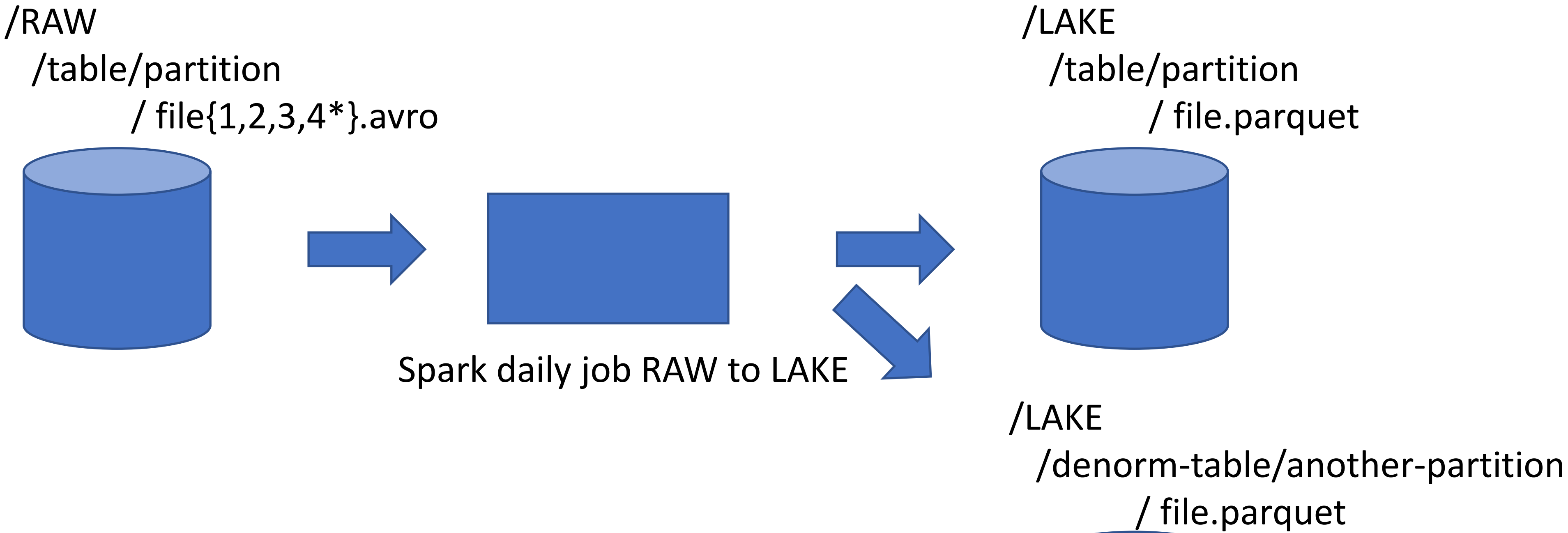
Interaction Files <-> Sql <-> Java DataSets

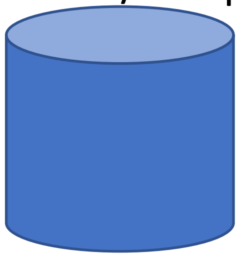
Dataset

Parallel Distribution

Reminder: Spark RAW to LAKE samples

Typical Usage: process RAW to LAKE





```
spark.sql(« select * from raw_table where day=…  join .. »)
   .map(row => …)
   .orderBy(« .. »)
   .write.mode(SaveMode.Overwrite)
   .sortWithinPartition(« .. »)
   .format(« hive »).insertInto(« lake_table »);
```

Typical RAW to LAKE as Spark Java code

Reminder

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 »);
```

Reminder

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,
    ..
FROM
       raw_team_domain.table
JOIN
       lake_anotherTeam_domain.anotherTable x ON x.ID=id
WHERE date='2022-10-22' AND ..
SORT BY col1, col2, col3     -- idem sortWithinPartition
```

transform

read

transform

read

write

Reminder

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
```

Reminder

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

Reminder

read Step **1/4**

transform Step **2/4**

Step **3/4**

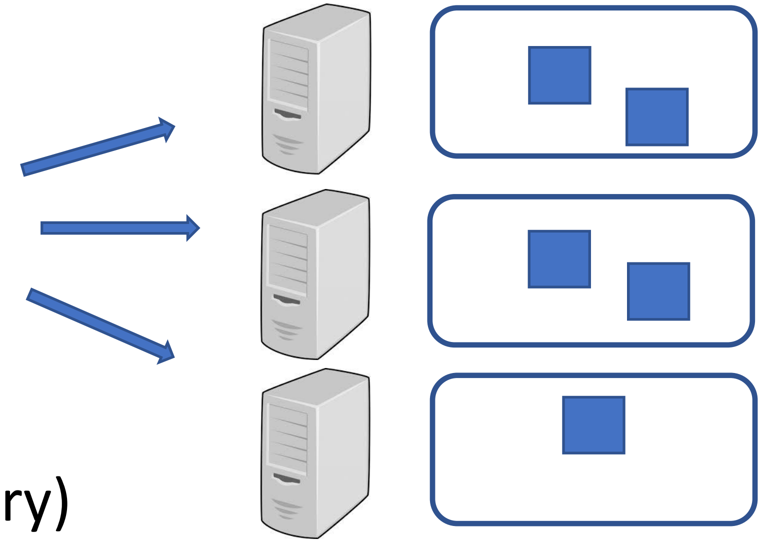
write Step **4/4**

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

   .as(Encoder.bean(Bean.Class)
   .map(bean ->  transformBean(bean) )
   .toDF()
   .repartition(3, « col1 »)
   .sortWithinPartition(« col1, col2, col3 »)

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

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

read

```
Dataset<Row> ds =
    spark.read
    .format(« csv »)
    .option(«schema », « col1 type1, … colN typeN »)
    .load(« hdfs://raw/team/domain/table/date=2022-10-12 »)
```

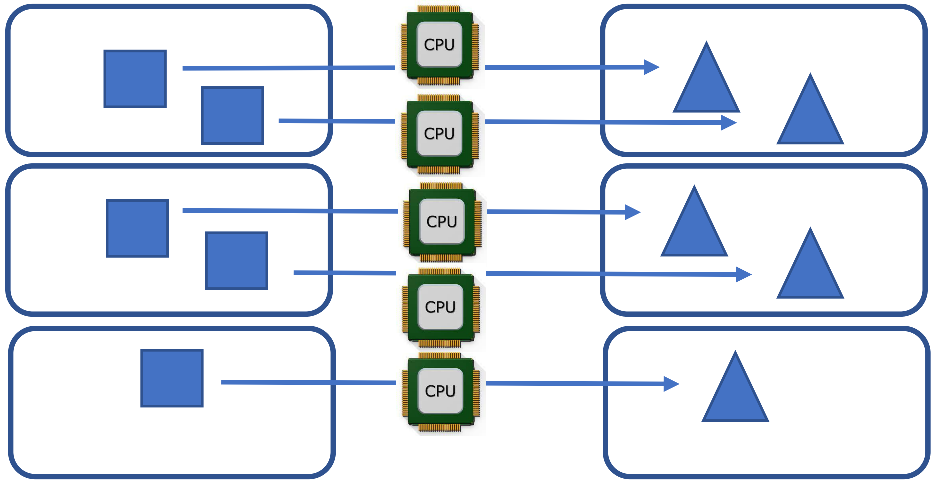


Input = Files
(scanned from Directory)
Distributed Read from Storage

Result =
Distributed Parts in-memory

RAW to LAKE – Step 2/4 : Transform Dataset

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

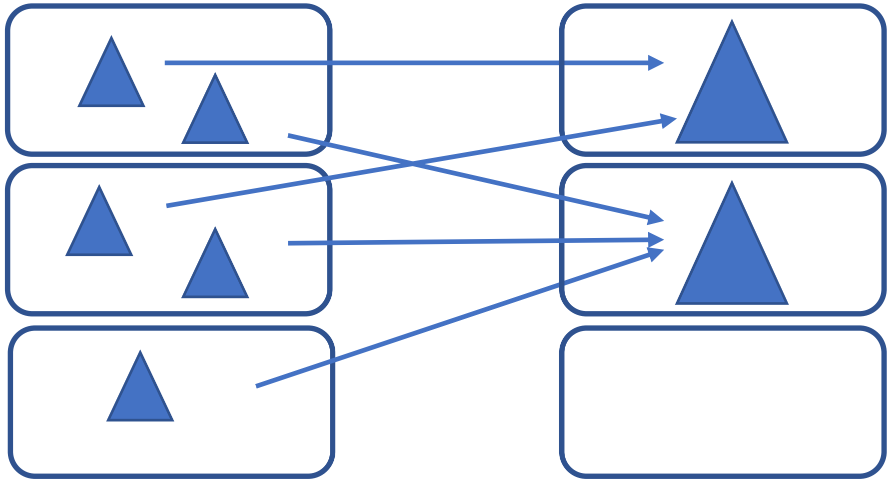


Distributed Processing to compute each new part

RAW to LAKE – Step 3/4 : Repartition Dataset

transform

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

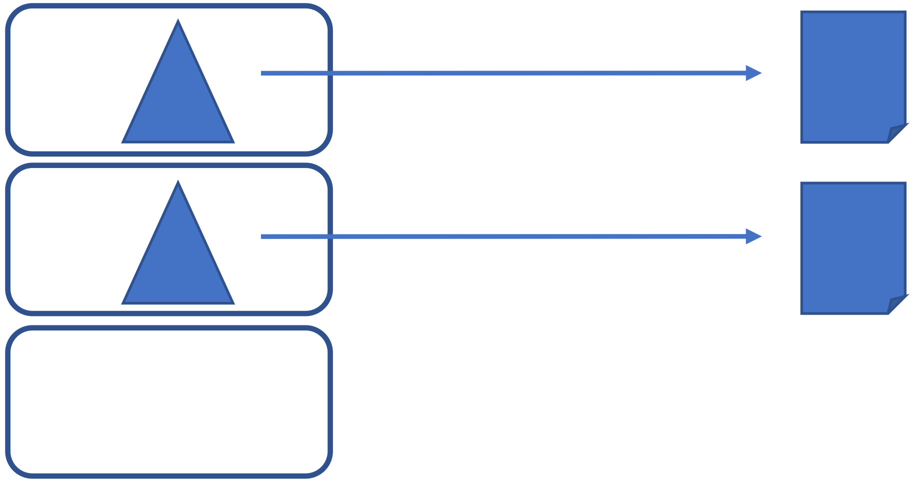


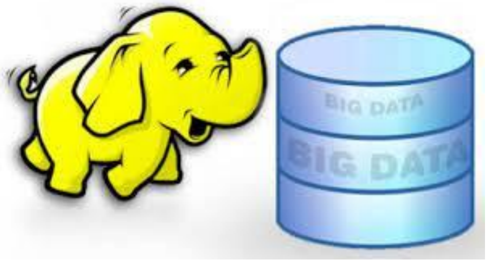
Network Shuffle to distribute / group / sort data

RAW to LAKE – Step 4/4 : Write Dataset

write

```
ds3.write
  .fomat(« parquet »)
  .save(« hdfs://lake/trig/domain/table »)
```



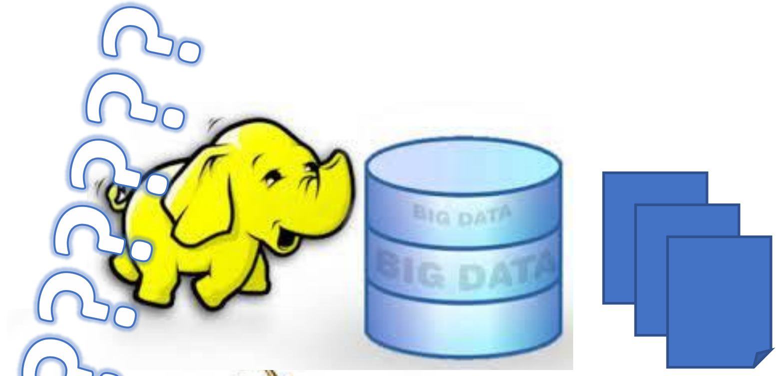


Distributed Write Dataset to Storage

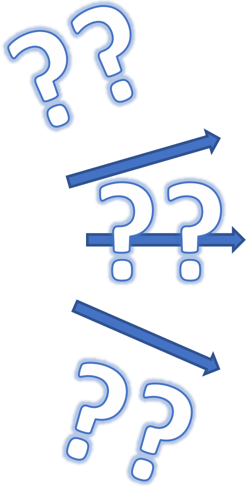
How it works ?

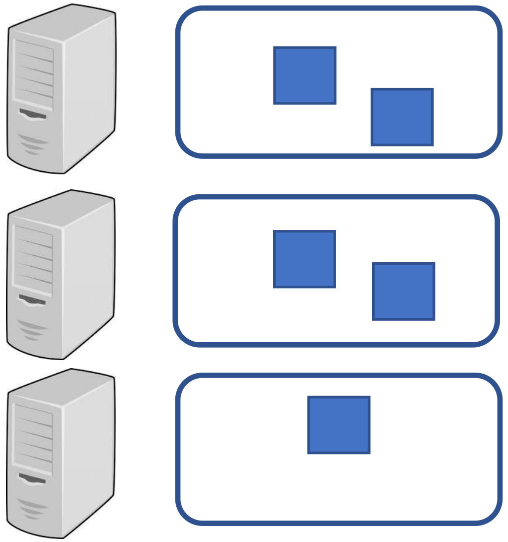
Zooming more ..

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



Input = Files
from Distributed Storage



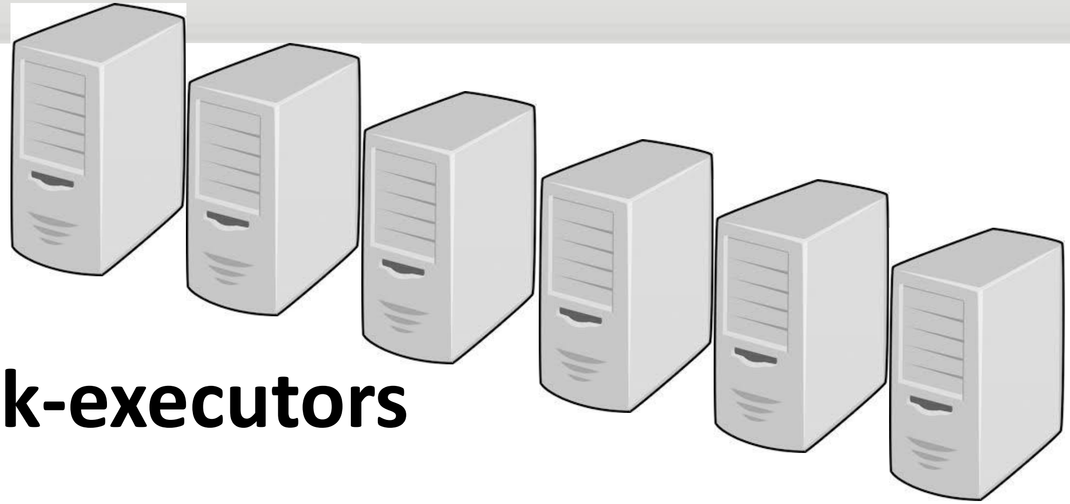


Result = Distributed Parts in-memory

?? **How to Assign** N x Files – P x blocks ⇔ to Q x Executors **??**
+ Retry on Error ?? + Communicate more ??

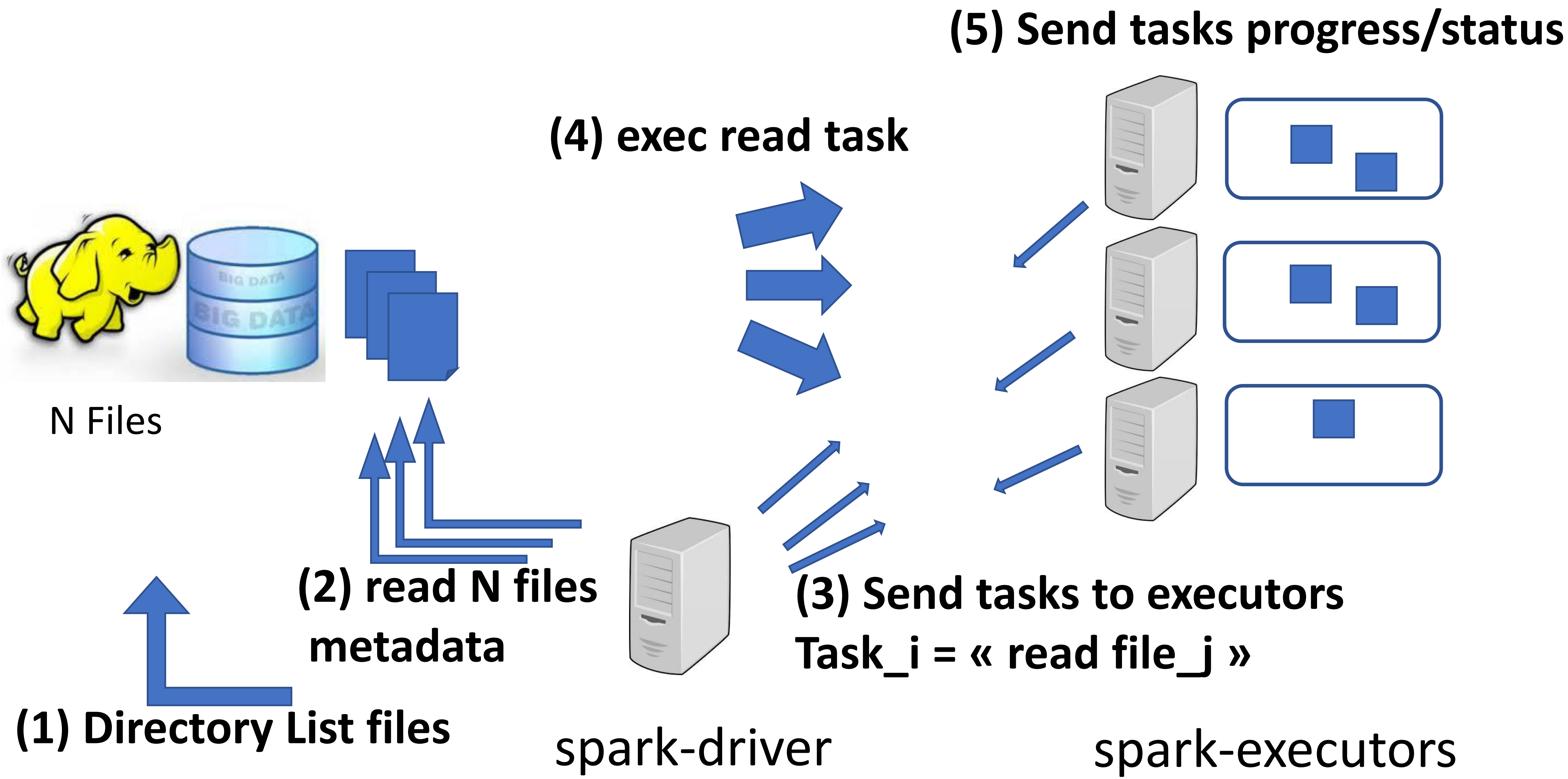
Analogy : How to play music ?
(N musicians without 1 Conductor != 1 Orchestra)

spark-driver

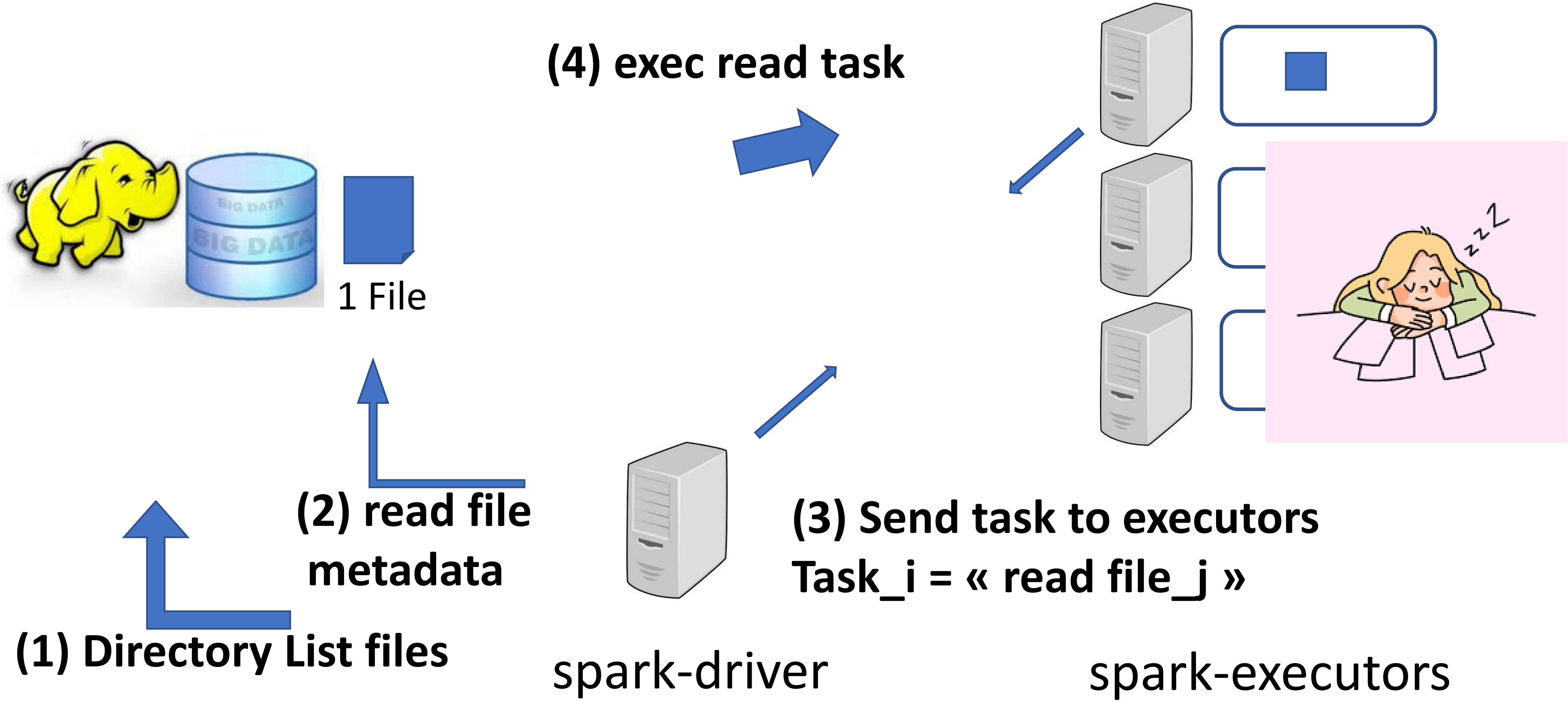


spark-executors

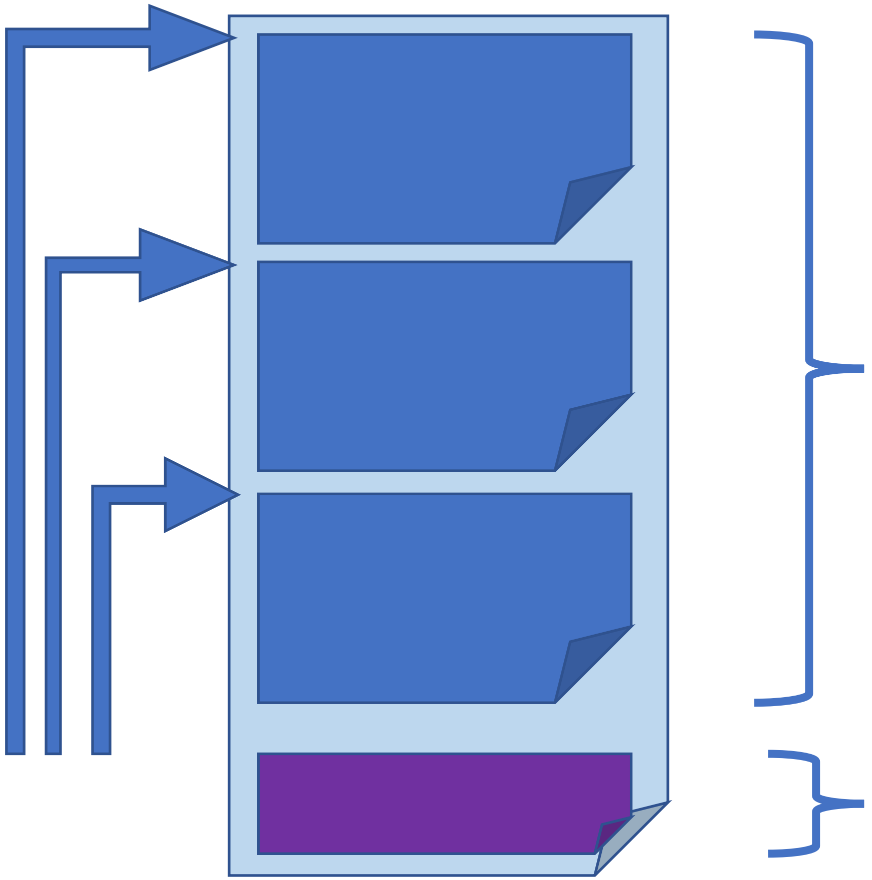
Read N Files – assign Tasks to Executors



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

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

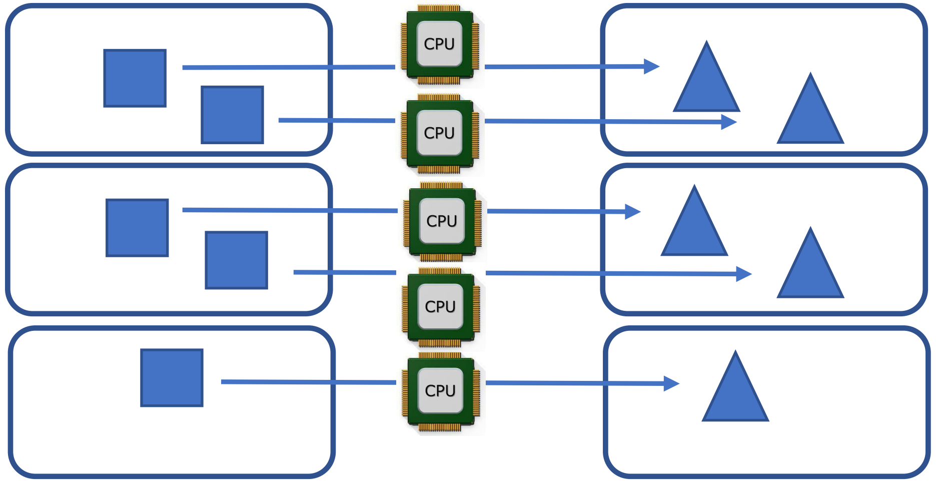
File metadata
= schema + N blocks infos
(offset + stats)

1 splittable file

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

transform

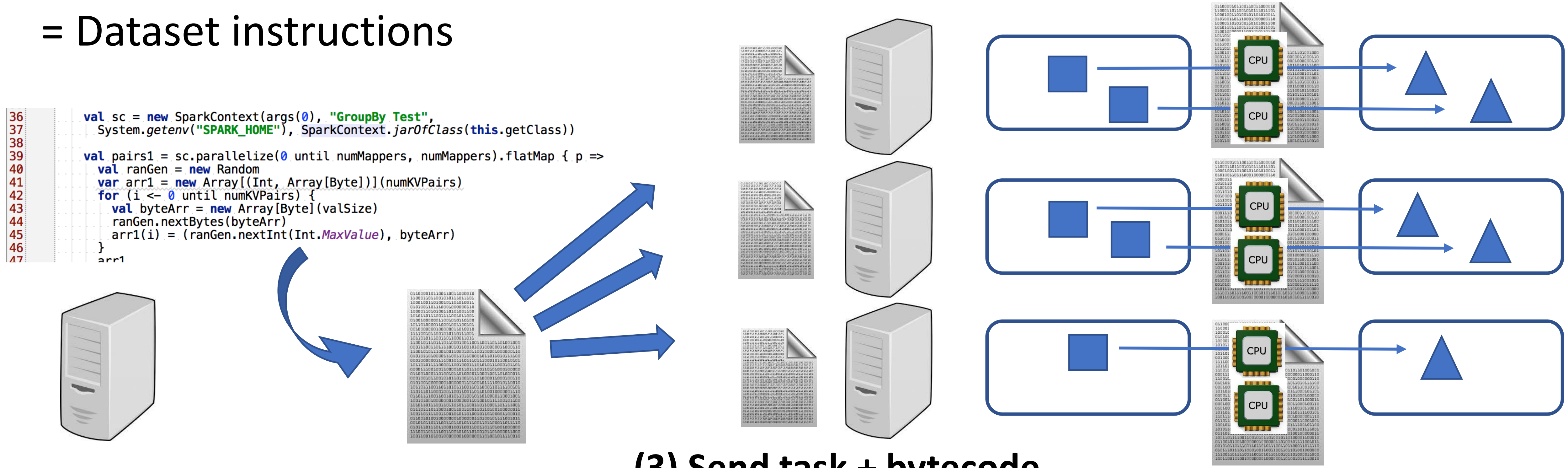
Dataset<Row> ds2 = ds.map(row -> transformData(row))



Distributed Processing to compute each new part

WholeStageCodeGen

Program
= Dataset instructions



(3) Send task + bytecode to spark-executors

(4) Execute tasks

(1) Generate java code

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

(2) Compile Bytecode

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()

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

=

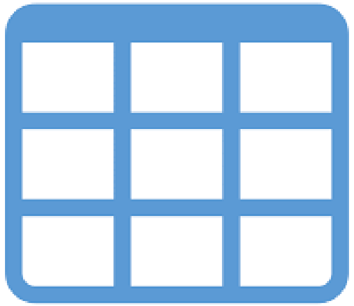
```
Dataset<Row> ds = …

// 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();
```

Converting Tabular SQL Row to Java Beans

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

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

Dataset<Row> df = …

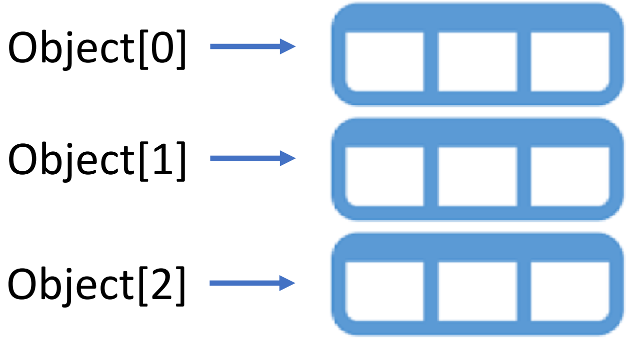


encoder = Encoders.bean(MyBean.class)

df.as(encoder)

ds.**toDF**()

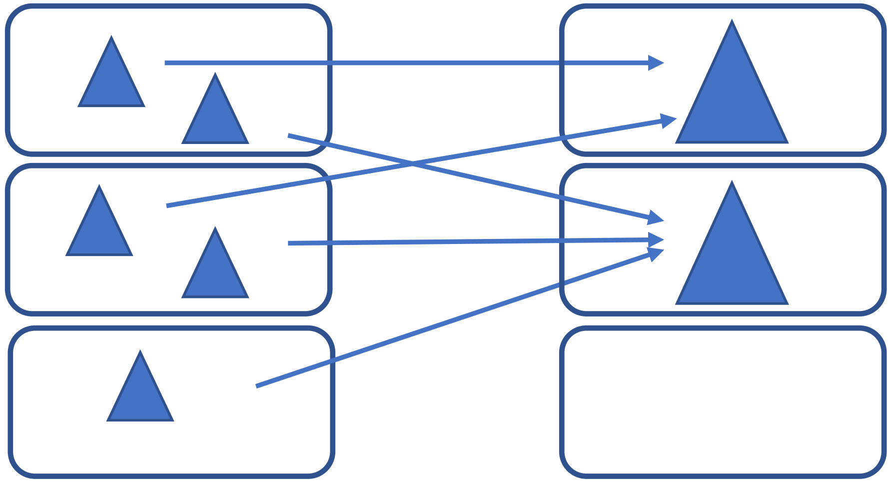
Dataset<MyBean> ds = …



RAW to LAKE – Step 3/4 : Repartition Dataset

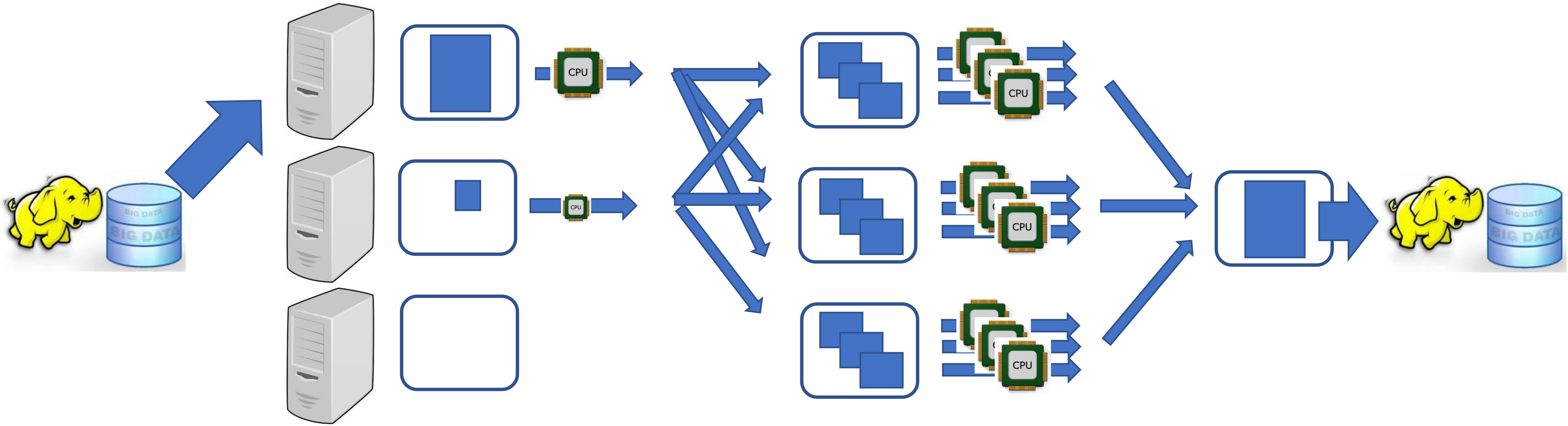
transform

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



Network Shuffle to distribute / group / sort data

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



INNEFFICIENT !!!
Undistributed
/ Badly Skewed Data

repartition(N)

EFFICIENT

repartition(1)

Avoid too many small files

Example transformation … in SQL

Typical RAW to LAKE as Spark SQL

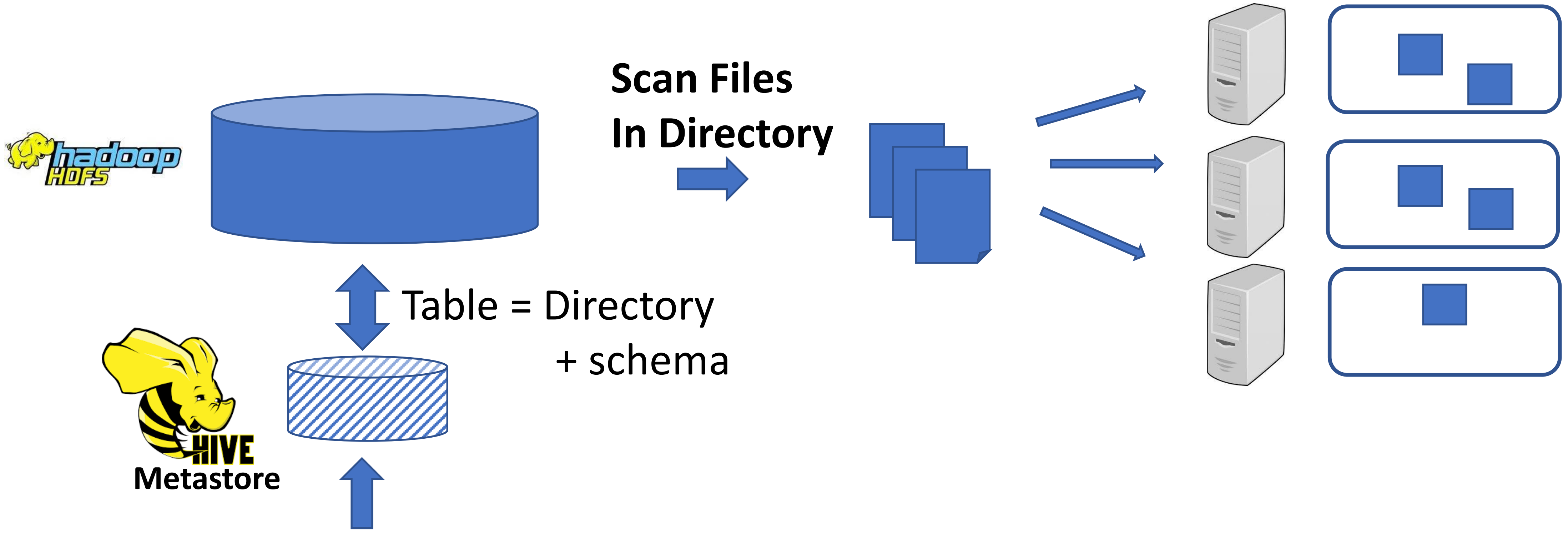
Reminder

write	INSERT OVERWRITE lake_team_domain.table
transform	SELECT /* +REPARTITION(col1, 2) */ col1, col2, udf_func1(col3, col4) as col3, udf_func2(col4, col5) as col4, ..
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

read

SELECT ..
FROM
    raw_team_domain.table

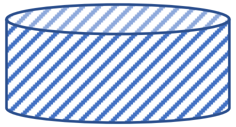


Input = SELECT .. from SQL Table

(Hive) MetaStore

Store ONLY metadatas (DDL + partitions)

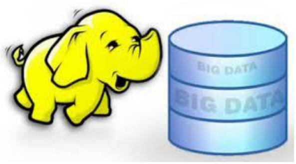
Mapping SQL – Dirs+Files



DDL:

CREATE EXTERNAL TABLE **students** (
  socialSecuId: Int,
  firstName string, lastName string,
  birth: Date, …
) PARTITIONED BY (promo: Int)
STORED AS **parquet**
LOCATION ‘hdfs://lake/students’

Location Dir + Partitions



« / » = root hdfs://host/

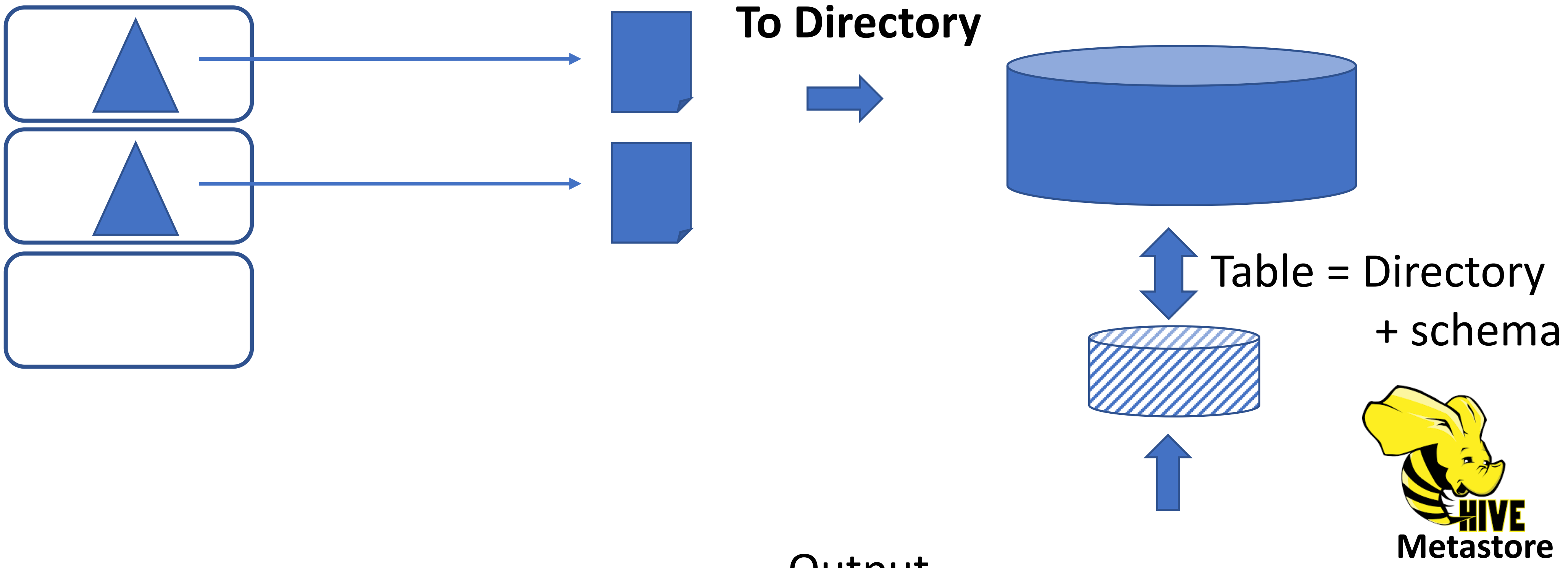
- « /lake » (dir)
  - « /students » (table storage dir)
    - « /promo=2021 » (partition dir)
    - « /promo=2022 » (partition dir)
      - « file1.parquet» (data file)
      - « file2.parquet» (data file)

Dataset -> INSERT SQL Table (-> Files)

write

INSERT OVERWRITE
lake_team_domain.table

Write Files
(append/overwrite)
To Directory



Output

SQL: INSERT INTO/OVERWRITE Table

More Java <-> Sql Interactions

Executing Single SQL from Java

SQL

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(), …

./bin/spark-sql.sh -f sql-script.hql

3.5.4 Overview Programming Guides API Docs Deploying More

Search the docs

Spark SQL Guide

- Getting Started
- Data Sources
- Performance Tuning
- Distributed SQL Engine
  - Running the Thrift JDBC/ODBC server
  - Running the Spark SQL CLI
- PySpark Usage Guide for Pandas with Apache Arrow
- Migration Guide
- SQL Reference
- Error Conditions

Spark SQL Command Line Options

You may run `./bin/spark-sql --help` for a complete list of all available options.

```
CLI options:
 -d,--define <key=value>          Variable substitution to apply to Hive
                                  commands. e.g. -d A=B or --define A=B
    --database <databasename>     Specify the database to use
 -e <quoted-query-string>         SQL from command line
 -f <filename>                    SQL from files
 -H,--help                        Print help information
    --hiveconf <property=value>   Use value for given property
    --hivevar <key=value>         Variable substitution to apply to Hive
                                  commands. e.g. --hivevar A=B

 -i <filename>                    Initialization SQL file
 -S,--silent                      Silent mode in interactive shell
 -v,--verbose                     Verbose mode (echo executed SQL to the
                                  console)
```

The hiverc File

When invoked without the `-i`, the Spark SQL CLI will attempt to load `$HIVE_HOME/bin/.hiverc` and `$HOME/.hiverc` as initialization files.

HQL = Hive Query Langage
... extension of SQL
";"-separated sequence of statements
(DML Queries + DDL)

```
String multiStatementSql =
        "create table xx as select.. from Table1 ...  \n "
      + "; " // <==== separator
      + "select ... from Table2\"
      + ";"  // <==== separator
      + "drop table xx";

spark.sql(  multiStatementSql )   //  <=== FAIL !!!
```

spark-sql.sh Equivalent to custom "spark" code + SQL escape parsing

```
String hqlFileContent = .....

// split by ";"
// escape ";"  in sql  line comment "-- .. ; ... ",
// escape in sql multi-lines comment  "/* ...  ; ...  */"
// escape in sql chars "\;" but not "\\"
// (not in spark API !! write yourself )
List<String>  sqlList =  splitHql(hqlFileContent);

for(String singleSql : sqlList) {
    spark.sql(singleSql).show();
}
```

Java DataSet as SQL View

Dataset[Row] ds = ..

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

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

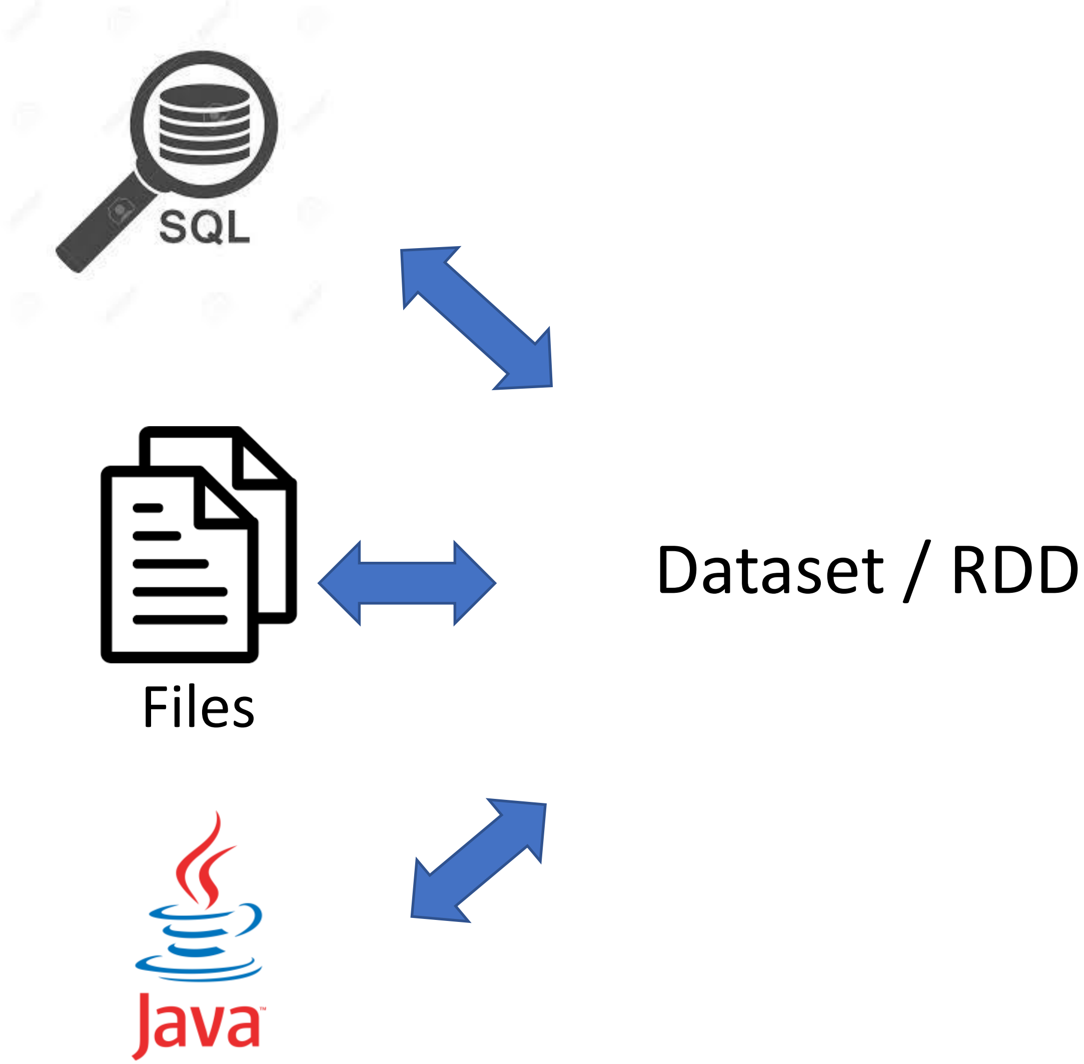
Calling Java from SQL : User-Defined Function

transform

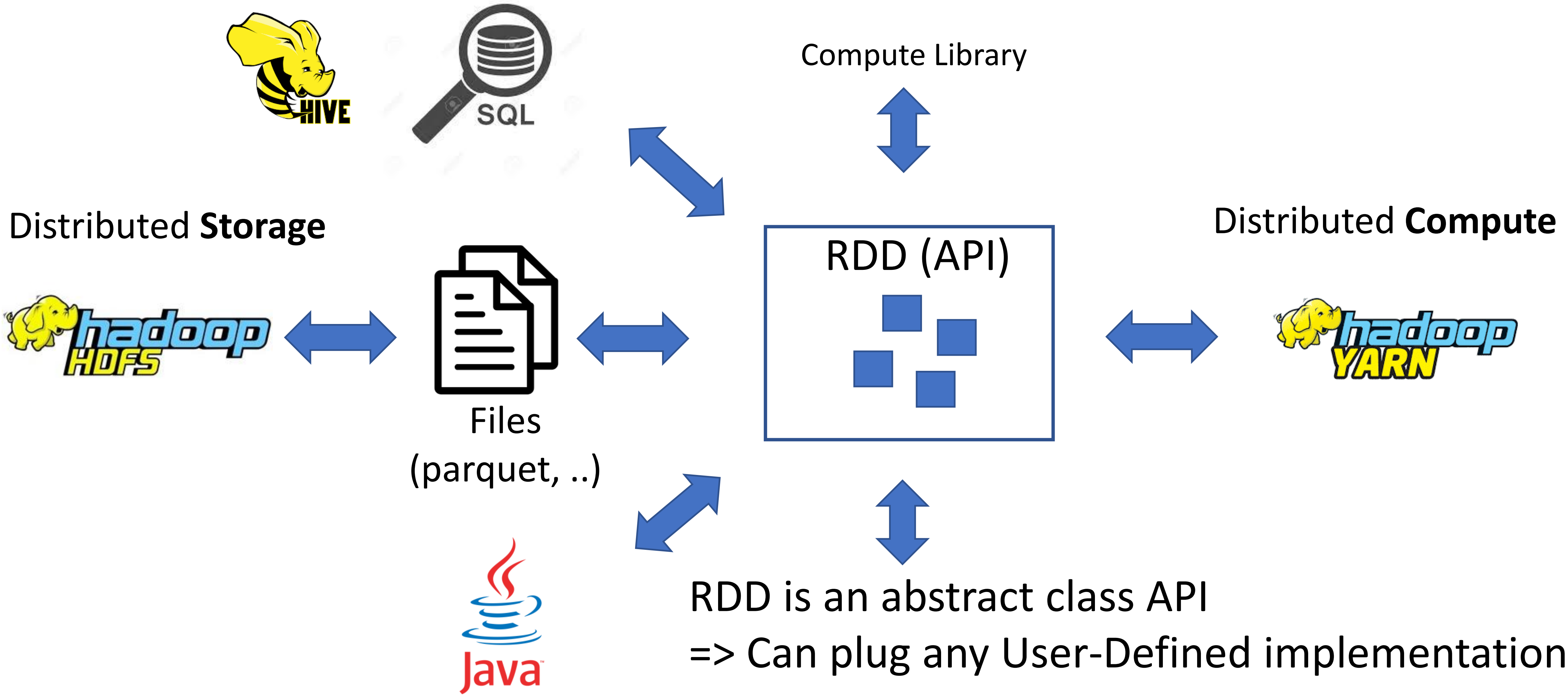
```
SELECT ..
   udf_func1(col3, col4)  as col3,
   udf_func2(col4, col5)  as col4,
```

```
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)



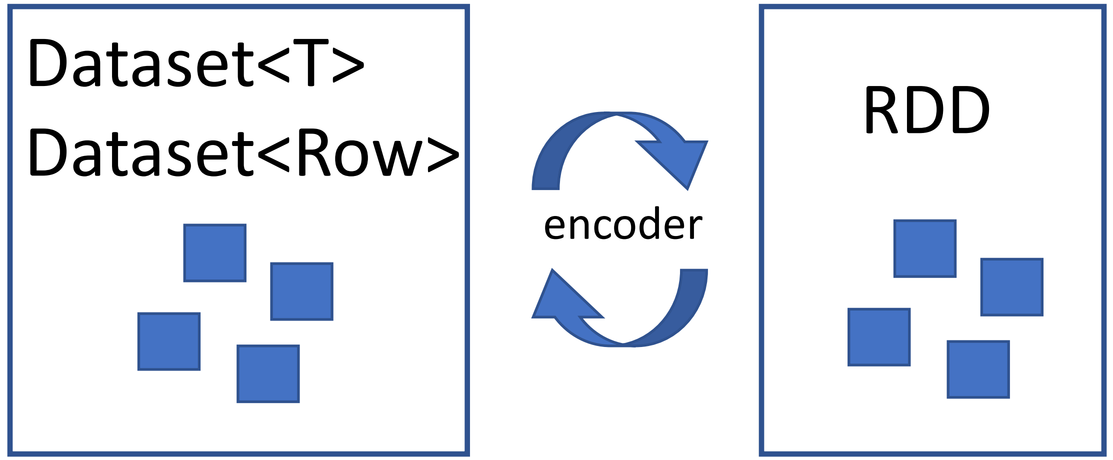
RDD is an abstract class API

=> Can plug any User-Defined implementations

Dataset API .... SQL Extensions

Column API
Expression API
(operator overload)

Lamba /Function
+ runtime **compile to bytecode**
(WholeStageCodeGen)



More Extensions: Hadoop FileSystem API

HDFS implements FileSystem

Distributed Storage API

java.io.File adapter

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

Amazon S3

Azure Data Lake Storage Gen2

More adapters

....

Spark rely on API
=> Can plug any implementations

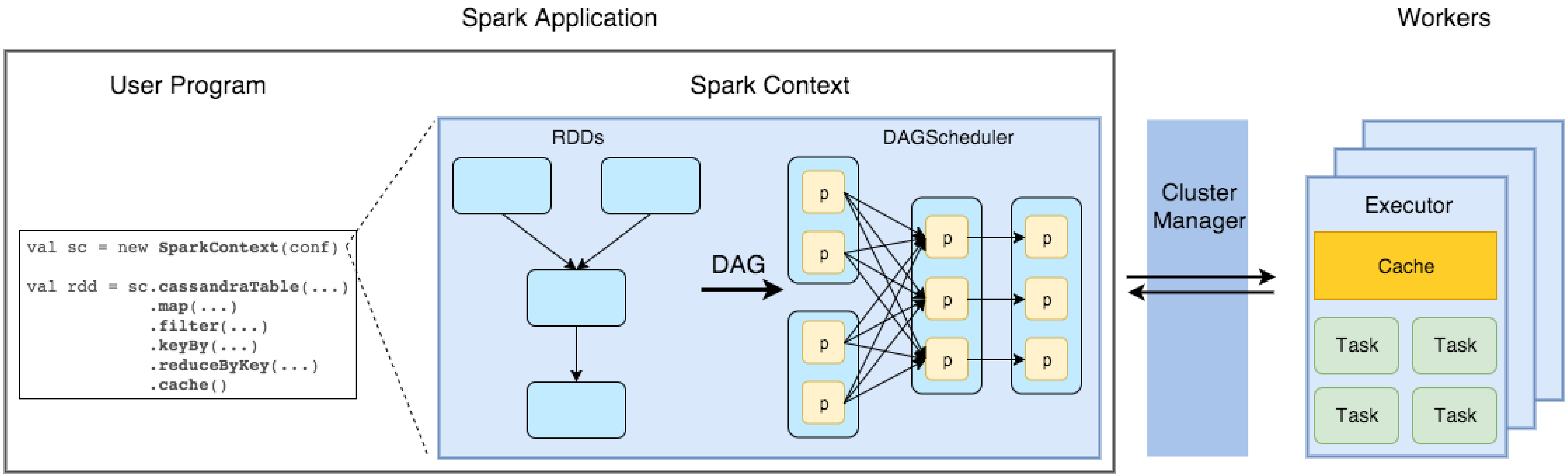
More Extensions: Cluster Scheduler API

Cluster Manager
Scheduler API

TaskScheduler (SPI)

abstract class **TaskScheduler {**
..start,stop,
submitTasks,cancelTasks,
notify Host-Executor-Task changes
}

Standalone Cluster


Spark Application
Workers
User Program
Spark Context
RDDs
DAGScheduler
val sc = new SparkContext(conf)
val rdd = sc.cassandraTable(...)
.map(...)
.filter(...)
.keyBy(...)
.reduceByKey(...)
.cache()
DAG
p
Cluster Manager
Executor
Cache
Task
Task
Task
Task

More « Extensions »
backport API to other Langages (Python, R)

class DataFrame:

… hundred methods …

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

python API

Java API

public class Dataset {

… hundred methods …

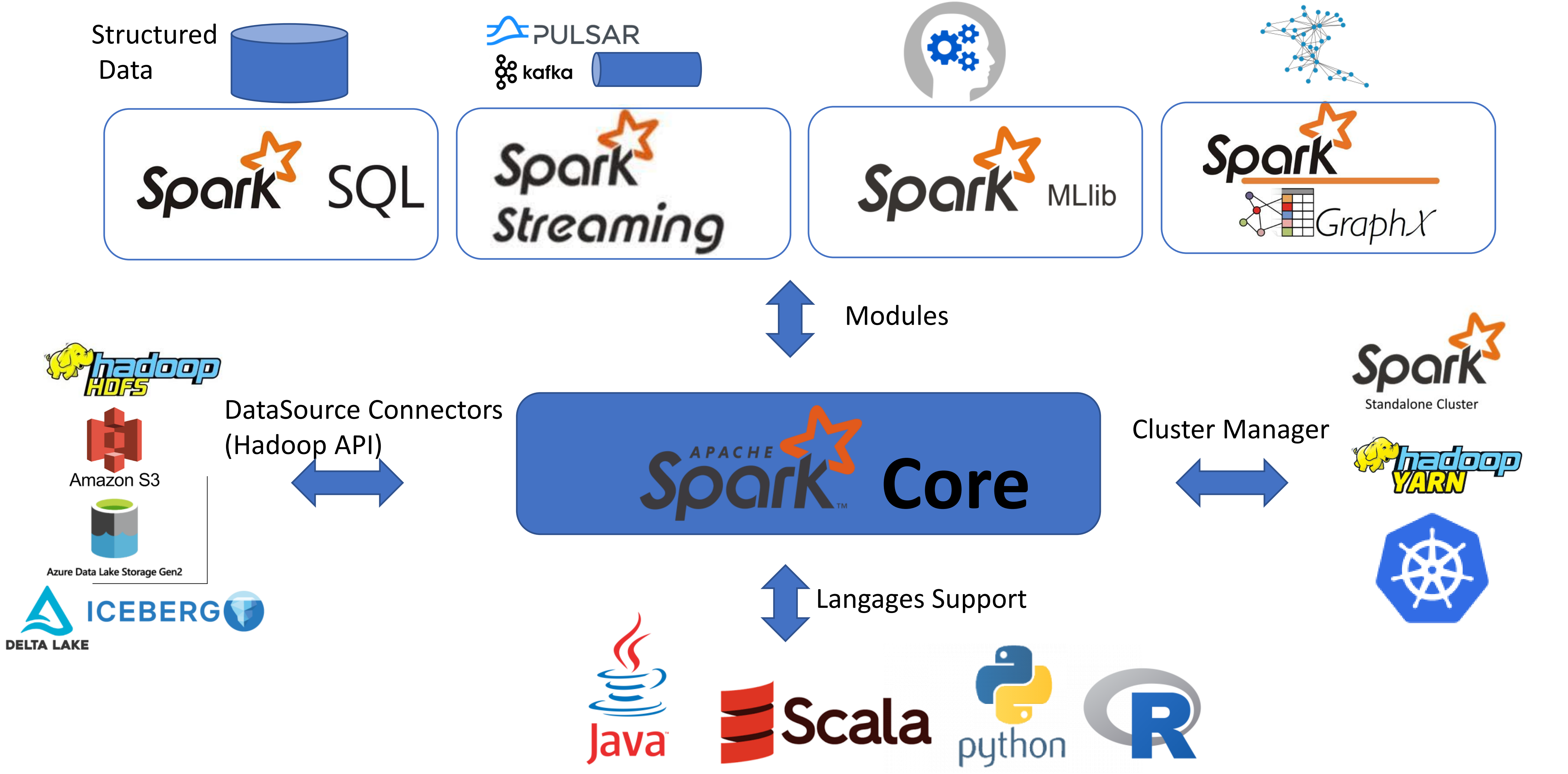
public Y **method123** (X x) {
  … internals sparkContext…
}

py4j

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

Scala **SparkContext**

Spark-Core + …

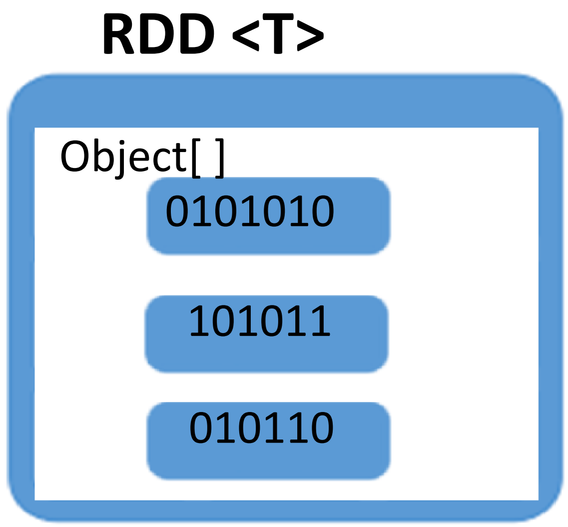


Structured
Data
PULSAR
kafka
Spark SQL
Spark Streaming
Spark MLlib
Spark GraphX
Modules
hadoop HDFS
Amazon S3
Azure Data Lake Storage Gen2
DELTA LAKE
ICEBERG
DataSource Connectors
(Hadoop API)
APACHE Spark Core
Cluster Manager
Spark
Standalone Cluster
hadoop YARN
Langages Support
Java
Scala
python
R

RDD principles

Low-level internal SPI,
should not be used directly

RDD[T] = Resilient Distributed Dataset of "T"



"T" java object instances are just Serializable in binary "01010101"

For end-user, you can not display rows   (can not do "rdd.show()"),
you can not query SQL columns (can not do "rdd.select(..)" or "rdd.sql(..)")

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 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
```

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 Doc (3/3)

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

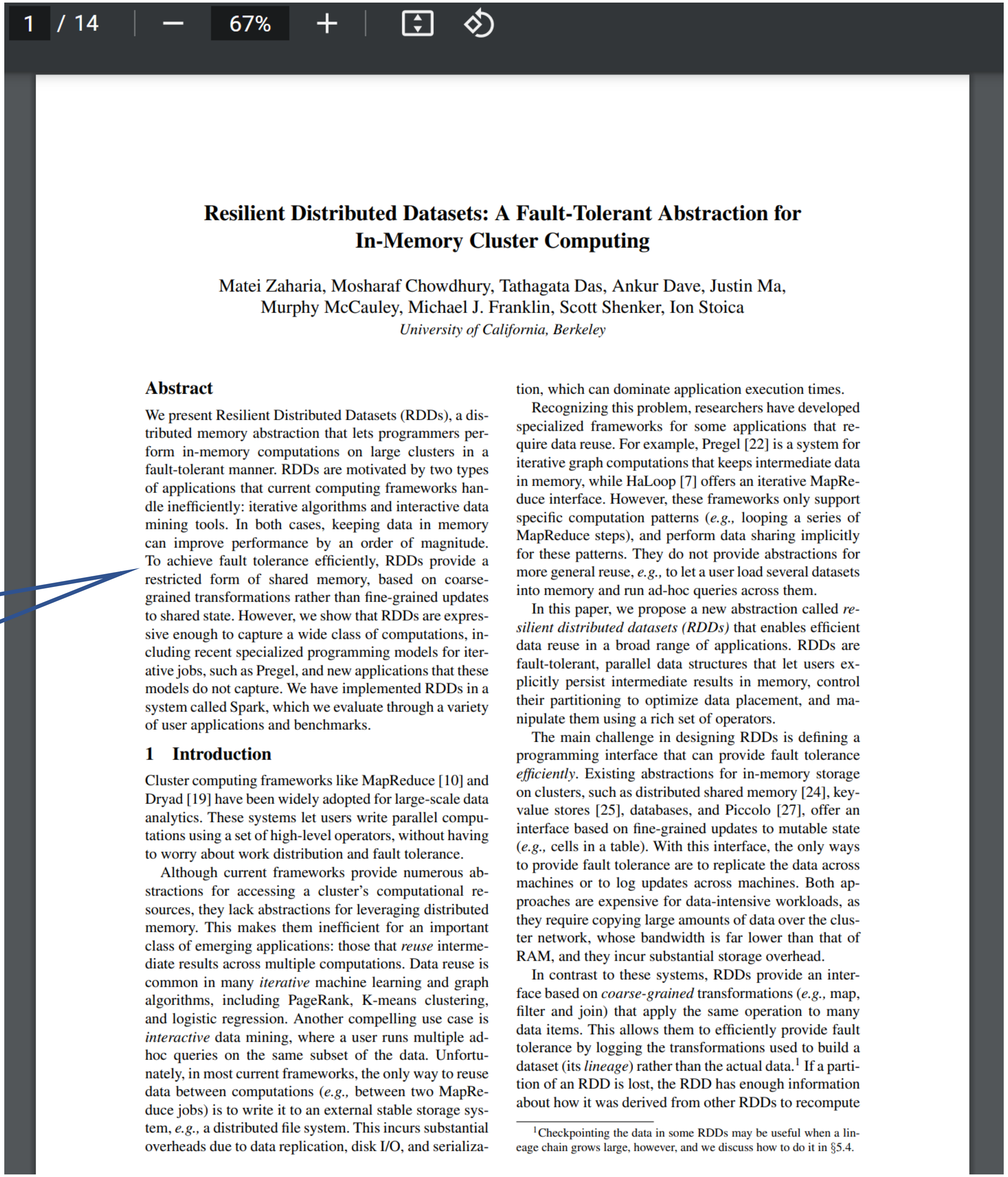
Indeed, users **can implement** custom RDDs
(e.g. for reading data from a new storage system)
by **overriding** these functions.

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



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.

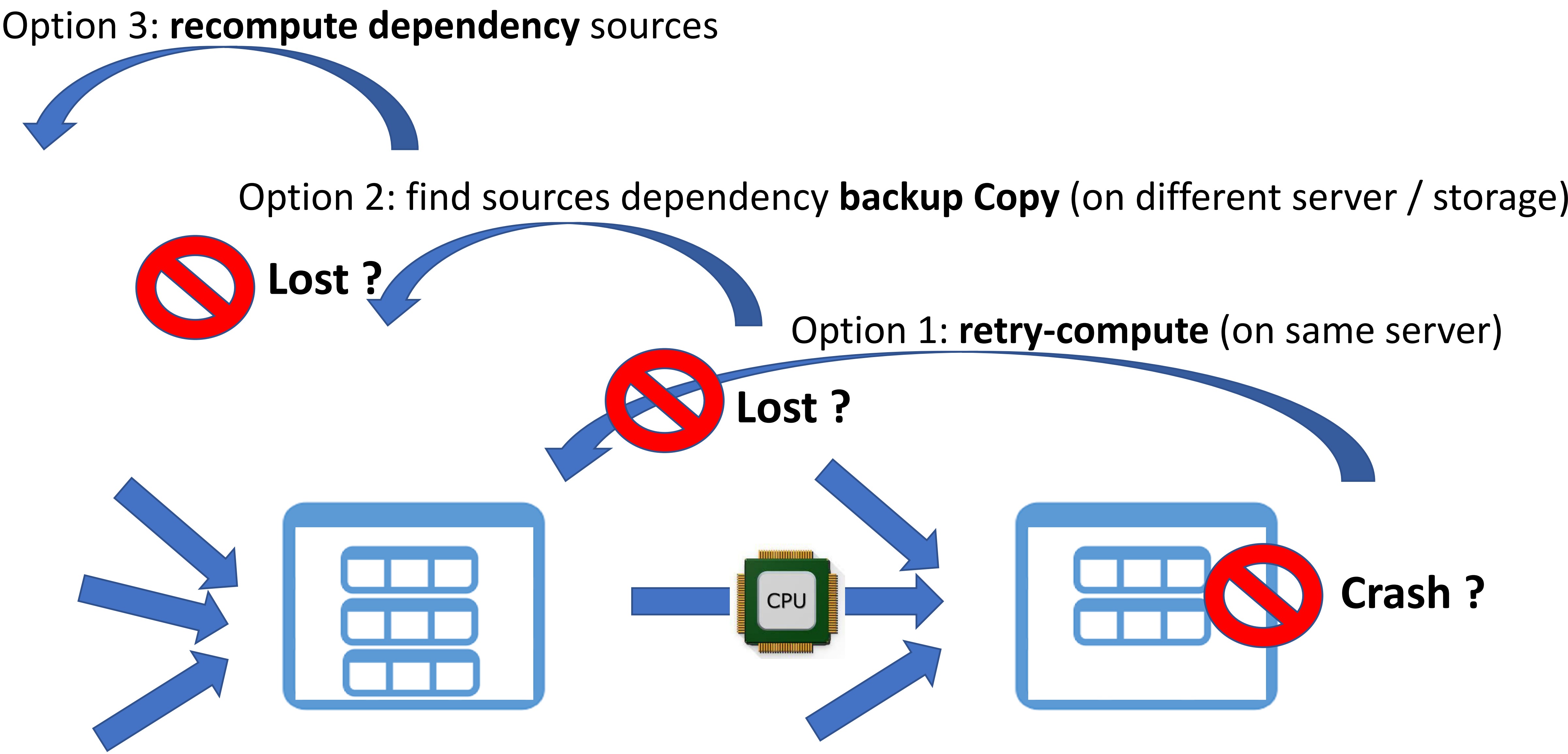
In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning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

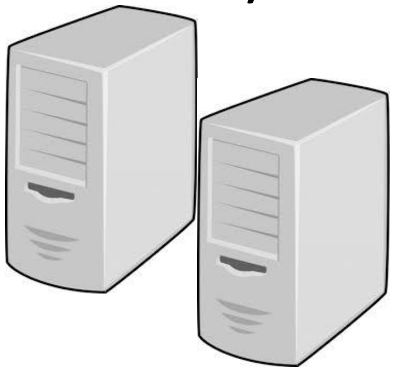
Fault Tolerant - Computation



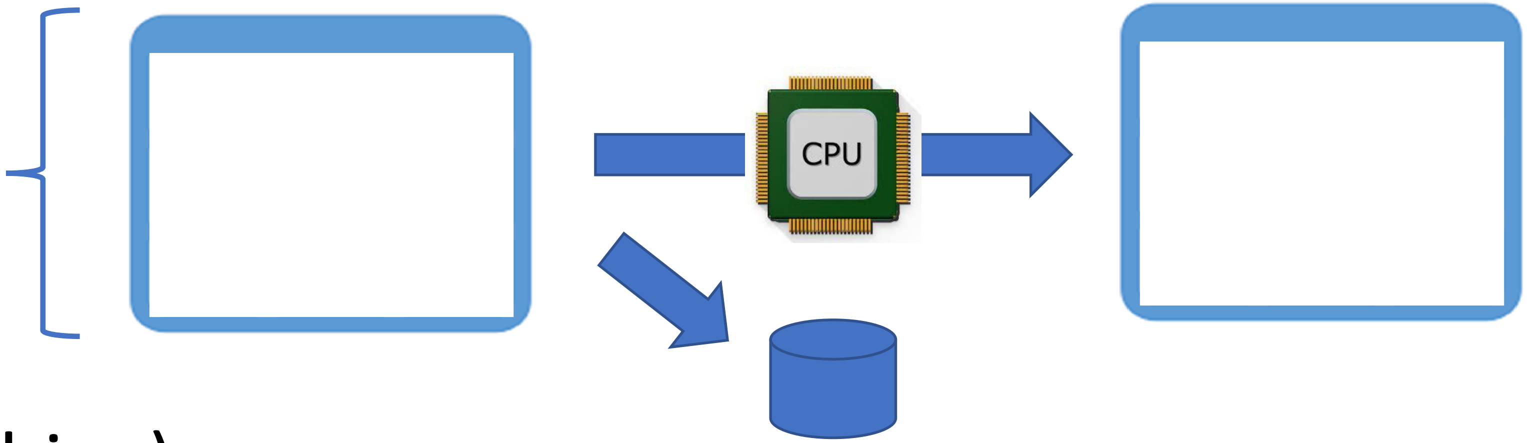
CoarseGrain … Scheduler/Executer

Spark-driver
Implements Fault Tolerance+Distribution
… internally called « CoarseGrainScheduler »

Spark-executor
Implements task main loop
… internally called « CoarseGrainExecutor »

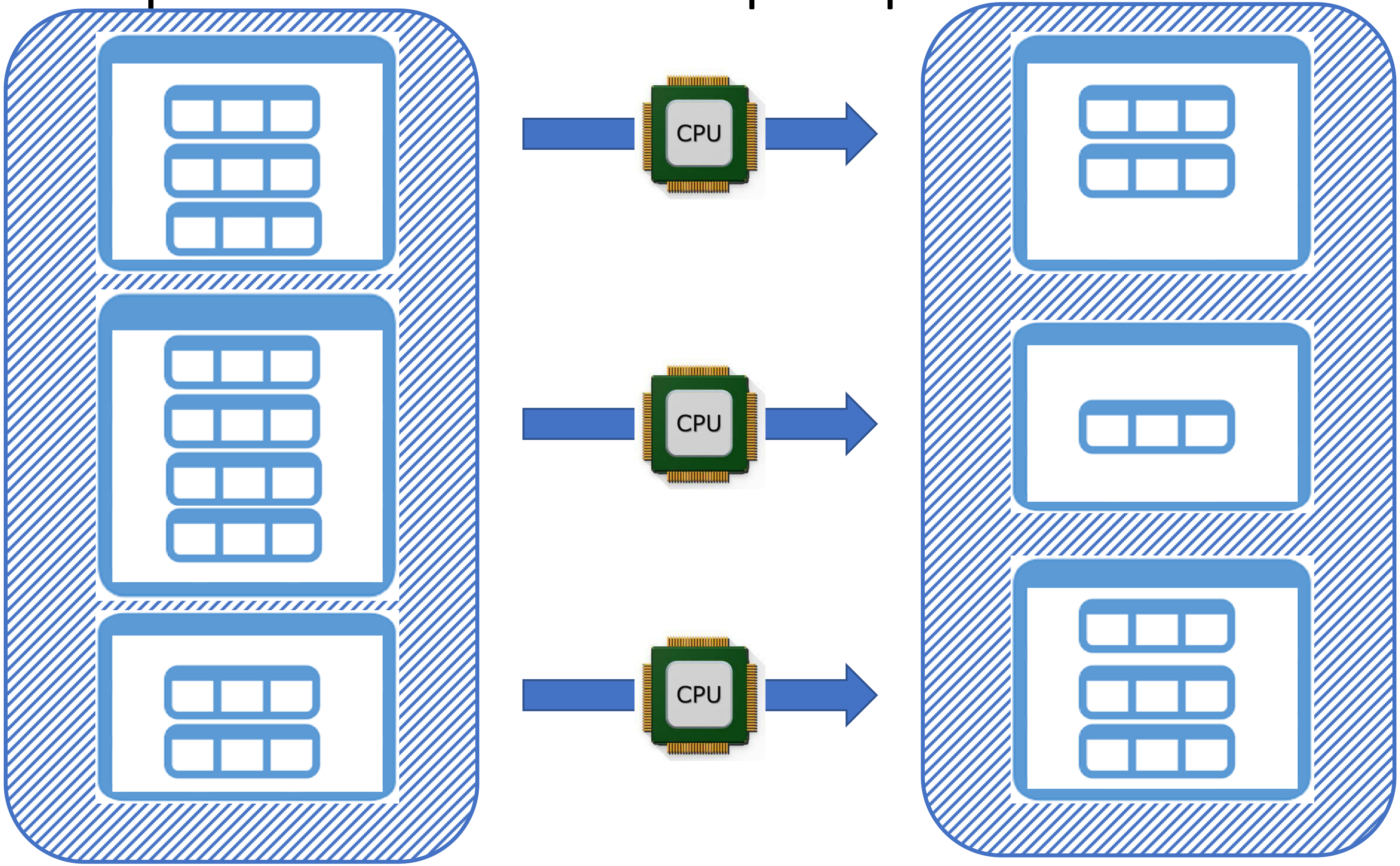


CoarseGrain
partition
= unit of caching/
recomputation
(all elements or nothing)

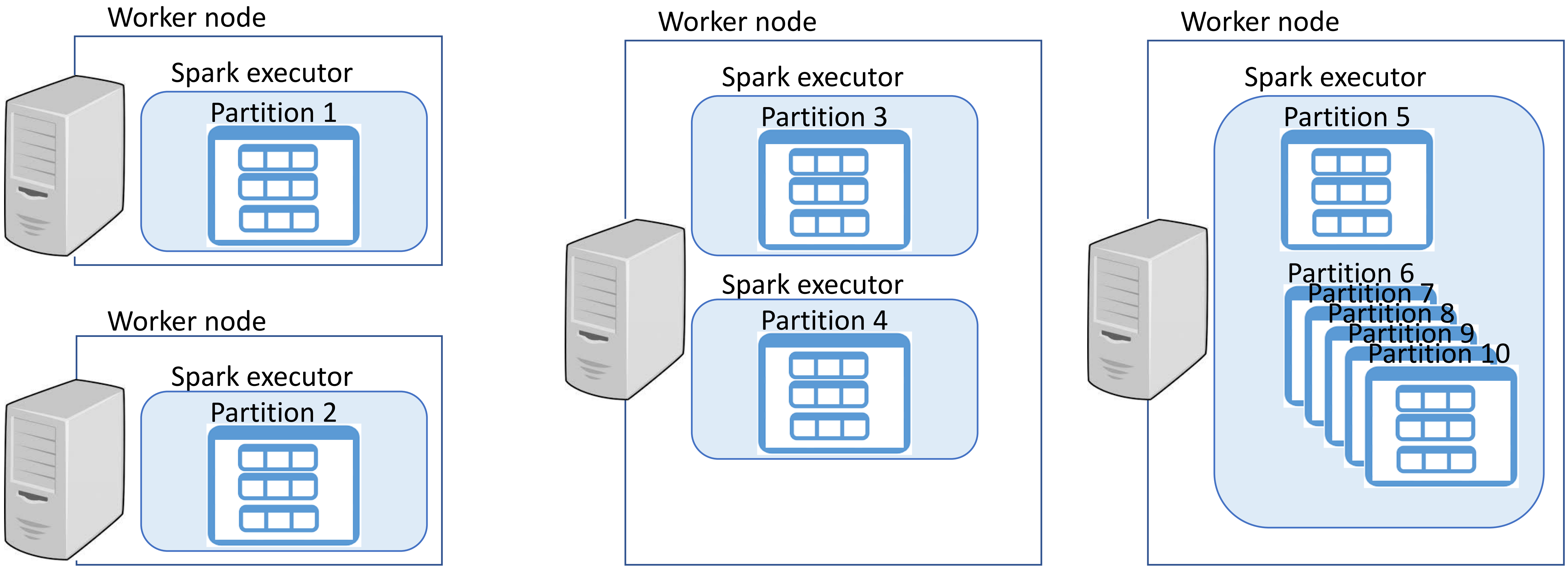


RDD = R.. **Distributed** Dataset

DataSet : Collection of Objects, parallelize 1 CPU per partition



Distribution: Partition < Executor < Node



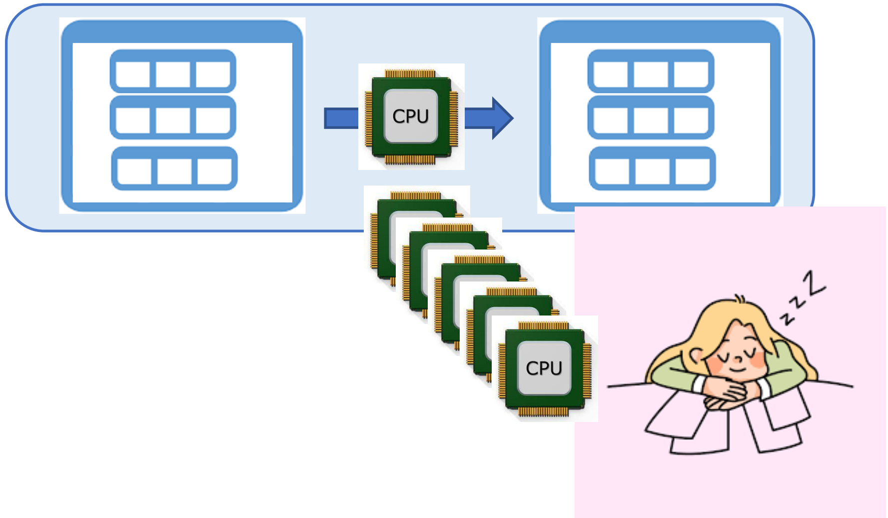
several worker nodes

Several spark-executor processes per nodes

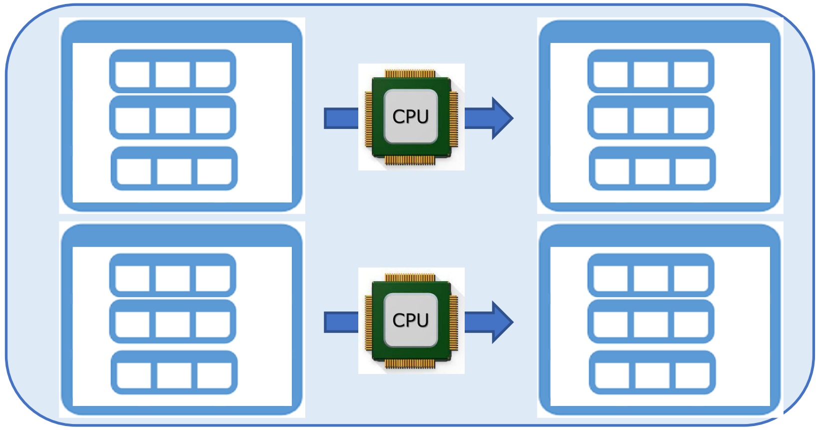
Several partitions Per spark-executor

Optimize parralelism:
Adapt partitions to number of Cores

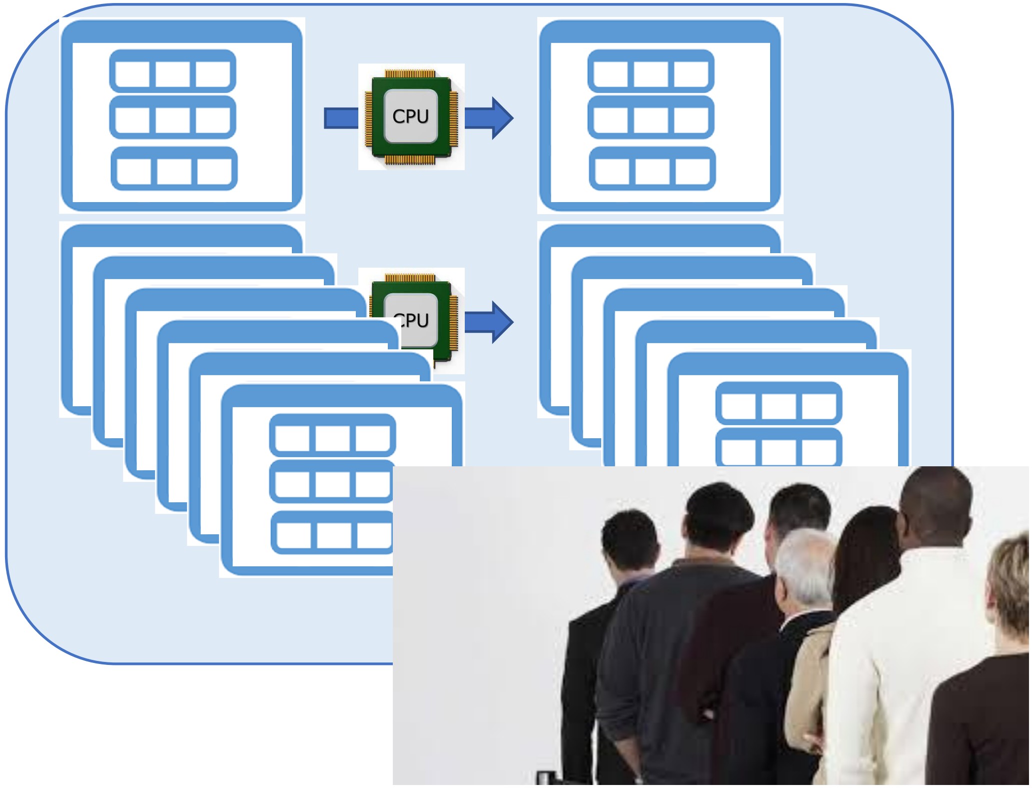
Spark-executor: 1 partition << N cores



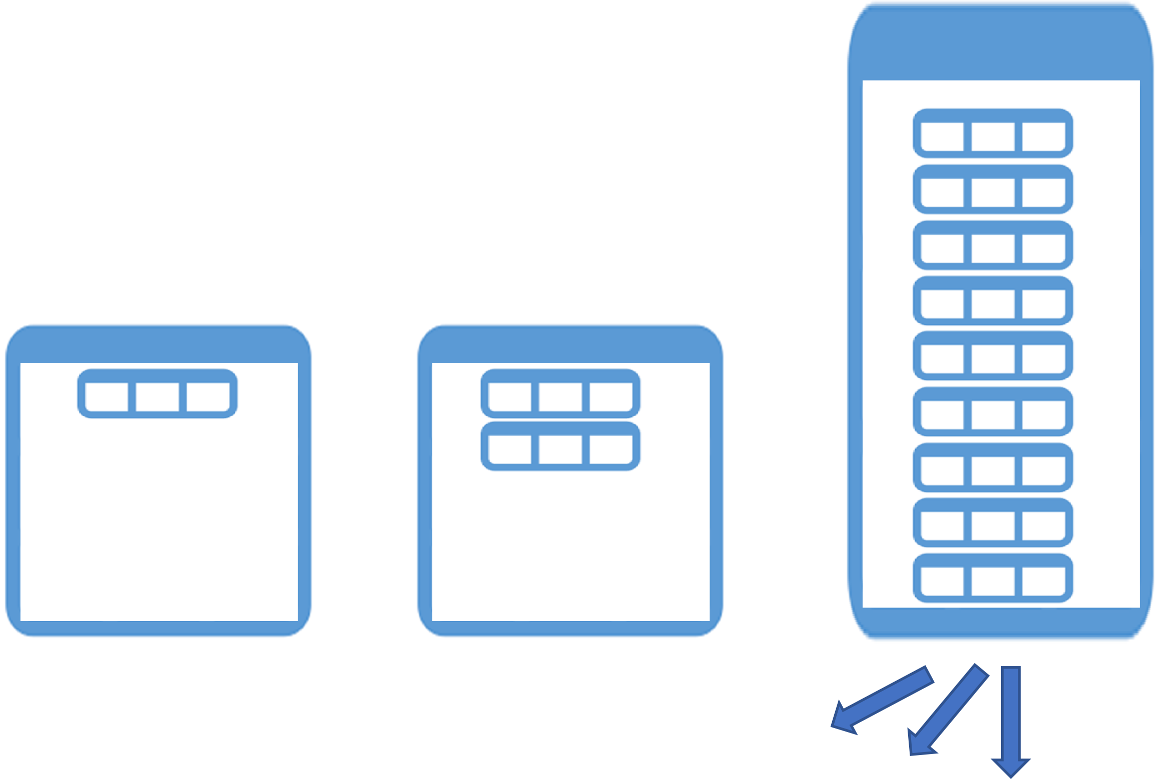
Spark executor: N partitions ~ N cores



Spark executor: N partitions >> few cores

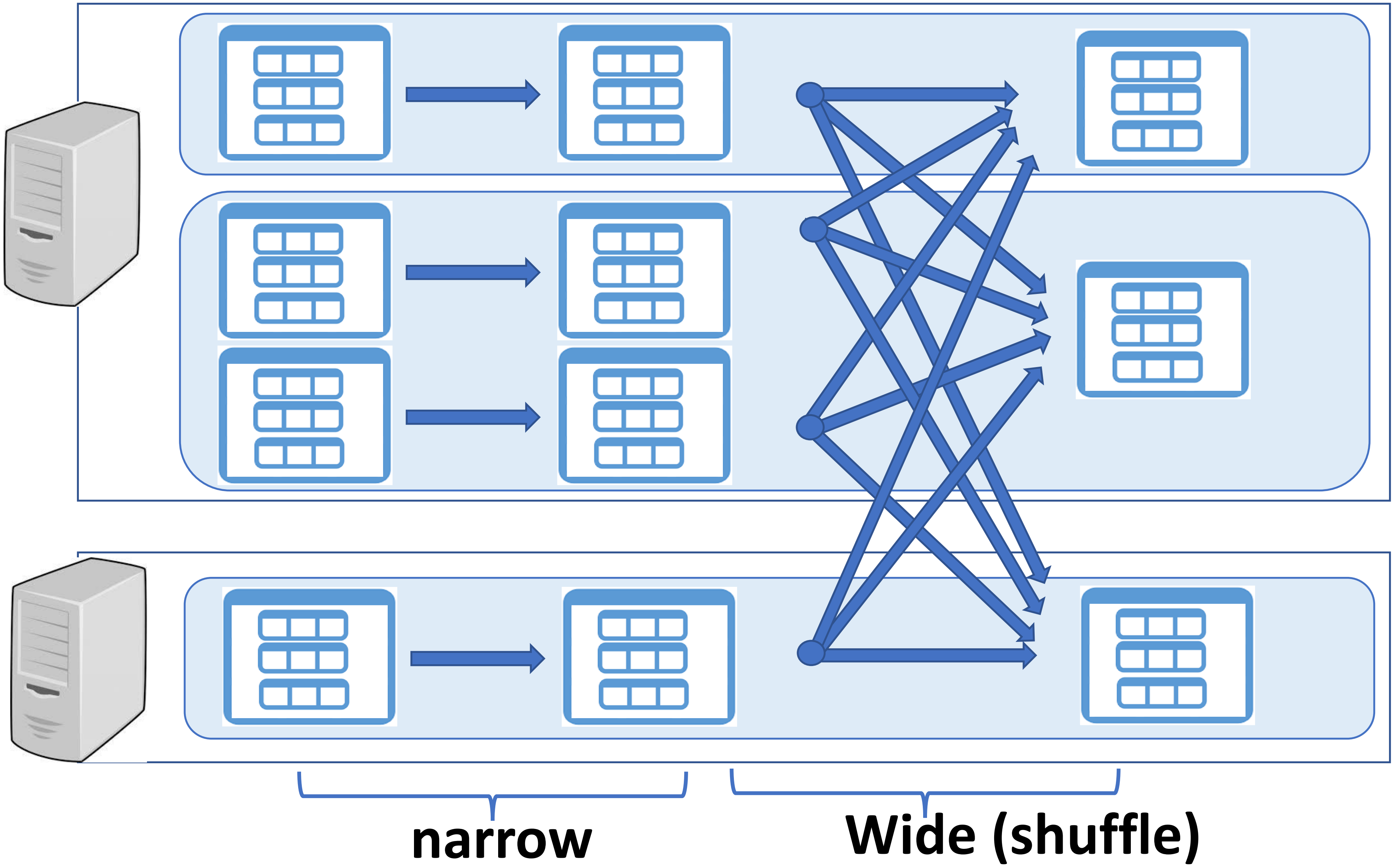


Skewed Data … need Repartitioned equally



Target: move each row[i] to node[j] j = « rowId modulo N »

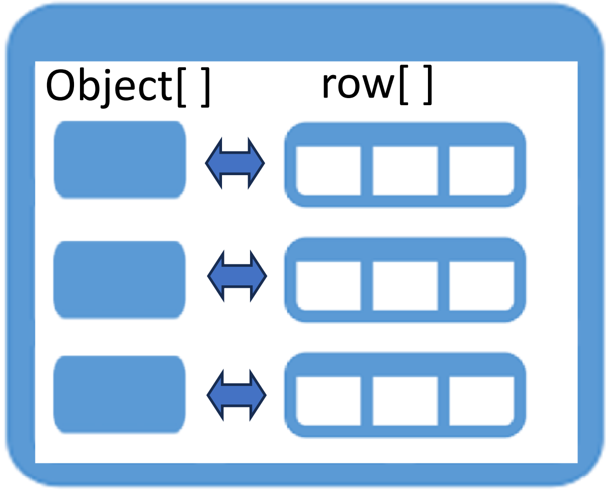
« Narrow » / « Wide » Transformations



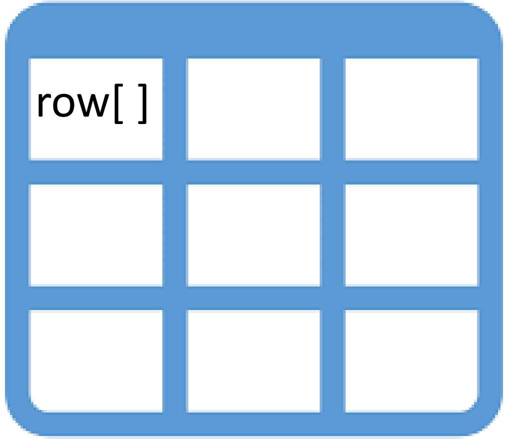
DataSet

DataSet = (~RDD) Set of <Data> + Encoder

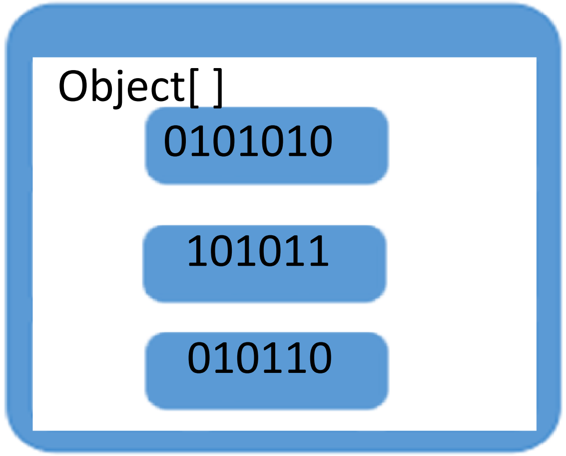
Dataset<UserDefinedClass> (~RDD+Encoder)



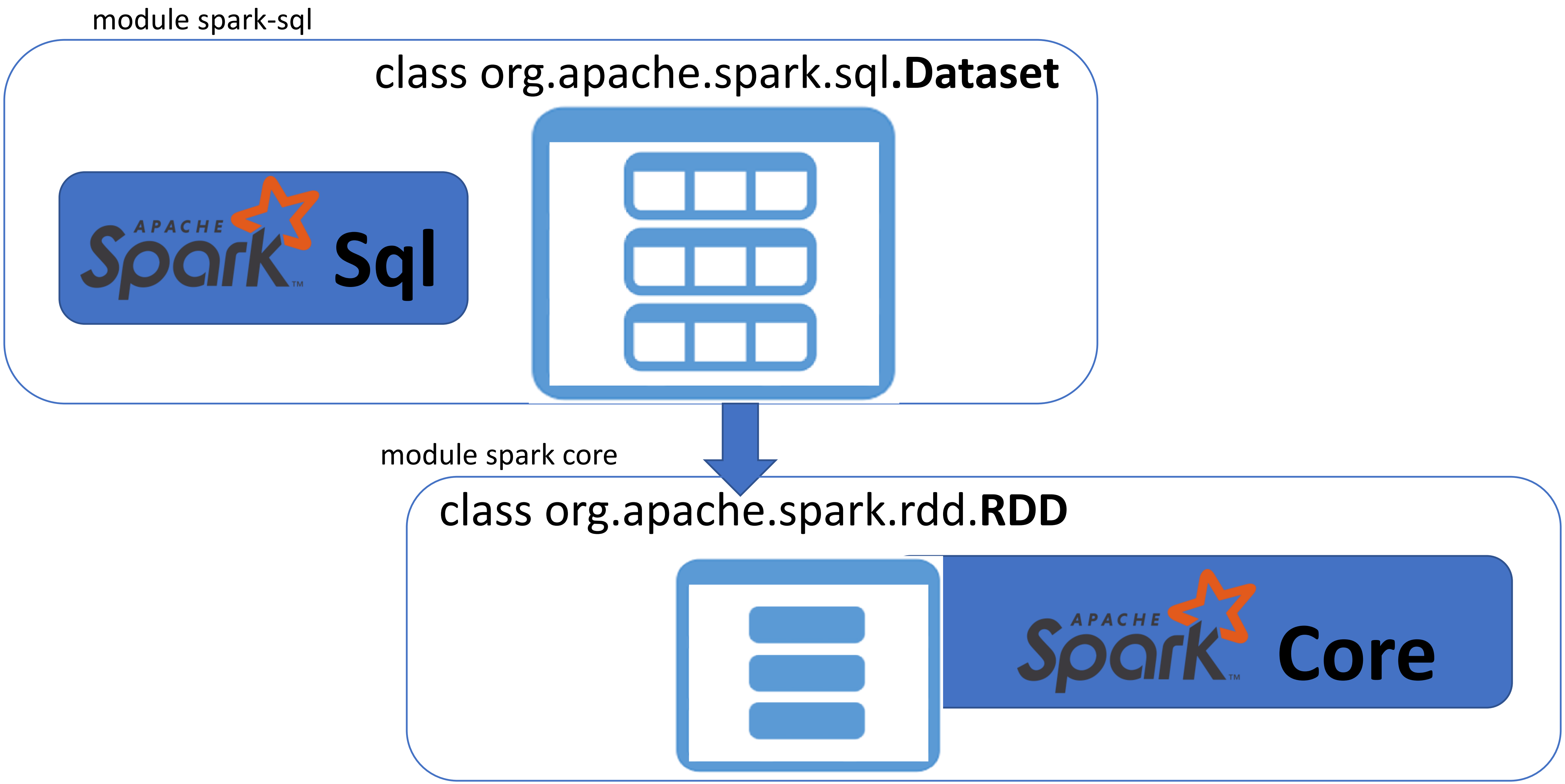
Dataset<Row> = « DataFrame »



RDD



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



module "spark-sql"

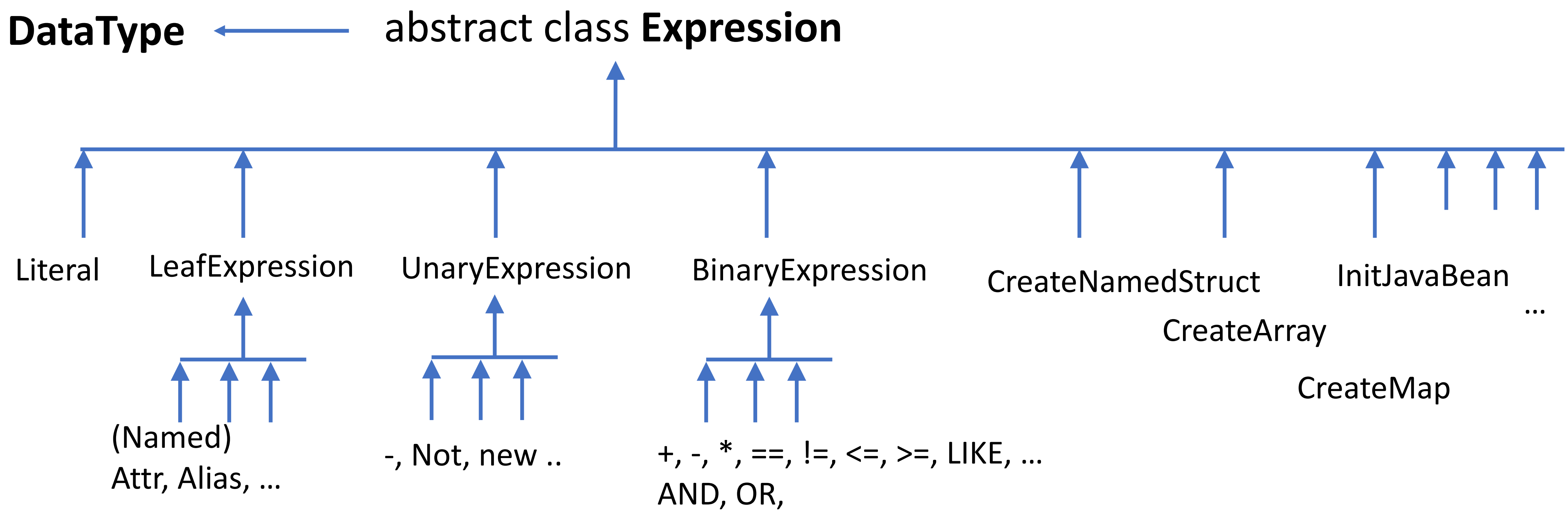
=> SQL Grammar to Code parser

Expression AST (=Abstract Syntaxic Tree)

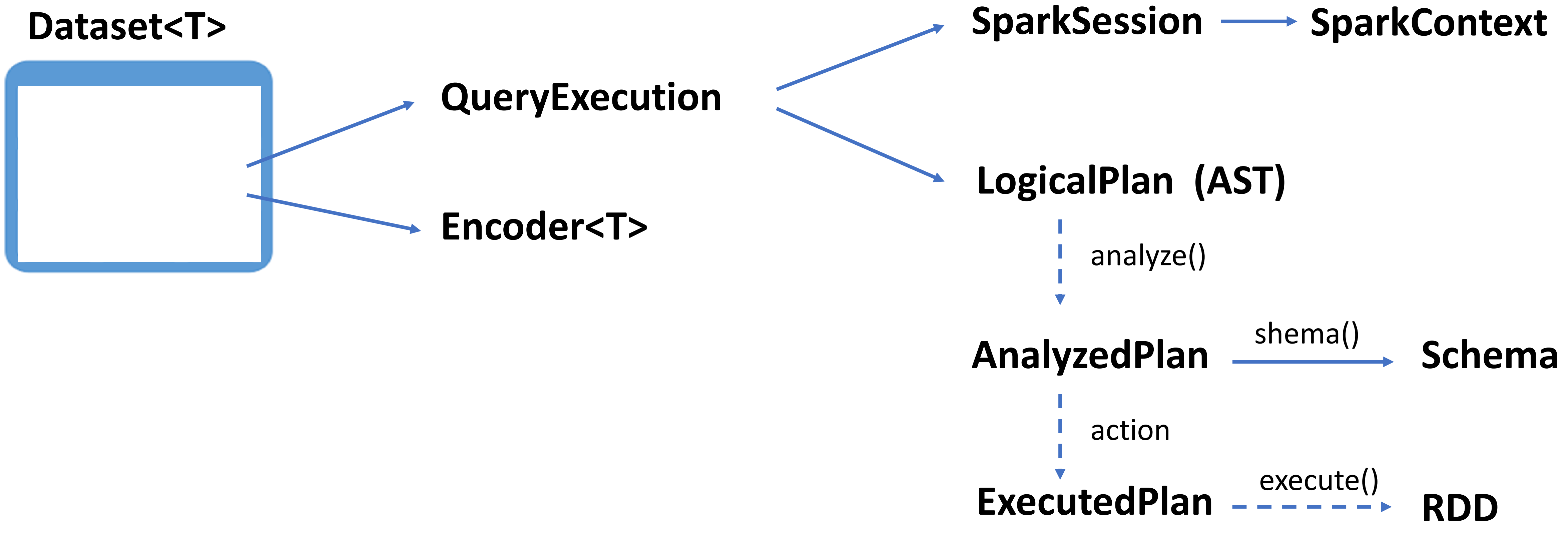
Annexe

Encoder .. Internal Expression with DataType

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

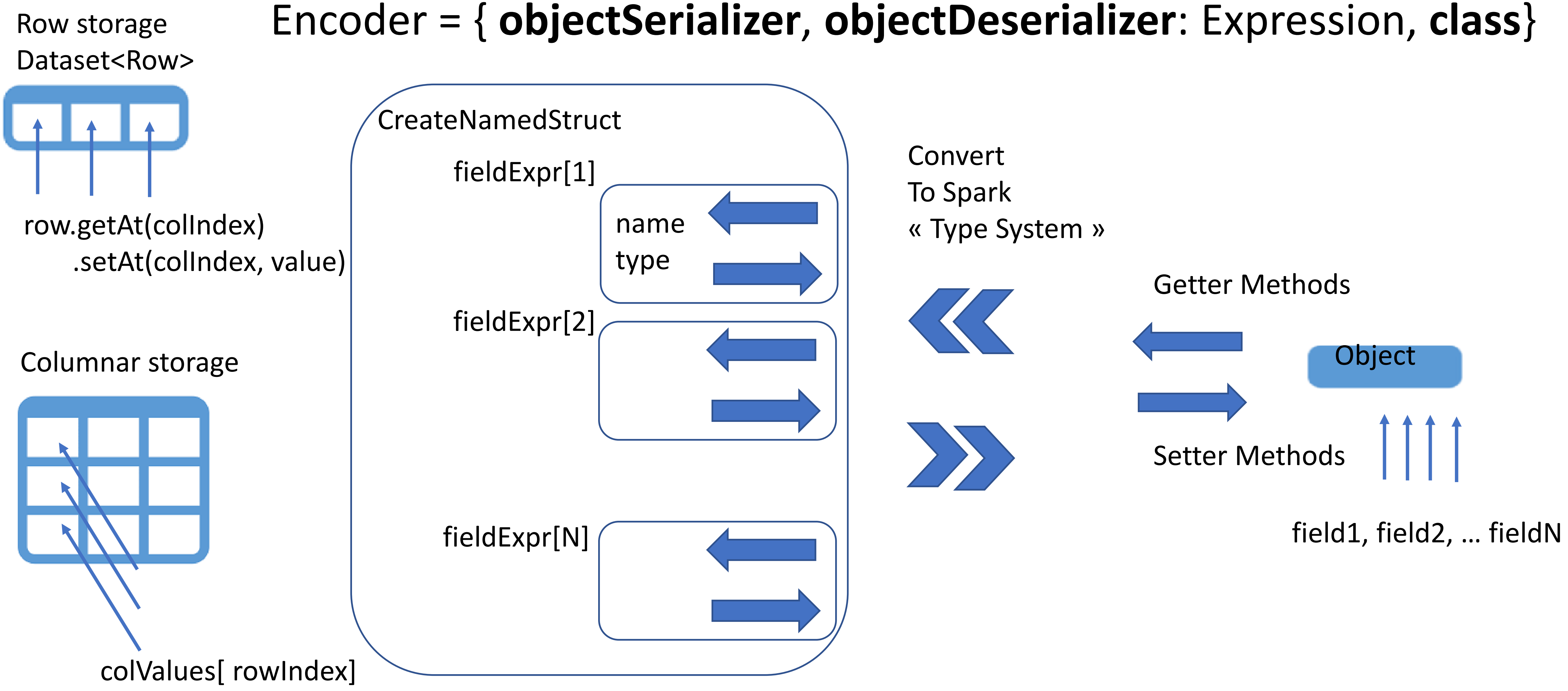


class Dataset<T> { internals... }



Encoder<T>

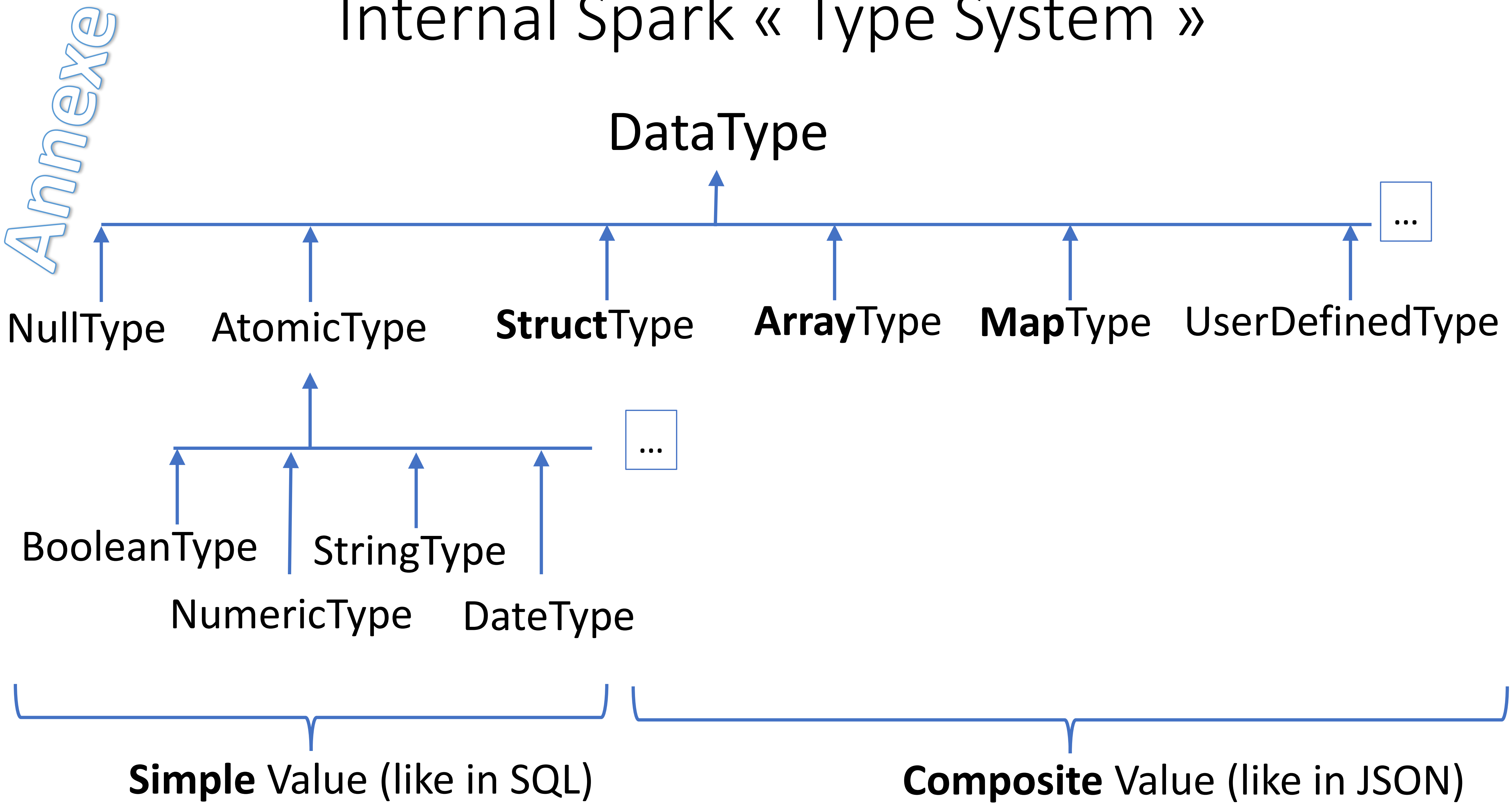
Encoder = { **objectSerializer**, **objectDeserializer**: Expression, **class**}



Types, Nested struct/map/array

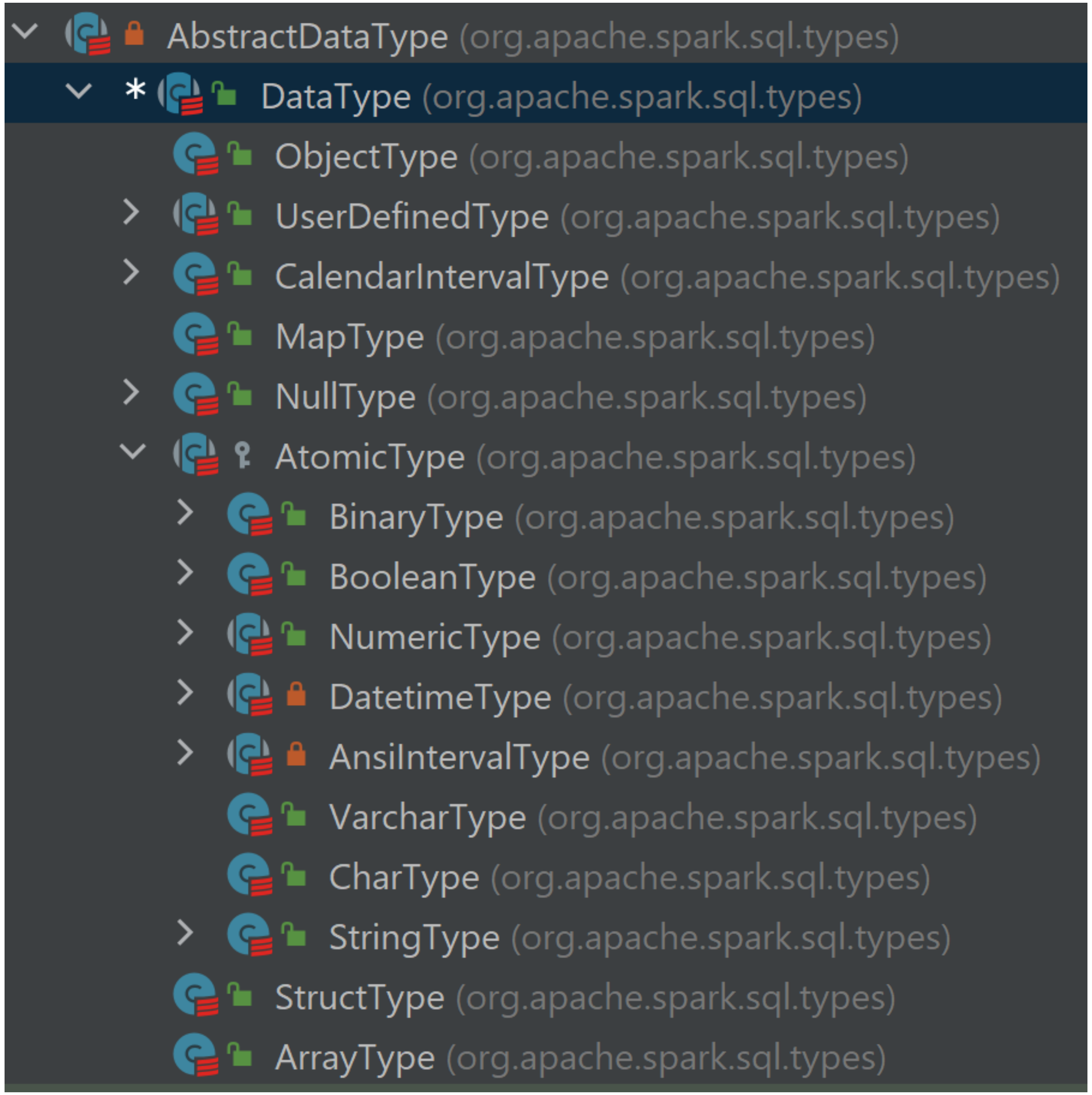
SQL "lateral view"

Internal Spark « Type System »



Annexe

Spark DataType

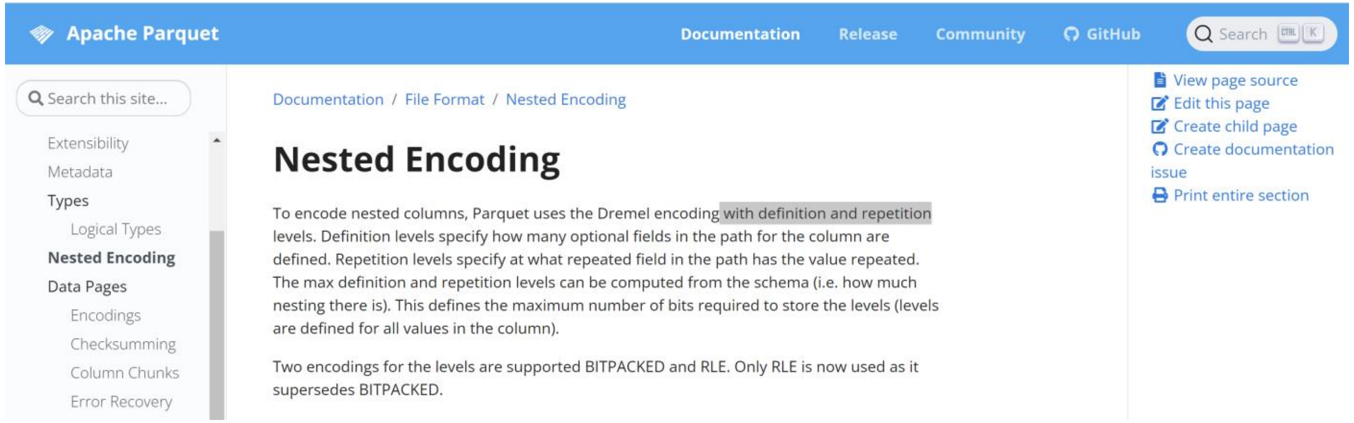


Hive - Spark SQL supports Struct, List, Map ...

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 »

DataType

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      |
+---------+----------+
```

Annexe

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) { return x + y }

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

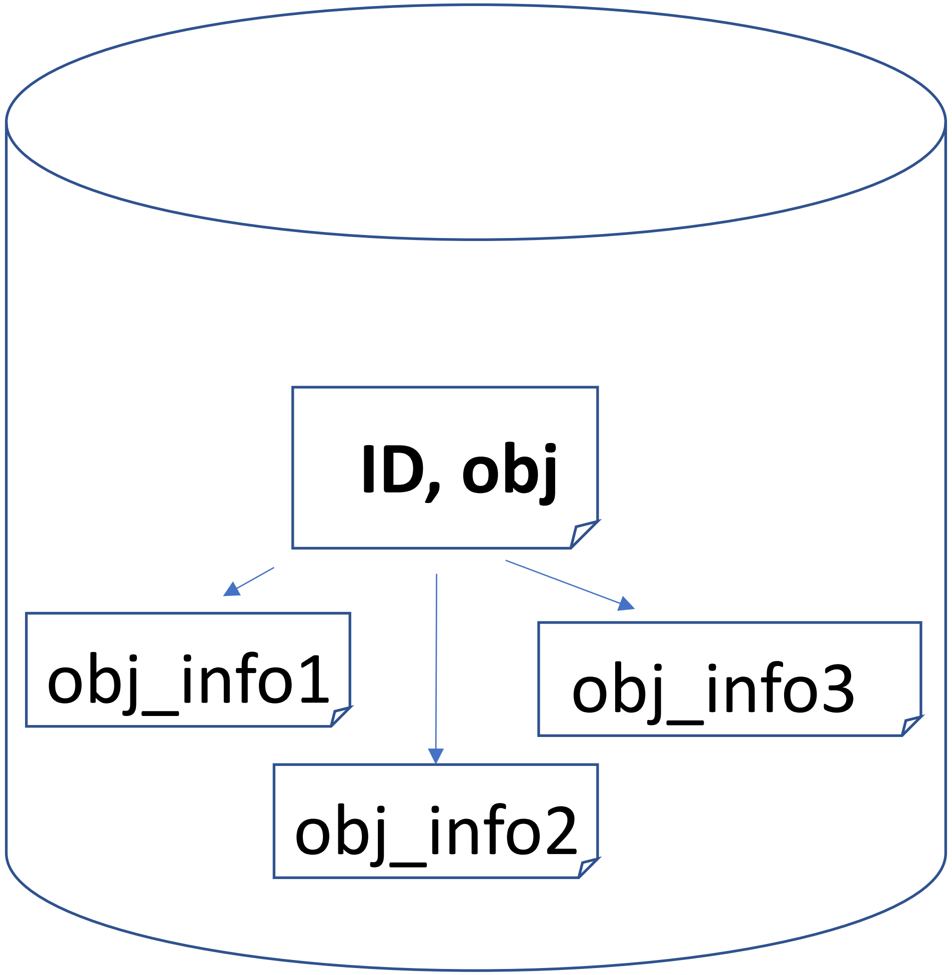
! = UDAF : Aggregate / Accumulator

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

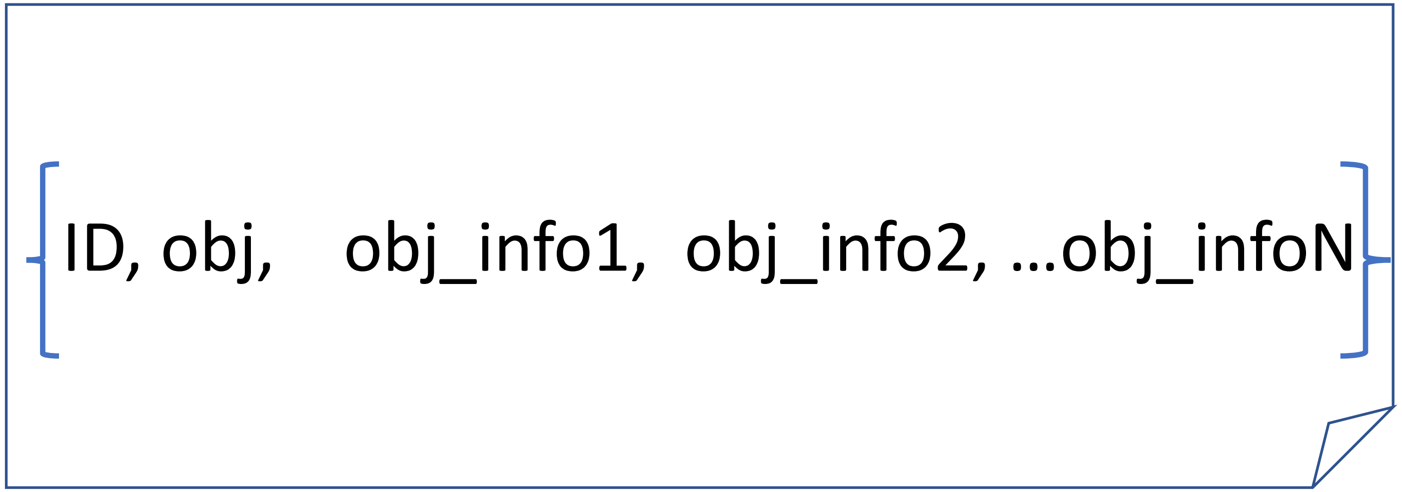
Object instance, Class with 3 methods:

```
init()
add(value)
Result getResult()
```

List, Map, Struct … denormalize data, avoid Joins



Normalized relationnal database



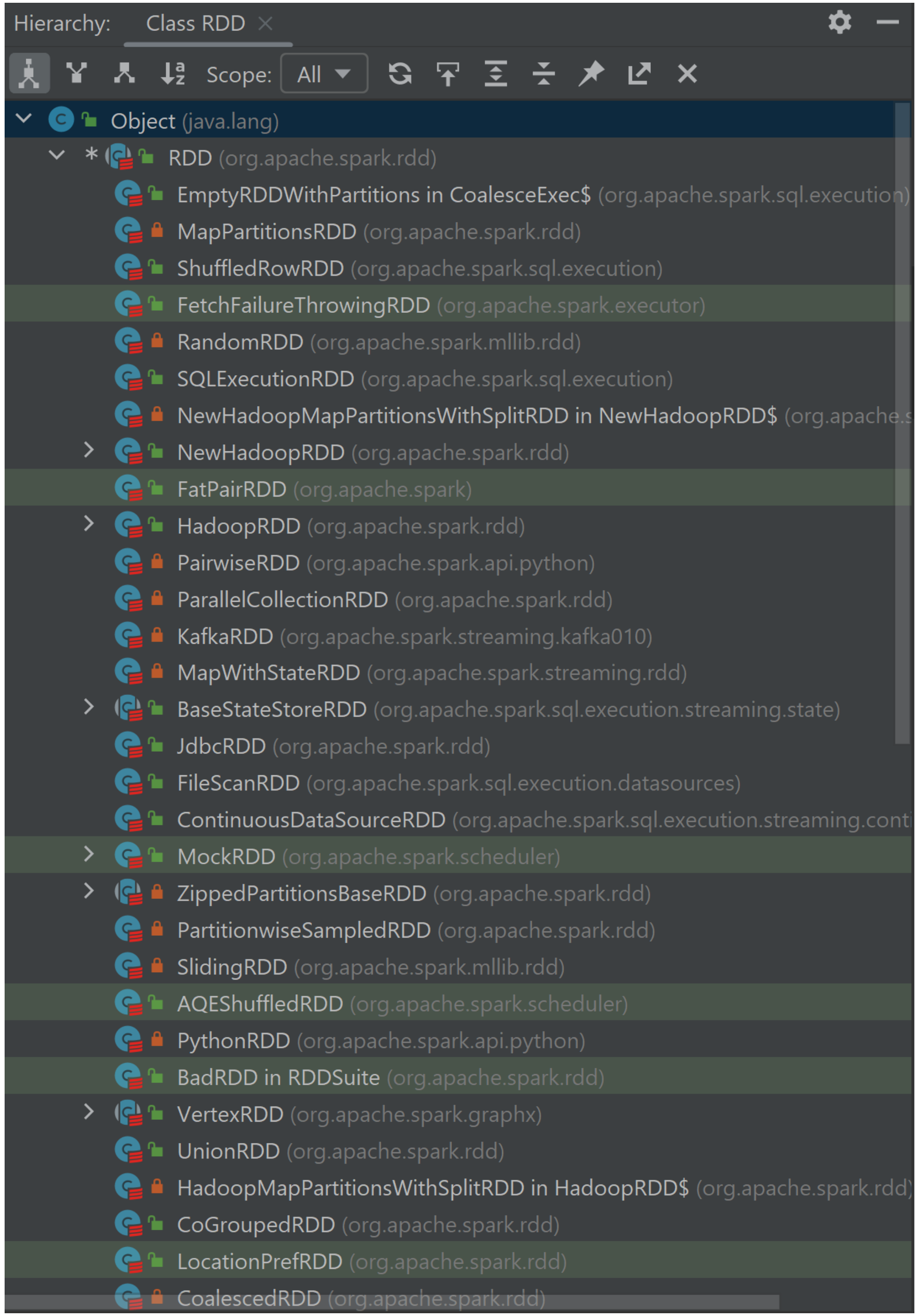
Efficient DE-normalized analytics system

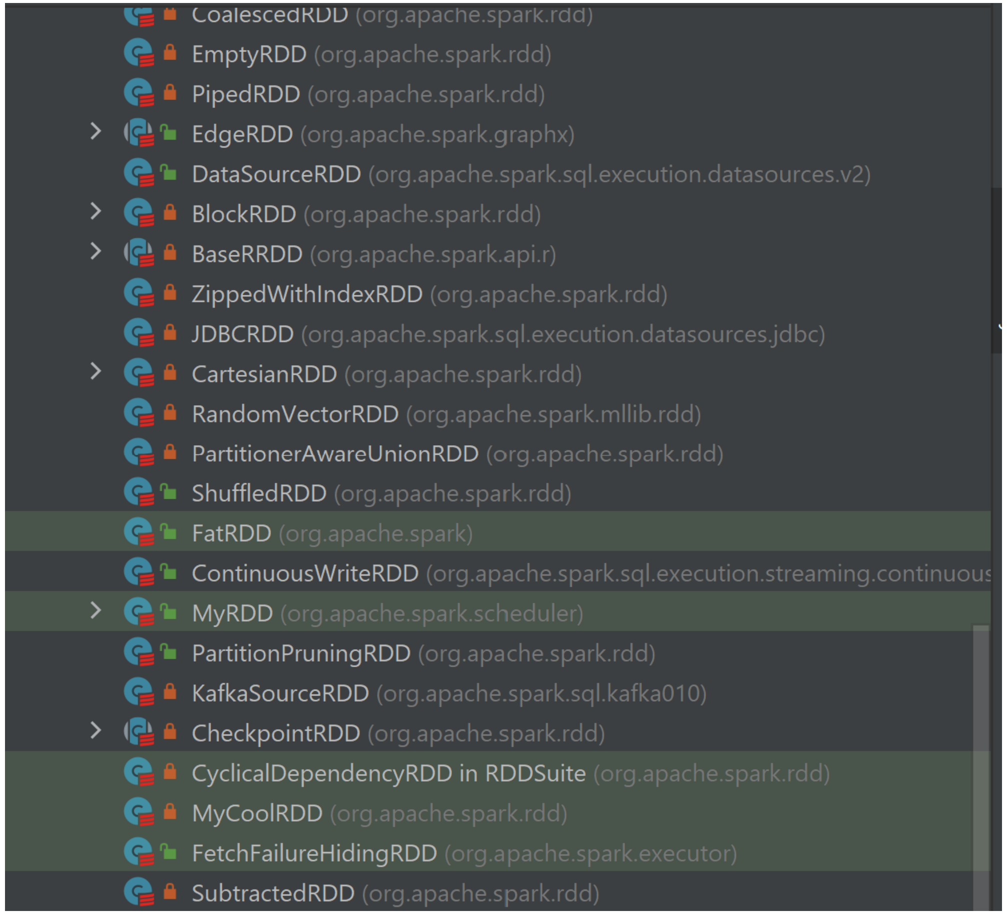
Data Transformations

Data Lineage

DAG (= Directed Acyclic Graph)

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





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) {
```

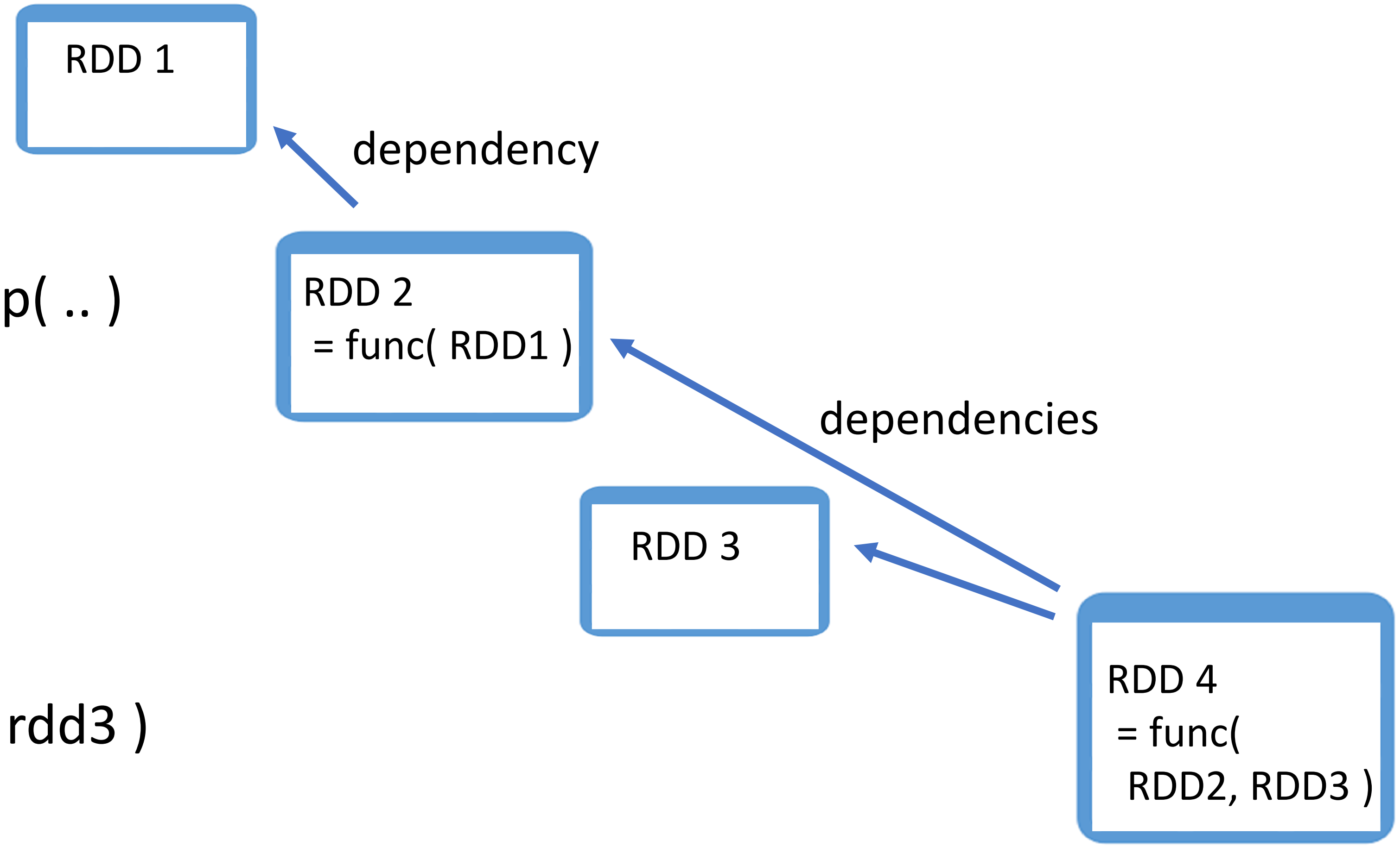
Call transform function => Create new RDD (linked)

RDD rdd1 = … read

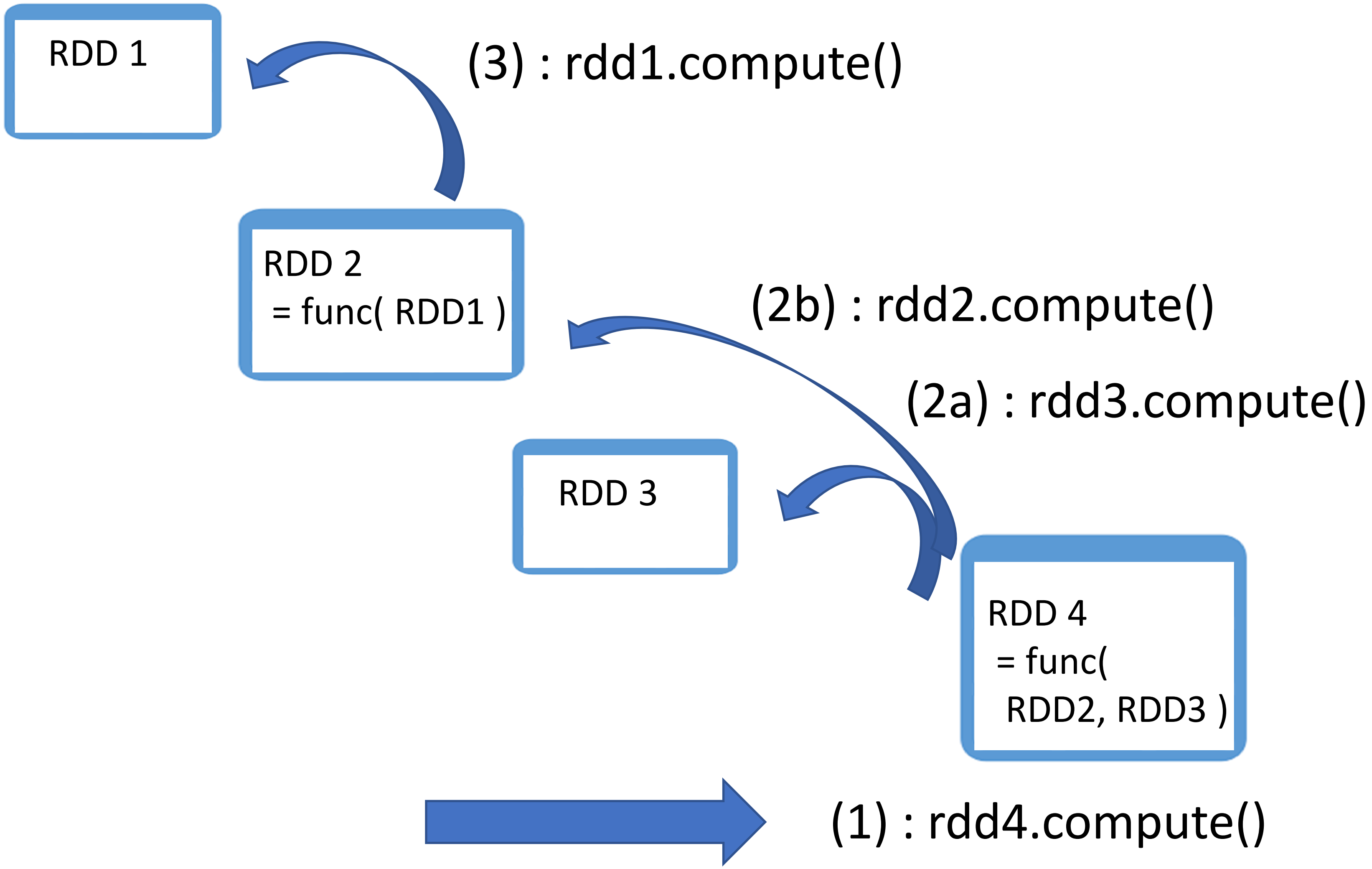
RDD rdd2   = rdd1 .map(..)

RDD rdd3 = .. read

RDD rdd4 = rdd2 .join(rdd3)



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



Dataset Transformations API
(Similar to RDD methods, different sub-classes)

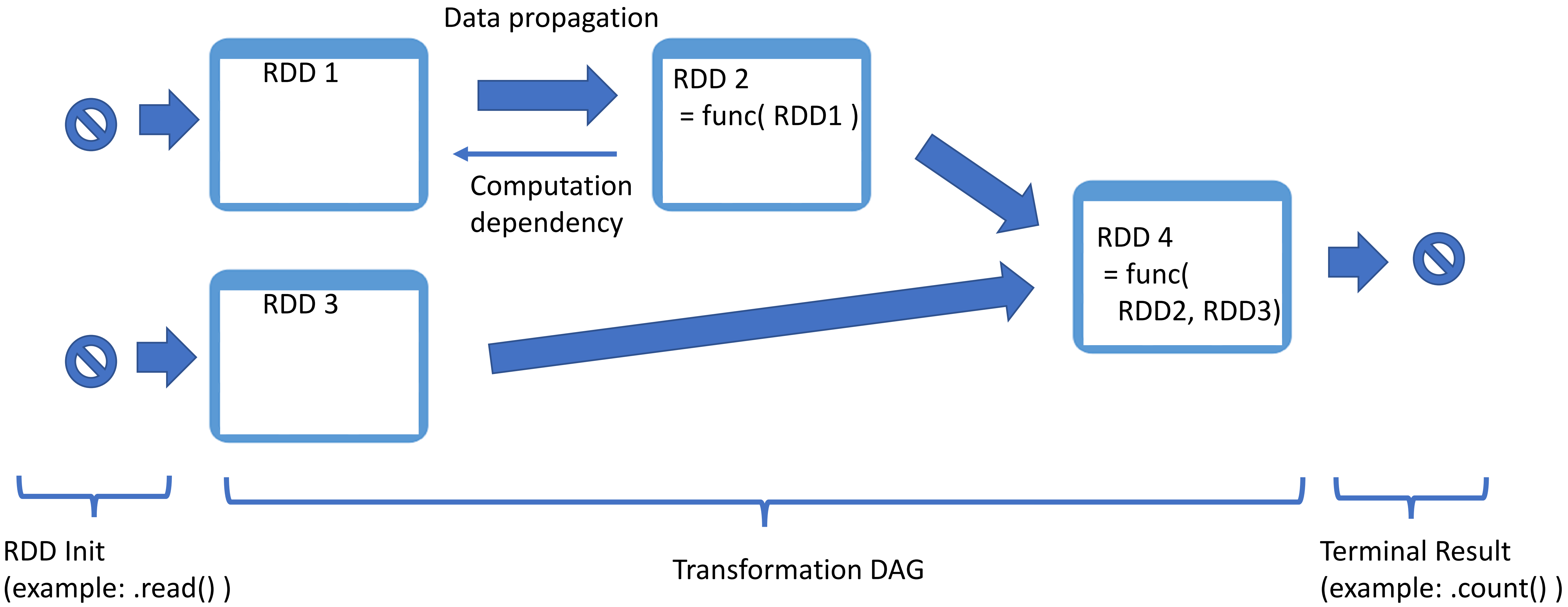
```
abstract class RDD<T> {

    public RDD<U> map(func<T,U>) {
       return new MapRDD(func);
    }
}
```

```
class Dataset<T> {

    public Dataset<U> map(func<T,U>,  Encoder<U>) {
       return Dataset(
               new QueryExecution(sc, new MapLogicalPlan(this, func)),
               encoder));
    }
}
```

```
class MapRDD extends RDD {
   ..
}
```

```
class QueryExecution { .. }
abstract class LogicalPlan extends Expression {   ..  }

class MapLogicalPlan extends MapLogicalPlan {   .. }
```

Dependencies : DAG (Directed Acyclic Graph)



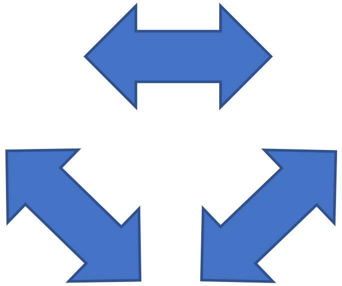
3 equivalent formalisms:
SSA create Api, Expression Algebra, DAG

SSA = Single State Assignments
RDD API

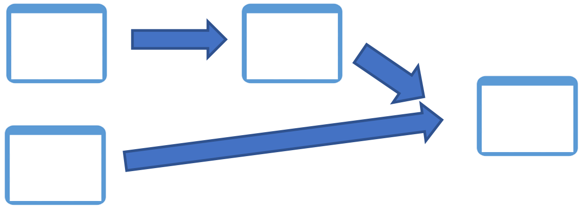
```
RDD rdd1 = ... read
RDD rdd2   =  rdd1 .map( .. )
RDD rdd3 = .. read
RDD rdd4 = rdd2 .join( rdd3 )
```

Expression Algebra, Sql

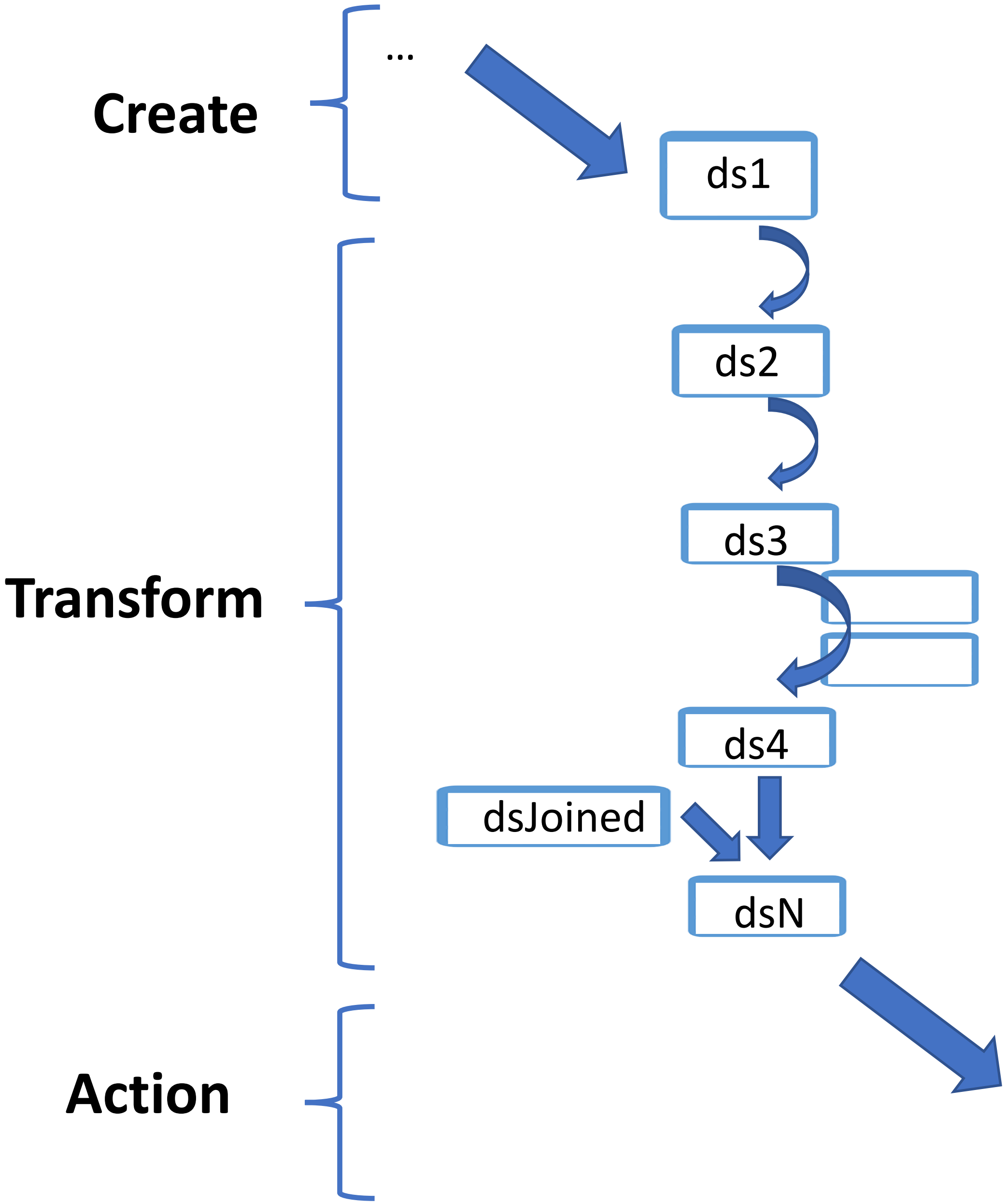
```
SELECT  map(t1) FROM Table1 t1
JOIN Table2 t2 on ..

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



DAG



Dataset operations…



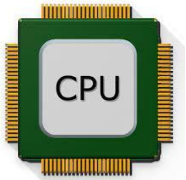

```
Dataset<T1>   ds1 =  spark.read ….
```

```
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 Transformation != Action

Transformations = lazy, returning another Dataset (on driver),
but virtual "data" would be computed later on executors

!=

Actions = immediate, return value (long or List) on Driver

Avoiding Dataset multiple recomputations (compromise RAM+Disk <-> CPU)

Dataset API

.cache()

// idem **.persist**(MEMORY_AND_DISK) cf also DISK_ONLY, ..

.unpersist() // maybe unnecessary (gc on driver)?

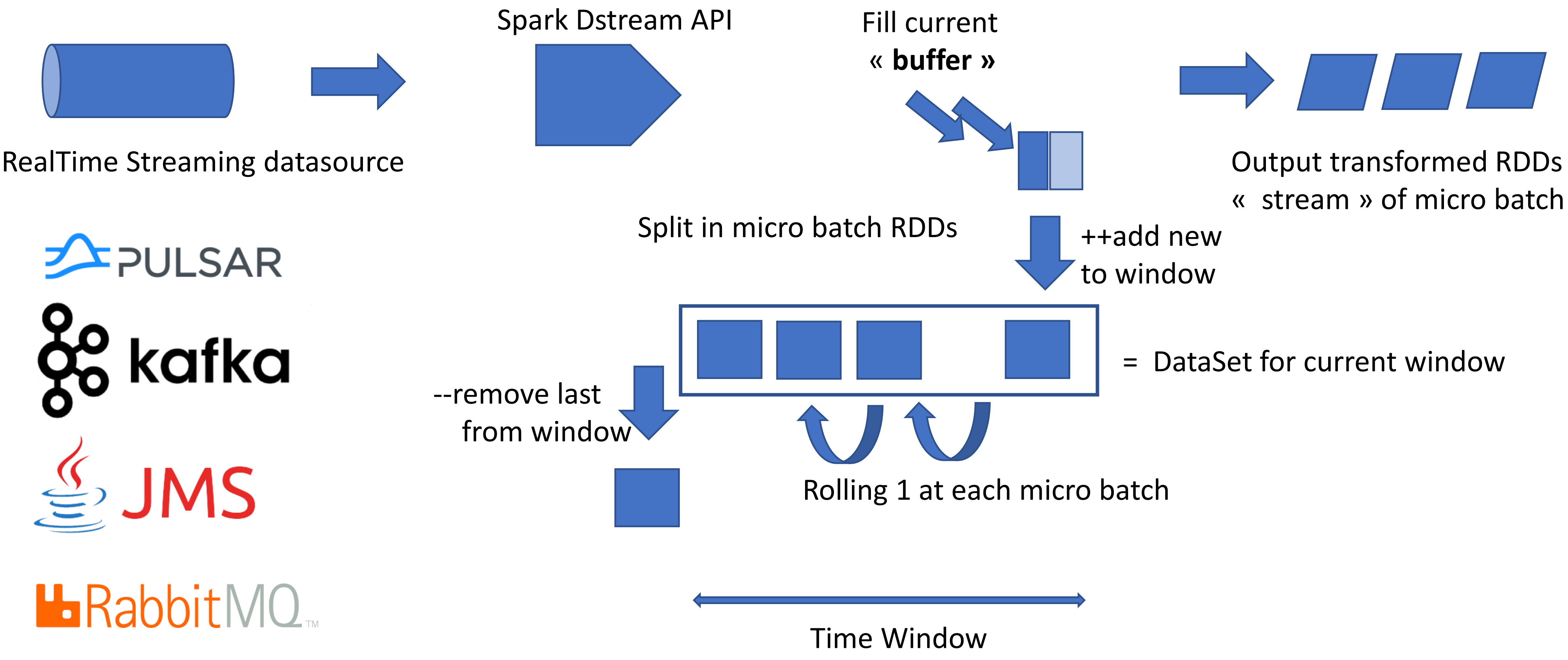
newDs = ds.**localCheckpoint()** // idem cache() + cut from DAG to read from memory

newDs = ds.**checkpoint()** // idem ".save()" to reliable storage + cut from DAG to ".read()"

cf next doc

Spark Streaming : micro Batches

Spark DStream API … as + enriched DataSet API



DStream
using similar to Dataset API

all the API in Dataset are similar on DStream

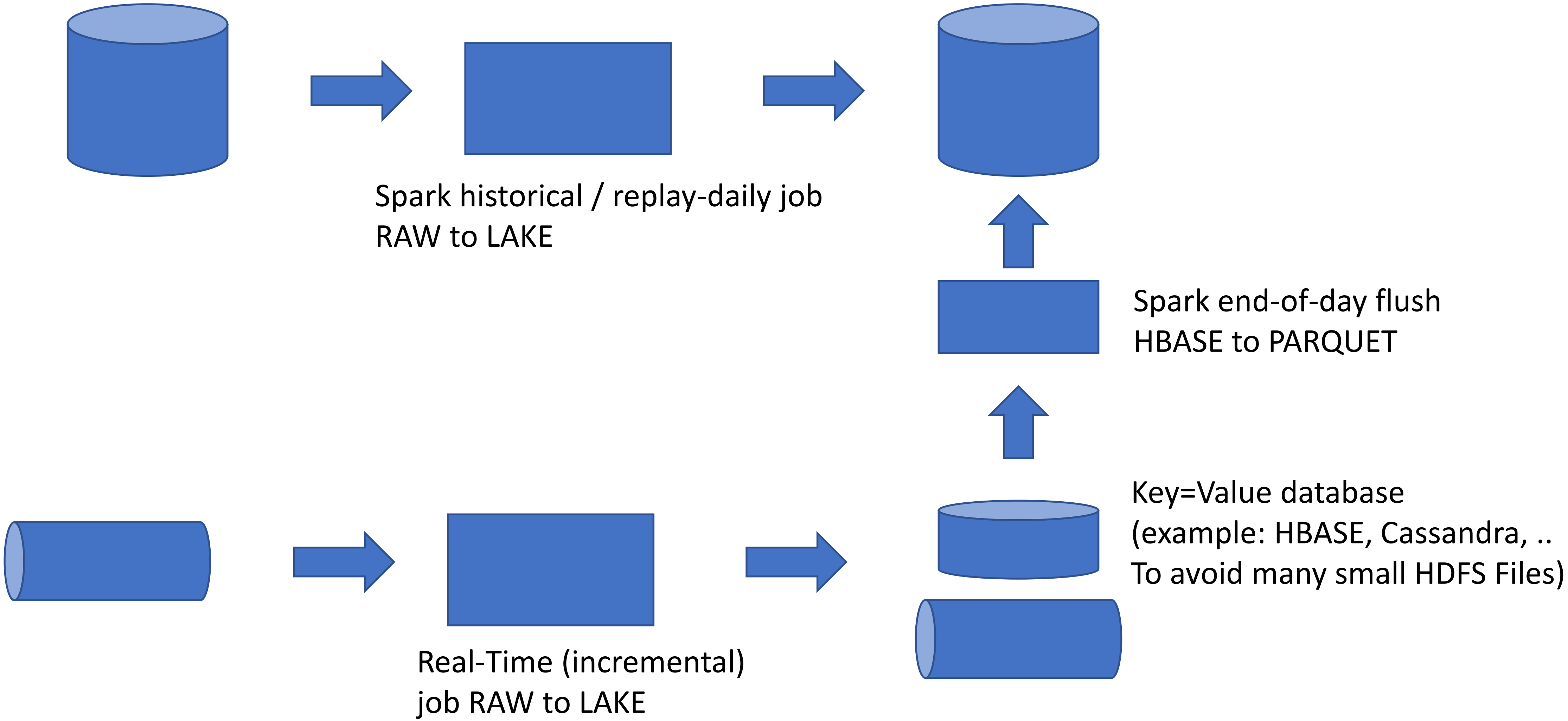
example:

DStream sourceDstream = ...
DStream transformedDstream = sourceDstream.filter(..).map(..).repartition(..);

is similar to

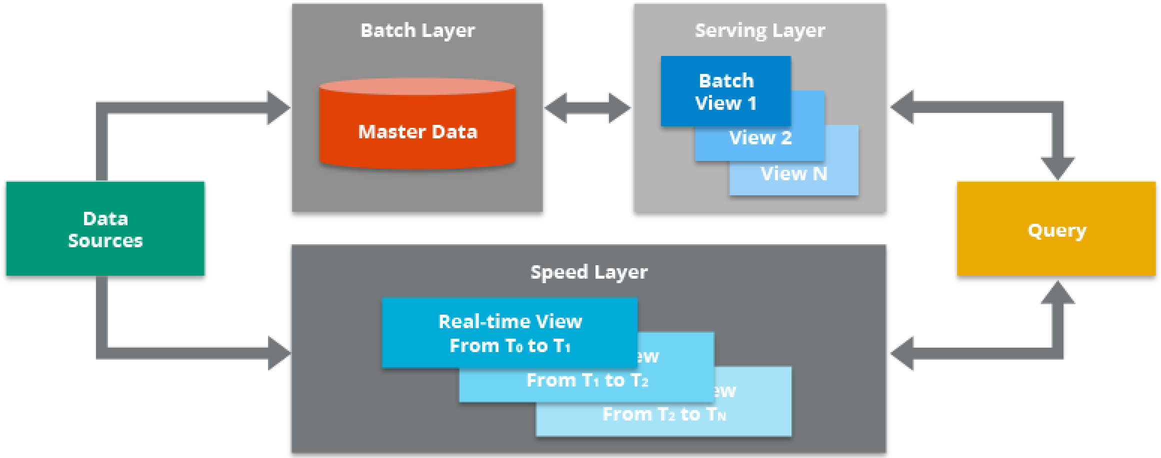
Dataset nextDs_5s = ... sourceDstream. <<next microbatch dataset>>
Dataset transformedDs = nextDs_5s.filter(..).map(..).repartition(..);

Typical Usage ... Streaming vs Daily

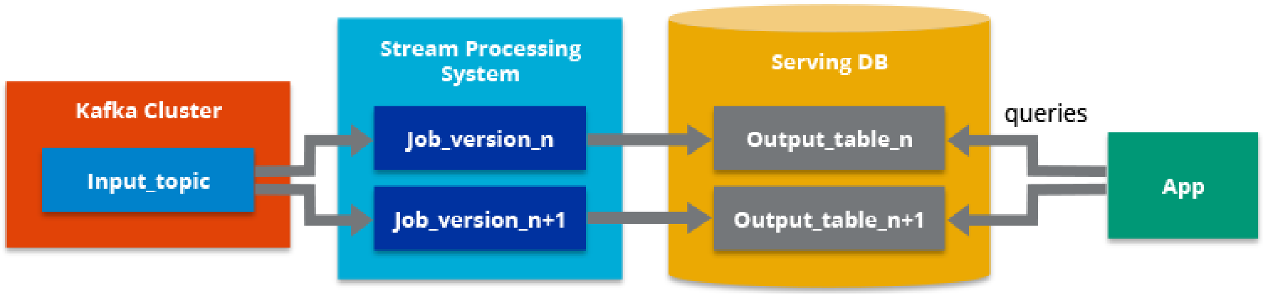


Typical Architectures.. Lambda vs Kappa

Lambda



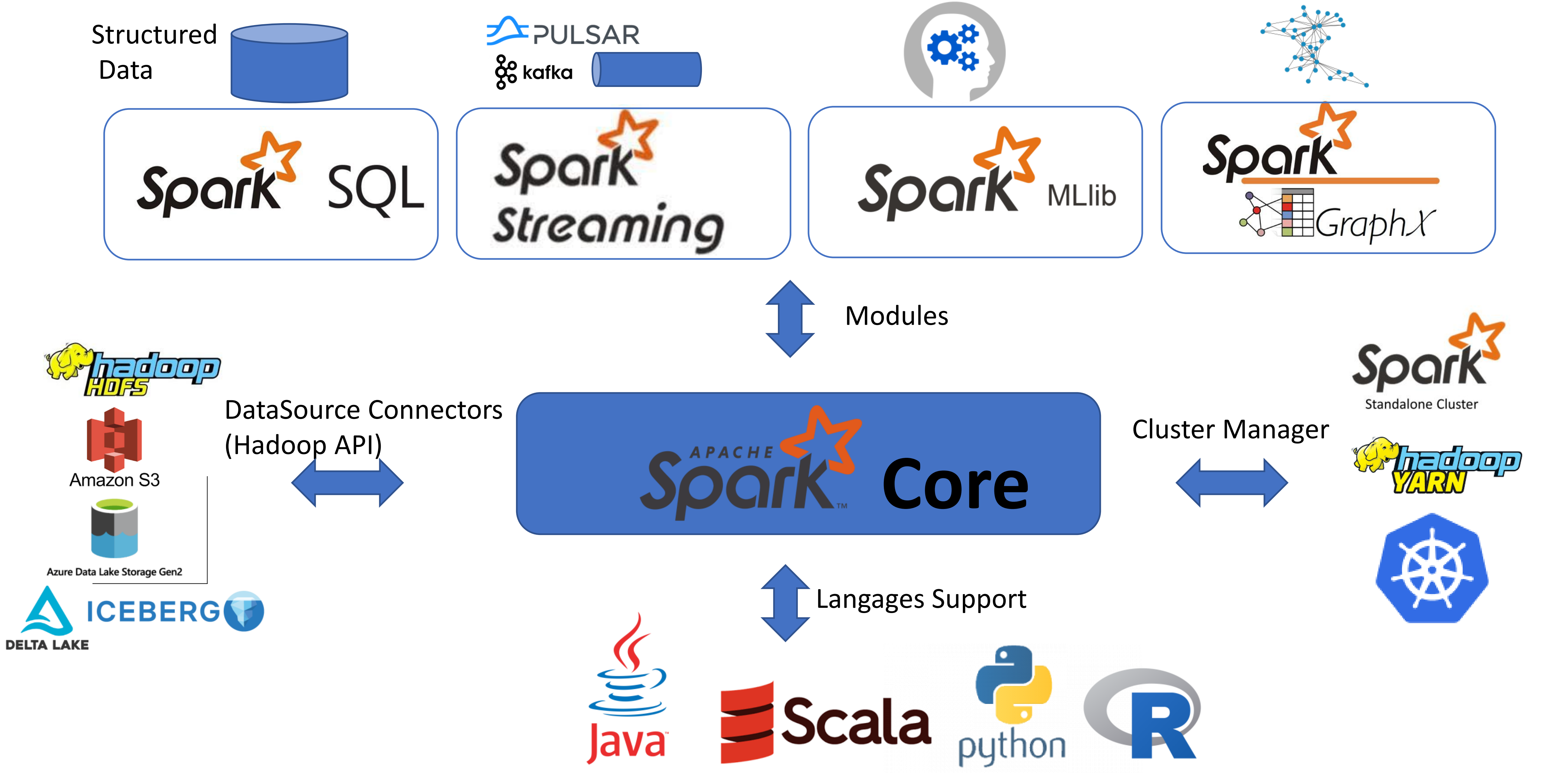
Kappa



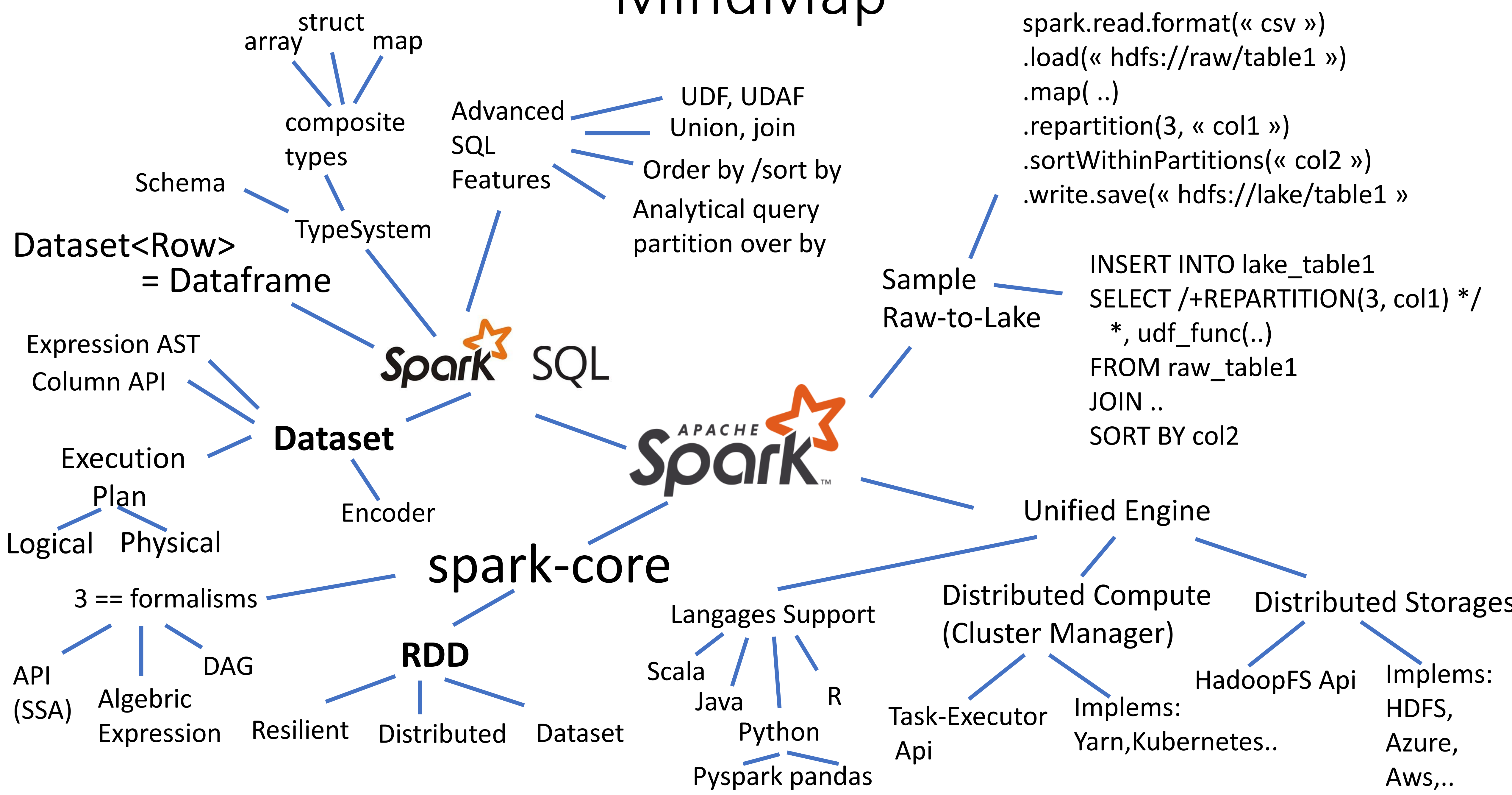
Take Away

What Did you learn ?

Spark-Core + ...


Structured Data
PULSAR
kafka
Spark SQL
Spark Streaming
Spark MLlib
Spark GraphX
Modules
hadoop HDFS
Amazon S3
Azure Data Lake Storage Gen2
DELTA LAKE
ICEBERG
DataSource Connectors
(Hadoop API)
APACHE Spark Core
Cluster Manager
Spark
Standalone Cluster
hadoop YARN
Langages Support
Java
Scala
python
R

MindMap


struct
array
map
composite types
Schema
TypeSystem
Advanced SQL Features
UDF, UDAF
Union, join
Order by /sort by
Analytical query partition over by
spark.read.format(« csv »)
.load(« hdfs://raw/table1 »)
.map(..)
.repartition(3, « col1 »)
.sortWithinPartitions(« col2 »)
.write.save(« hdfs://lake/table1 »
Dataset<Row> = Dataframe
Spark SQL
Sample Raw-to-Lake
INSERT INTO lake_table1
SELECT /+REPARTITION(3, col1) */
*, udf_func(..)
FROM raw_table1
JOIN ..
SORT BY col2
Expression AST
Column API
Dataset
Execution Plan
Logical
Physical
Encoder
APACHE Spark
Unified Engine
spark-core
3 == formalisms
API (SSA)
Algebric Expression
DAG
RDD
Resilient
Distributed
Dataset
Langages Support
Scala
Java
Python
R
Pyspark pandas
Distributed Compute (Cluster Manager)
Task-Executor Api
Implems: Yarn,Kubernetes..
Distributed Storages
HadoopFS Api
Implems: HDFS, Azure, Aws,..