Introduction to Big Data

SPARK API

Le Ngoc Thanh – Nguyen Ngoc Thao
{lnthanh, nnthao}@fit.hcmus.edu.vn

Outline

- Spark Structured API
- Resilient Distributed Datasets (RDD)
- Distributed shared variables

Spark Structured API

Spark Structured APIs

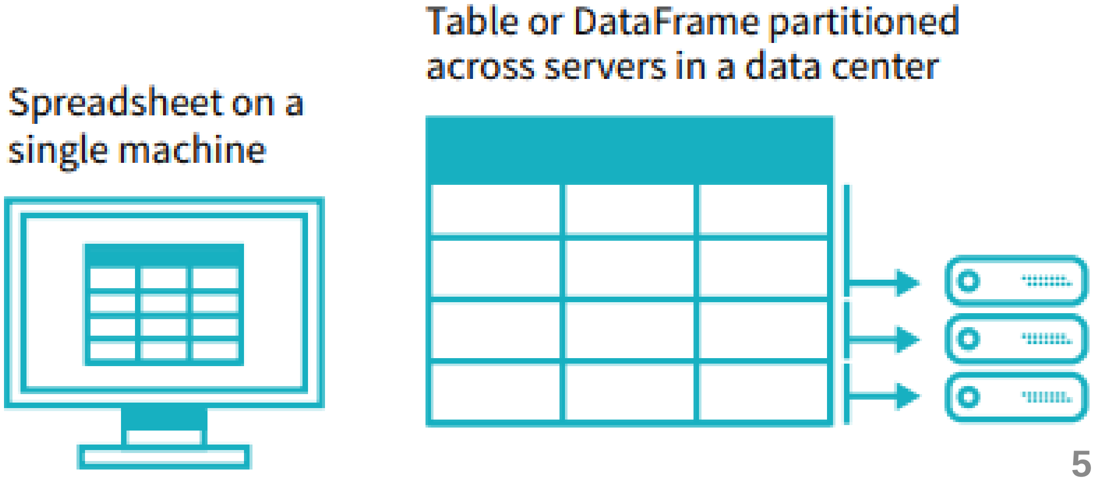
- The collection of structured APIs is a tool for manipulating all sorts of data.
- Three core types of distributed collection APIs include

Datasets | DataFrames (DF) | SQL Tables and Views

- Most APIs apply to both batch and streaming computation.

DataFrames and Datasets

- DataFrrames/Datasets are distributed table-like collections with well-defined rows and columns.
  - Each column must have the same number of rows and its type must be consistent for every row in the collection.
- They represent immutable and lazily evaluated plans
  - A plan defines what operations to apply to the data residing at a location to generate some output.



Tables: Columns and Rows

- `Column` serves a simple type, a complex type or a *null value*.
  - Simple type: integer or string. Complex type: array or map.
- `Row` is nothing more than a record of data.
  - It is either created manually from scratch or obtained from SQL, `RDDs`, or data sources.

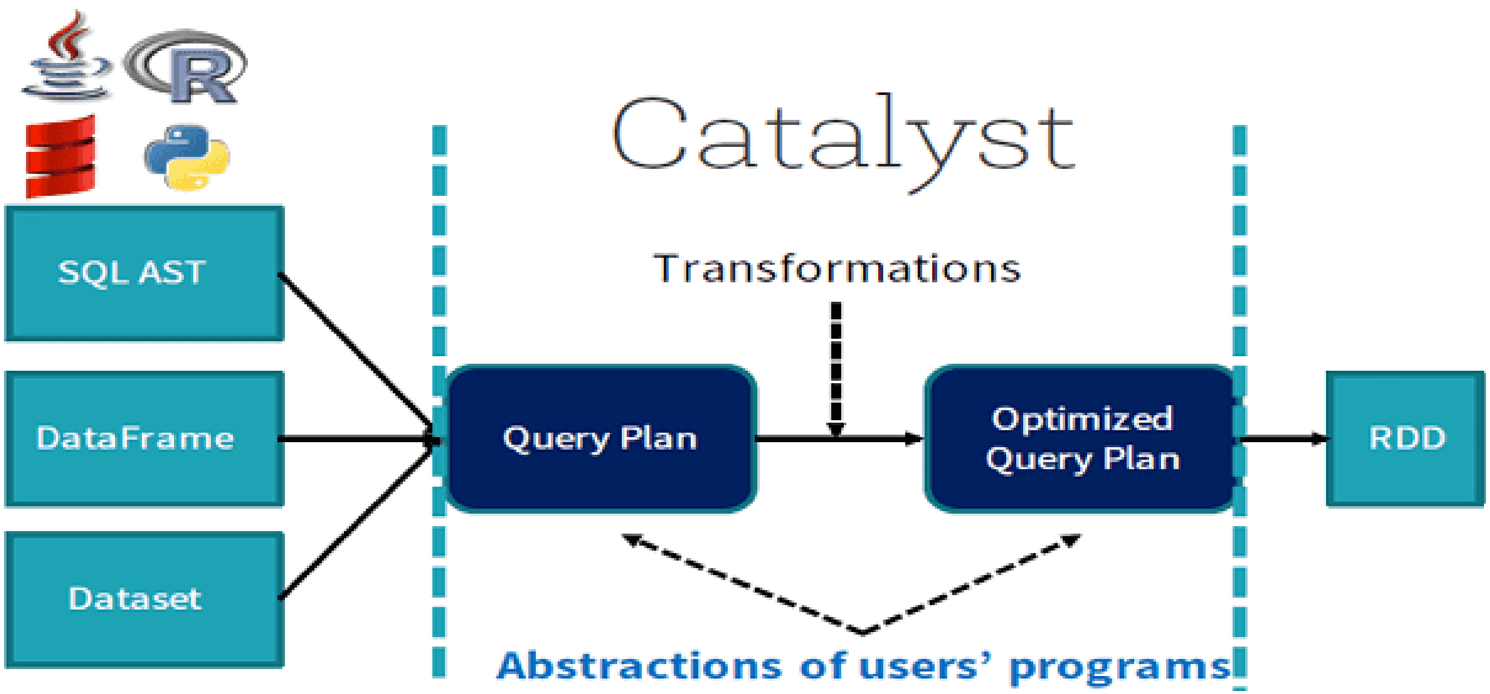
```
root
 |-- Category: string (nullable = true)
 |-- ID: long (nullable = true)
 |-- Truth: boolean (nullable = true)
 |-- Value: double (nullable = true)
```

```
+--------+---+-----+------+
|Category| ID|Truth| Value|
+--------+---+-----+------+
|       A|  1| true|121.44|
|       B|  2|false|300.01|
|       C|  3| null| 10.99|
|       E|  4| true| 33.87|
+--------+---+-----+------+
```

- A schema defines the name and type for every column.

Structured Spark types

- Spark uses **Catalyst** that maintains its own type information through the planning and processing of work.
  - An expression written in an input language is converted to its corresponding internal Catalyst representation.



Structured Spark types

- Spark types map directly to the different language APIs, using a lookup table for each of these.
  - Available for Scala, Java, Python, SQL, and R.

```scala
// in Scala
val df = spark.range(500).toDF("number")
df.select(df.col("number") + 10)
```

The following code segments do not perform addition in Scala or Python.

```python
# in Python
df = spark.range(500).toDF("number")
df.select(df["number"] + 10)
```

They actually perform addition purely in Spark.

- Various execution optimizations of significant differences

Spark DataFrames

- `DataFrame APIs` organizes the data into named columns like a table in relational database.
  - Programmers define schemas on a distributed collection of data.
- Each row in a `DataFrame` is of object type `Row`.
  - Each column must have same number of rows.
  - `Row` type: Spark's internal representation of its optimized in-memory format $\rightarrow$ highly specialized and efficient computation.
- An immutable collection, lazily evaluated plan
- Same efficiency gains to all language APIs

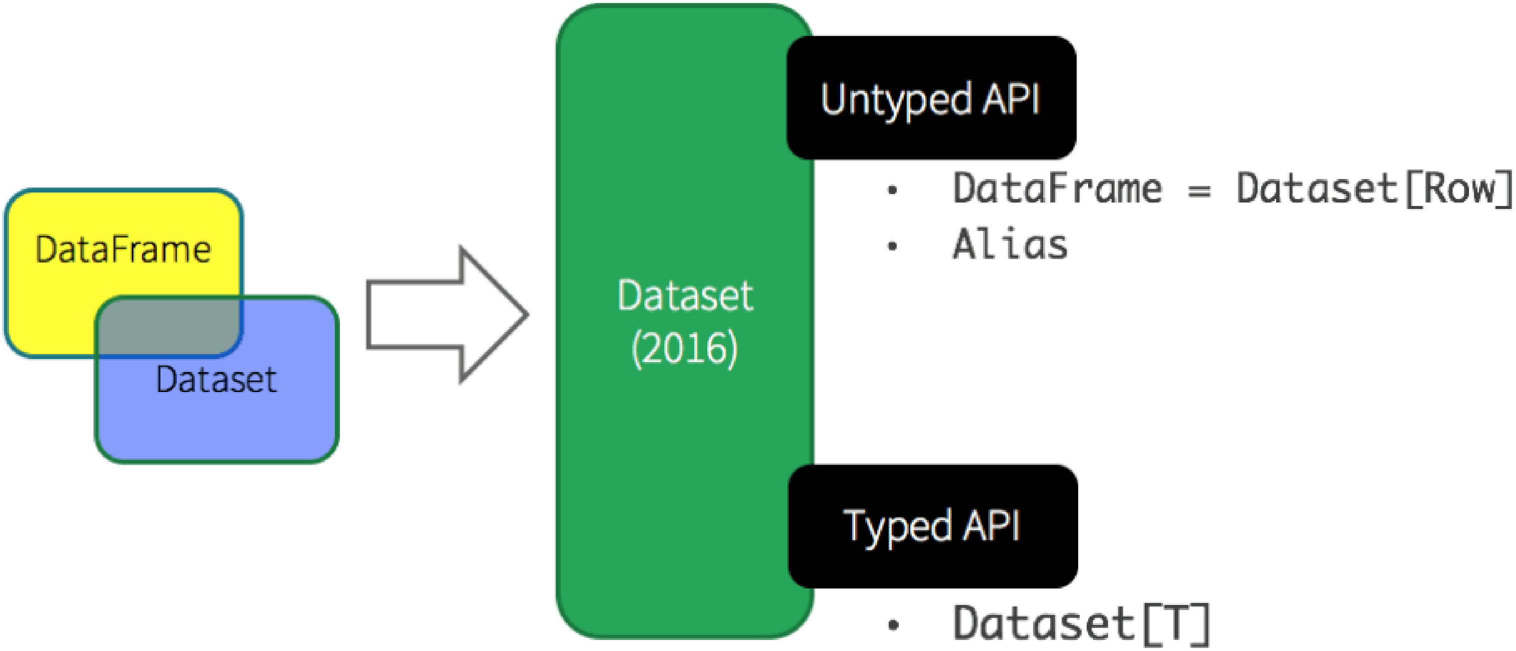
Spark DataFrames

- In-built optimization: Significantly improve the performance with Catalyst
- Fully Hive compatible
  - Access all hive data, queries, UDFs, etc. using Spark SQL from hive MetaStore; and execute queries against these hive databases
- Structured, semi-structured, and highly structured data support
- Multi-language support: Available in Python, R, Scala, and Java
- Schema support
  - A schema can be defined manually or read from a data source which defines the column names and their data types.
- **Type safety:** Not strictly typed. Compile time safety is not supported.

Spark Datasets

- A combination of `RDD` and `DataFrame`
  - It enables functional programming like `RDD` APIs and relational queries and execution optimization like `DataFrame` APIs.
- Type-safe: Datasets conforms the specification at compile time
  - It conforms the specification using defined case classes (for Scala) or Java beans (for Java).
- Limited language support: Only available in JVM-based languages, like Java and Scala
  - Python and R are not supported because these are dynamically typed languages.
- High garbage collection
  - JVM types can cause high garbage collection and object instantiation cost.

Spark DataFrames vs. Datasets

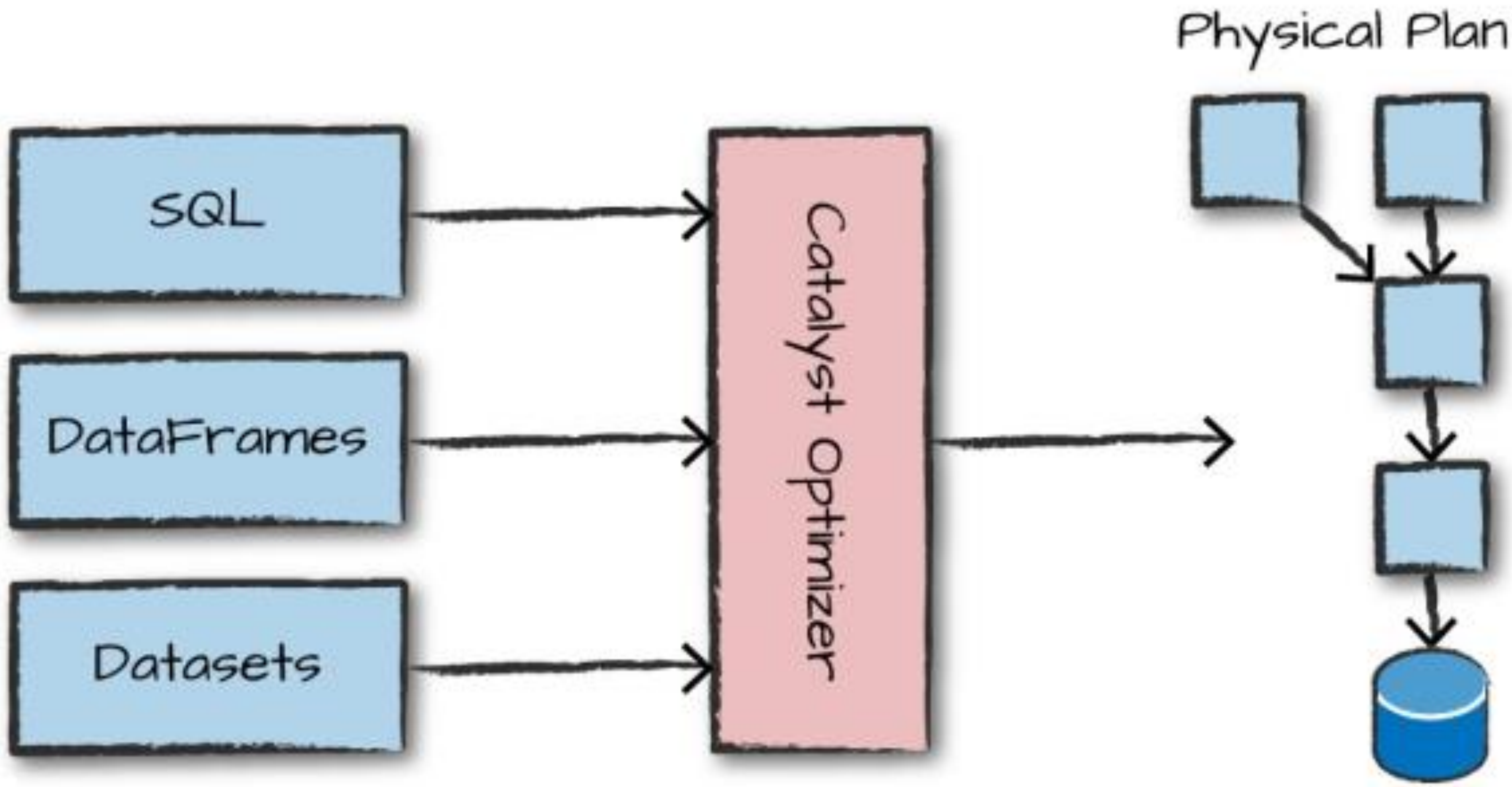


- `DataFrame` is considered as an alias for a collection of generic objects `Dataset[Row]`.
  - A `Row` is a generic untyped JVM object.
- `Dataset` is a collection of strongly-typed JVM objects, dictated by a case class defined in Scala or a class in Java.

Structured API Execution

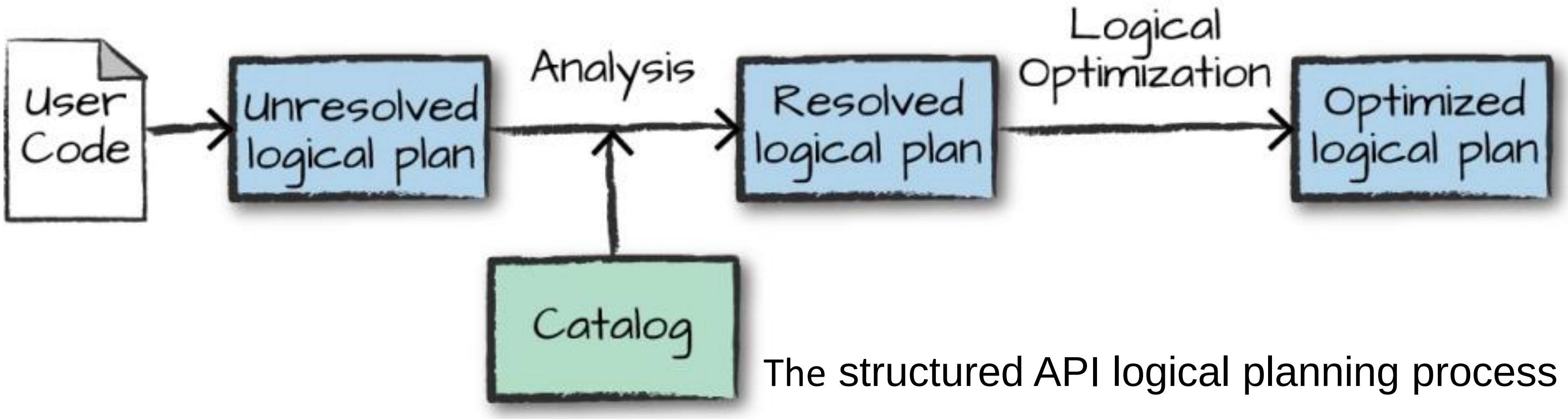
Structured API execution

- The execution of a single structured API query from user code to executed code



Logical planning

- Take user code and convert it into a logical plan



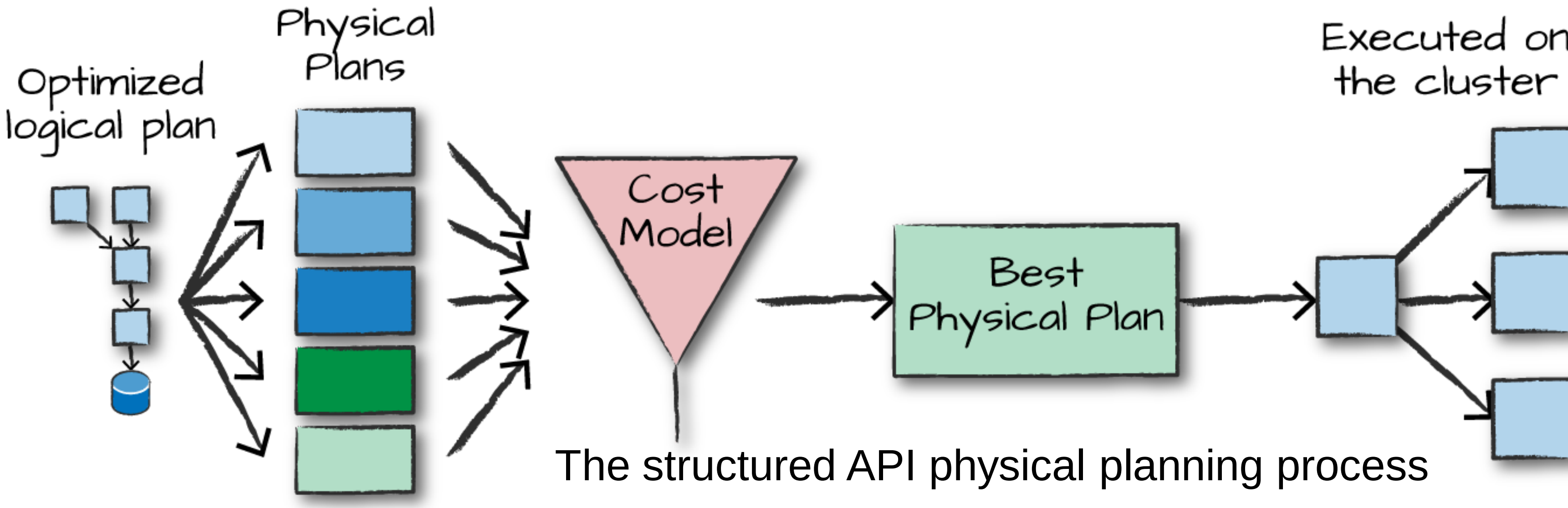
The structured API logical planning process

- A logical plan represents a set of abstract transformations that do not refer to executors or drivers.
  - It purely converts user's expressions into the most optimized version.

Logical planning

- Convert user code into an unresolved logical plan
  - Unresolved: although the code might be valid, the tables or columns that it refers to might or might not exist.
- Use **Catalog** to resolve columns and tables in the analyzer
  - **Catalog:** a repository of all table and DataFrame information
  - The analyzer might reject the unresolved logical if the required table or column name does not exist in the Catalog.
- Apply rule-based optimizations the logical plan
  - Constant folding, predicate pushdown, projection pruning, etc.
- Pass the resolved plan through the Catalyst Optimizer
  - This uses a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections.

Physical planning



The structured API physical planning process

- Specify how the logical plan will execute on the cluster
  - Generate different physical execution strategies and compare them through a cost model
  - E.g., choose how to perform a given join by looking at the physical attributes of the table (how big the table is or how big its partitions are).
- Result in a series of `RDDs` and transformations

Execution

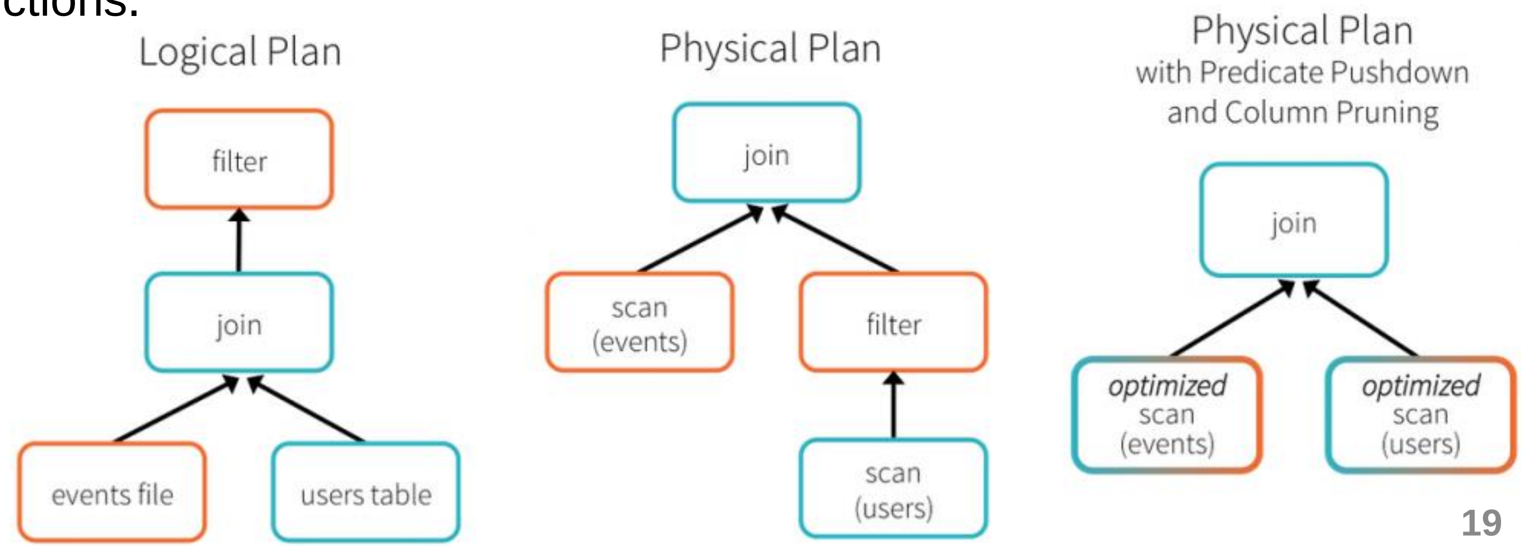
- With a physical plan, Spark runs all the code over `RDDs`.
  - `RDD` = the lower-level programming interface of Spark
- Generate native Java bytecode and possibly perform further optimizations at runtime
- Finally, the result is returned to the user.

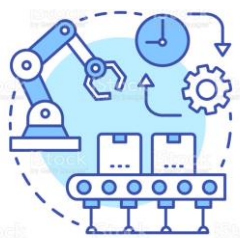
Plan optimization: An example

- Spark intermix DataFrame with custom Python, Java, R, or Scala code.

```
def add_demographics(events):
    u = sqlCtx.table("users")                     # Load partitioned Hive table
    events \
      .join(u, events.user_id == u.user_id) \     # Join on user_id
      .withColumn("city", zipToCity(df.zip))      # Run udf to add city column
events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "New York").select(events.timestamp).collect()
```

- Optimizations take place as late as possible → Spark SQL can optimize even across functions.





Basic DataFrame operations

Creating a DataFrame

- A `DataFrame` can be created manually or from a data

```
# create a DataFrame from a JSON file
df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
```

```
# create a DataFrame from an existing RDD named somerdd
df = somerdd.toDF()
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|   15|
|    United States|            Croatia|    1|
|    United States|            Ireland|  344|
|            Egypt|      United States|   15|
|    United States|              India|   62|
+-----------------+-------------------+-----+
only showing top 5 rows
```

Looking at the schema

- A schema defines the name and type of data in each column, and thus tie everything together.
- Use the **`schema`** property or **`printSchema`** function

```
StructType(List(StructField(DEST_COUNTRY_NAME,StringType,true),
StructField(ORIGIN_COUNTRY_NAME,StringType,true),
StructField(count,LongType,true)))
```

```
# a schema can be displayed by either of the two functions
df.schema
df.printSchema()
```

```
root
 |-- DEST_COUNTRY_NAME: string (nullable = true)
 |-- ORIGIN_COUNTRY_NAME: string (nullable = true)
 |-- count: long (nullable = true)
```

Columns and Expressions

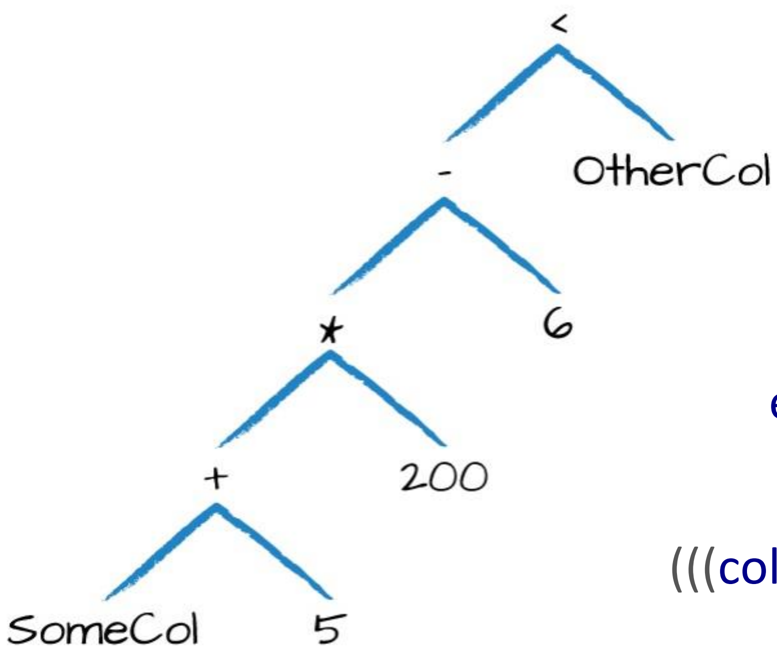
- A column represents a computation expression that can be performed on each individual record.
  - It is like Excel spreadsheet, R DataFrame, or pandas DataFrame.
- There are different ways to refer to columns.
- Use **`col`**, **`column`**, **`expr`**, or **`[]`** brackets

```
# use functions from Spark SQL
from pyspark.sql.functions import col, column
col("someColumnName")
column("someColumnName")

from pyspark.sql.functions import expr
expr("someColumnName")


# explicit column references
df[“someColumnName“]
df.DESC_COUNTRY_NAME
```

Columns and Expressions

- An expression is a set of transformations on one or more values in a record in a `DataFrame`.
- The **`expr`** function applies expressions to input columns to create a single value for each record.

expr("someCol – 5”) ~ col("someCol ”) – 5 ~ expr("someCol ”) – 5



Columns and transformations of those columns compile to the same logical plan as parsed expressions

expr("(((someCol + 5) * 200) - 6) < otherCol")

~

(((col("someCol") + 5) * 200) - 6) < col("otherCol")

Records and Rows

- Each row is a single record, which is an object of type `Row`.
  - It does not have any schema, which is unlike `DataFrame`.
- The values must be specified in the same order as in the schema of the `DataFrame` to which they might be appended

```
# create a row as a tuple of specified values
from pyspark.sql import Row
myRow = Row("Hello", None, 1, False)

# access elements in a row
myRow[0]
myRow[2]
```

- You cannot access a row in a typical procedural way since DataFrames are distributed.

Transformation and Actions

- A transformation reads a `DataFrame`, manipulates some of the columns, and eventually returns another `DataFrame`.
- Transformations are evaluated in a lazy fashion.
  - No Spark jobs are triggered, no matter the number of transformations are scheduled.

- An action either returns a result or writes to the disc.
- All actions are executed in an eager manner.
  - All unevaluated transformations are executed prior to the action.

Transformation and Actions

A sample DataFrame

id	name	gender	age
1111	simon jones	male	32
2222	joan hurt	female	21

A Spark transformation that creates a new column named group

id	name	gender	age
1111	simon jones	male	32
2222	joan hurt	female	21

→

id	name	gender	age	group
1111	simon jones	male	32	3
2222	joan hurt	female	21	2

A Spark action that counts the number of rows

id	name	gender	age
1111	simon jones	male	32
2222	joan hurt	female	21

2

Credit

Columns: select, expr, and selectExpr

- `select` does the DF equivalent of SQL queries on a table.

```
# in Python
df.select("DEST_COUNTRY_NAME").show(2)

-- in SQL
SELECT DEST_COUNTRY_NAME FROM dfTable LIMIT 2
```

```
+-----------------+
|DEST_COUNTRY_NAME|
+-----------------+
|    United States|
|    United States|
+-----------------+
```

```
+-----------------+-------------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|
+-----------------+-------------------+
|    United States|            Romania|
|    United States|            Croatia|
+-----------------+-------------------+
```

```
# in Python
df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)

-- in SQL
SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME FROM dfTable LIMIT 2
```

Columns: select, expr, and selectExpr

- **expr** is the most flexible reference that it refers to a plain column or a string manipulation of a column.

```
# in Python
df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)

-- in SQL
SELECT DEST_COUNTRY_NAME as destination FROM dfTable LIMIT 2
```

```
# expr can do more inside a select statement
df.select(expr("DEST_COUNTRY_NAME as destination")\
                         .alias("DEST_COUNTRY_NAME")).show(2)
```

Columns: select, expr, and selectExpr

- **`selectExpr`** helps build up complex expressions that create new `DataFrames`.

```
# in Python
df.selectExpr("*", "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME)
                                      as withinCountry").show(2)
```

```
+-----------------+-------------------+-----+-------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|
+-----------------+-------------------+-----+-------------+
|    United States|            Romania|   15|        false|
|    United States|            Croatia|    1|        false|
+-----------------+-------------------+-----+-------------+
```

```
-- in SQL
SELECT *, (DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry
FROM dfTable LIMIT 2
```

Columns: select, expr, and selectExpr

- `selectExpr` can also specify aggregations over the entire `DataFrame` by taking advantage of the available functions.

```
# in Python
df.selectExpr("avg(count)", "count(distinct(DEST_COUNTRY_NAME))").show(2)
```

```
+-----------+---------------------------------+
| avg(count)|count(DISTINCT DEST_COUNTRY_NAME)|
+-----------+---------------------------------+
|1770.765625|                              132|
+-----------+---------------------------------+
```

```
-- in SQL
SELECT avg(count), count(distinct(DEST_COUNTRY_NAME)) FROM dfTable LIMIT 2
```

Columns: withColumn and drop

- **`withColumn`**: add a column

```
+-----------------+-------------------+-----+---------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|numberOne|
+-----------------+-------------------+-----+---------+
|    United States|            Romania|   15|        1|
|    United States|            Croatia|    1|        1|
+-----------------+-------------------+-----+---------+
```

```
# in Python
df.withColumn("numberOne", lit(1)).show(2)

-- in SQL
SELECT *, 1 as numberOne FROM dfTable LIMIT 2
```

```
# show all columns after adding a new column to df
df.withColumn("withinCountry",\
        expr("ORIGIN_COUNTRY_NAME == DEST_COUNTRY_NAME")).columns
```

```
['DEST_COUNTRY_NAME', 'ORIGIN_COUNTRY_NAME', 'count', 'withinCountry']
```

Columns: withColumn and drop

- **`withColumn`** can also be used to replicate a column

```
df.withColumn("Destination", expr("DEST_COUNTRY_NAME")).columns
```

```
['DEST_COUNTRY_NAME', 'ORIGIN_COUNTRY_NAME', 'count’, ‘Destination']
```

- **`withColumnRenamed`**: rename a column

```
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns
```

```
[dest', 'ORIGIN_COUNTRY_NAME', 'count']
```

- **`drop`**: remove a column

```
# drop a single column
df.drop("ORIGIN_COUNTRY_NAME")
# drop multiple columns at a time
df.drop("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME")
```

Columns: size and array_contains

- **size** tells the number of elements in an array column.

```
# create a new column that show the number of actors in each movie
df.withColumn("actors", f.size(df.cast)).show()
```

```
+--------------------------------------------+-------------+----------------------------+------+
|cast                                        |genres       |title                       |actors|
+--------------------------------------------+-------------+----------------------------+------+
|[The Three Stooges]                         |[Comedy, Short]|An Ache in Every Stake    |1     |
|[Ingrid Bergman, Warner Baxter, Susan Hayward]|[Drama]    |Adam Had Four Sons          |3     |
|[Herbert Marshall, Virginia Bruce]          |[]           |Adventure in Washington     |2     |
|[Tom Tyler, Frank Coghlan Jr., Louise Currie]|[]          |Adventures of Captain Marvel|3     |
|[Merle Oberon, Rita Hayworth]               |[Comedy]     |Affectionately Yours        |2     |
+--------------------------------------------+-------------+----------------------------+------+
```

- **array_contains** checks if an element is in the array

```
# select only rows in which the field 'cast' includes 'Tom Tyler' as an element
df.filter(f.array_contains(df.cast, "Tom Tyler")).show()
```

Rows: filter, where, and distinct

- **filter** and **where**: select rows according some criteria

```
# in Python
df.filter(col("count") < 2).show(2)
# or
df.where("count < 2").show(2)

-- in SQL
SELECT * FROM dfTable WHERE count < 2 LIMIT 2
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Croatia|    1|
|    United States|          Singapore|    1|
+-----------------+-------------------+-----+
```

- **distinct**: get unique rows

```
# in Python
df.select("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME").distinct().count()

-- in SQL
SELECT COUNT(DISTINCT(ORIGIN_COUNTRY_NAME, DEST_COUNTRY_NAME))
FROM df
```

Rows: sort and orderBy

- **`sort`** and **`orderBy`** sort the specified columns in ascending order by default

```
+--------------------+-------------------+-----+
|   DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+--------------------+-------------------+-----+
|               Malta|      United States|    1|
|Saint Vincent and...|      United States|    1|
|       United States|            Croatia|    1|
|       United States|          Gibraltar|    1|
|       United States|          Singapore|    1|
+--------------------+-------------------+-----+
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|     Burkina Faso|      United States|    1|
|    Cote d'Ivoire|      United States|    1|
|           Cyprus|      United States|    1|
|         Djibouti|      United States|    1|
|        Indonesia|      United States|    1|
+-----------------+-------------------+-----+
```

```
# sort a single column
df.sort("count").show(5)

# sort multiple column
df.orderBy("count", "DEST_COUNTRY_NAME").show(5)
```

Rows: asc and desc

- The ordering is defined by using **asc** and **desc**.

```
# in Python
from pyspark.sql.functions import desc, asc
df.orderBy(col("count").desc(), col("ORIGIN_COUNTRY_NAME").asc()).show()
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
Guyana	United States	64
United States	Austria	63
United States	Guyana	63
United States	Grenada	62
United States	India	62
Austria	United States	62

```
-- in SQL
SELECT * FROM dfTable
ORDER BY count DESC, ORIGIN_COUNTRY_NAME ASC
```

- Note: expr("count desc") seems not working.

Rows: limit and show

- **`limit`**: get top N rows in the original `DataFrame`

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|   15|
|    United States|            Croatia|    1|
|    United States|            Ireland|  344|
|            Egypt|      United States|   15|
|    United States|              India|   62|
+-----------------+-------------------+-----+
```

```
# in Python
df.limit(5).show()
# or
df.show(5)

-- in SQL
SELECT * FROM df LIMIT 6
```

- **`take`** and **`collect`**: return lists instead of `DataFrames`.
- **`show`** with argument is used to limit how many rows shown.

Rows manipulation

- Just chain many filters sequentially instead of putting them in the same expression, and let Spark handle the optimization

```
# in Python
df.where(col("count") < 2).where(col("ORIGIN_COUNTRY_NAME") != "Croatia")\
.show(2)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|          Singapore|    1|
|          Moldova|      United States|    1|
+-----------------+-------------------+-----+
```

```
-- in SQL
SELECT * FROM dfTable
WHERE count < 2 AND ORIGIN_COUNTRY_NAME != "Croatia"
LIMIT 2
```

Rows: sample and randomSplit

- **`sample`**: randomly select a fraction of records from the original `DataFrame`.

```
# sample with replacement until 50% the total number of rows is reached
seed = 5
withReplacement = False
fraction = 0.5
df.sample(withReplacement, fraction, seed)
```

- **`randomSplit`**: randomly split the original `DF` into two different ones according to a predefined ratio.

```
# split the original DF into two subsets with ratio 1:3
dataFrames = df.randomSplit([0.25, 0.75], seed)
dataFrames[0].count() > dataFrames[1].count() # False
```

Rows: explode

- **explode** returns a new row for each element in the array.

```
# a separate row is reserved for each element in the field 'cast'
df.withColumn("individual", f.explode(df.cast)).show()
```

```
+--------------------+---------------+--------------------+-----------------+
|                cast|         genres|               title|       individual|
+--------------------+---------------+--------------------+-----------------+
|  [The Three Stooges]|[Comedy, Short]|An Ache in Every ...|The Three Stooges|
|[Ingrid Bergman, ...|        [Drama]|  Adam Had Four Sons|   Ingrid Bergman|
|[Ingrid Bergman, ...|        [Drama]|  Adam Had Four Sons|    Warner Baxter|
|[Ingrid Bergman, ...|        [Drama]|  Adam Had Four Sons|    Susan Hayward|
|[Herbert Marshall...|             []|Adventure in Wash...| Herbert Marshall|
|[Herbert Marshall...|             []|Adventure in Wash...|   Virginia Bruce|
|[Tom Tyler, Frank...|             []|Adventures of Cap...|        Tom Tyler|
|[Tom Tyler, Frank...|             []|Adventures of Cap...|Frank Coghlan Jr.|
|[Tom Tyler, Frank...|             []|Adventures of Cap...|    Louise Currie|
|[Merle Oberon, Ri...|       [Comedy]|Affectionately Yours|     Merle Oberon|
|[Merle Oberon, Ri...|       [Comedy]|Affectionately Yours|    Rita Hayworth|
+--------------------+---------------+--------------------+-----------------+
```

Aggregation functions

- Aggregation is the act of collecting something together over one or more columns in a table.
- That needs a key (or grouping) and an aggregation function.
  - The function specifies how to transform the columns, which must produce one result for each group, given multiple input values.

Aggregation functions

- `count` gets the number of values in one or more columns.
  - `countDistinct` gives the number of unique values.

```
from pyspark.sql.functions import count
df.select(count("ORIGIN_COUNTRY_NAME")).show()
df.select(countDistinct("ORIGIN_COUNTRY_NAME")).show()
```

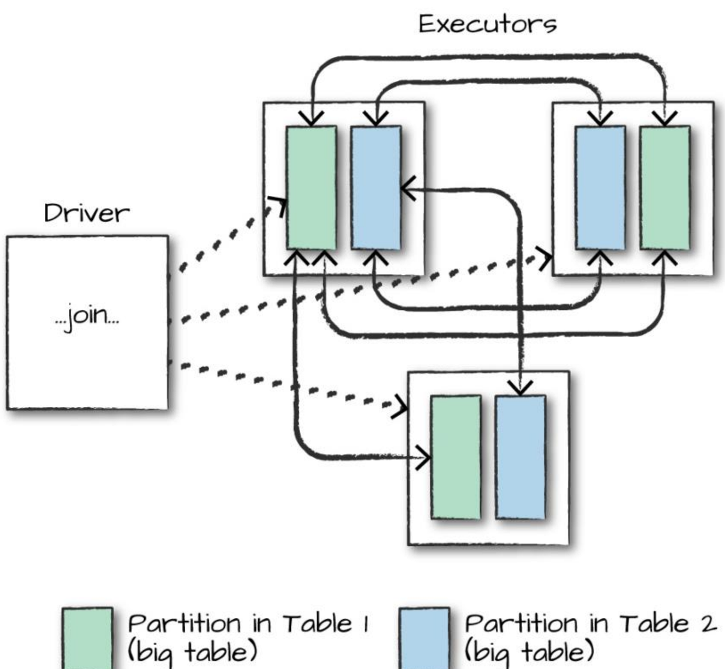
- `sum` adds all the values in a column
  - `sumDistinct` consider distinct values.

```
from pyspark.sql.functions
import sum
df.select(sum("count"))
df.select(sumDistinct("count"))
```

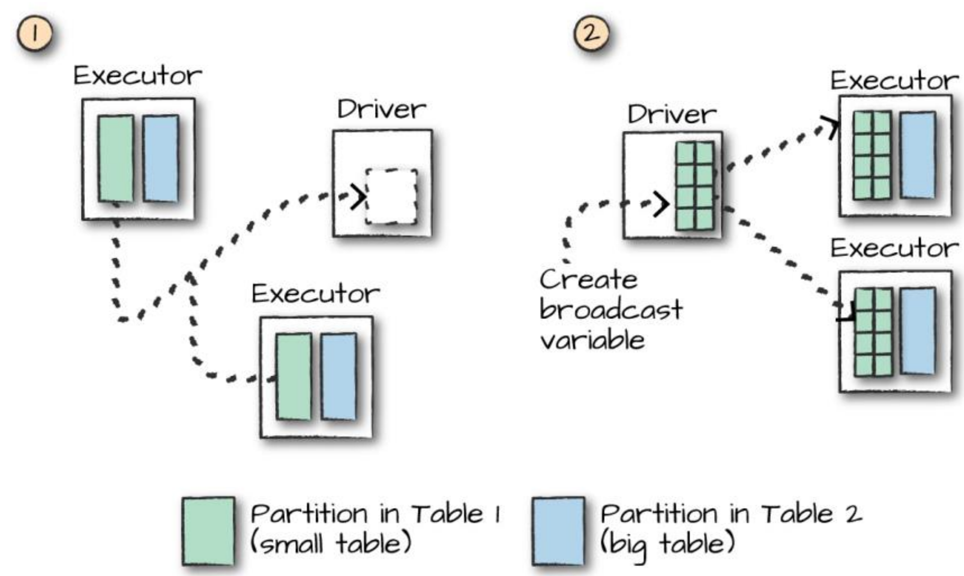
- Similar for
  - `avg:` calculate average by dividing `sum` by `count`
  - `first` and `last`: get the first and last values from a DF (based on rows, not values)
  - `min` and `max`: extract the minimum and maximum values from a DF

Joins

- There are a variety of different join types available in Spark.



Joining two big tables



A broadcast join

Finally, our Word Count!

```
linesDF = spark.read.text("ppap.txt")
linesDF.show(linesDF.count(),truncate = False)

from pyspark.sql import functions as f
wordsDF = linesDF.withColumn("word", f.explode(f.split(f.col("value"), " ")))\
   .groupBy("word")\
   .count()\
   .sort("count", ascending = False)
wordsDF.show()
```

```
+-----------------------------+
|value                        |
+-----------------------------+
|ppap                         |
|i have a pen                 |
|i have an apple              |
|ah apple pen                 |
|i have a pen                 |
|i have a pineapple           |
|ah pineapple pen             |
|ppap pen pineapple apple pen|
+-----------------------------+
```

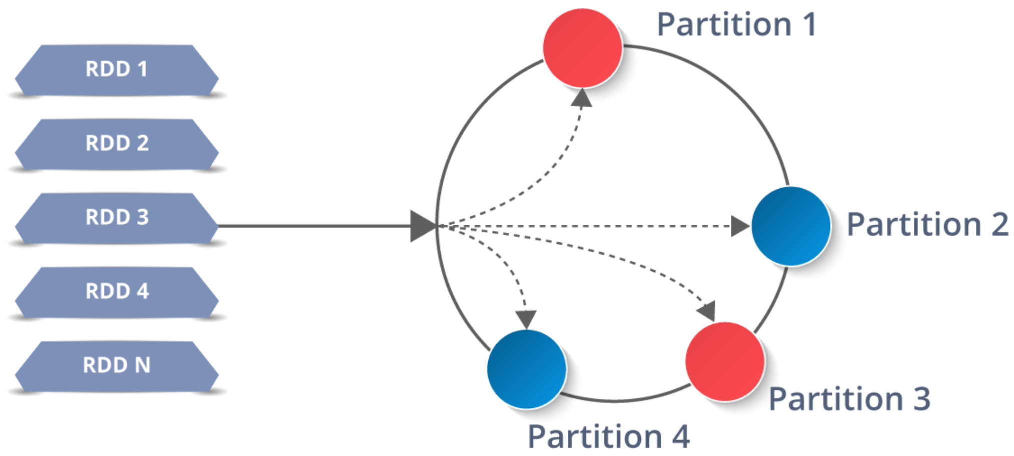
```
+---------+-----+
|     word|count|
+---------+-----+
|      pen|    6|
|     have|    4|
|        i|    4|
|    apple|    3|
|        a|    3|
|pineapple|    3|
|     ppap|    2|
|       ah|    2|
|       an|    1|
+---------+-----+
```

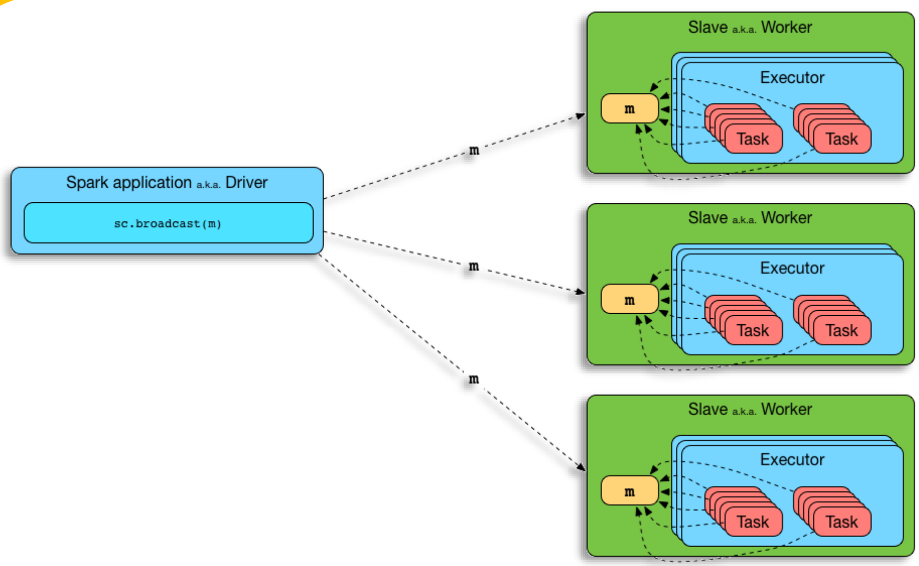
Resilient Distributed Datasets

What are low-level APIs?

Resilient Distributed Datasets (RDDs)

Manipulating distributed data





Broadcast variables and accumulators

Distributing and manipulating distributed shared variables

When to use low-level APIs?

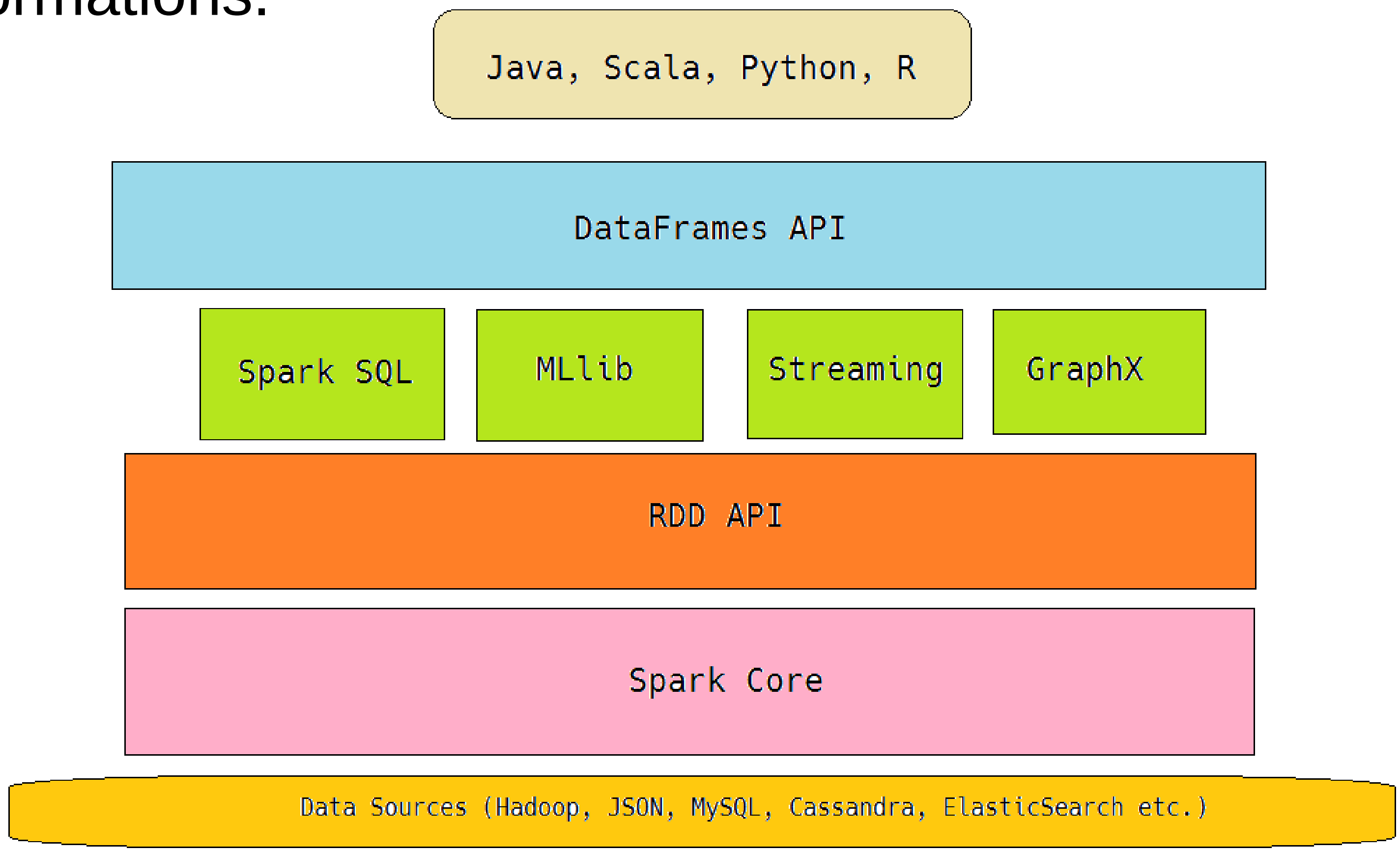
- Need some functionality not available in higher level APIs
  - E.g., very tight control over physical data placement across cluster
- Maintain some legacy codebase written using `RDDs`
- Do some custom shared variable manipulation

`DataFrame`** is more efficient, stable, and expressive than **`RDDs`

for the vast majority of use cases.

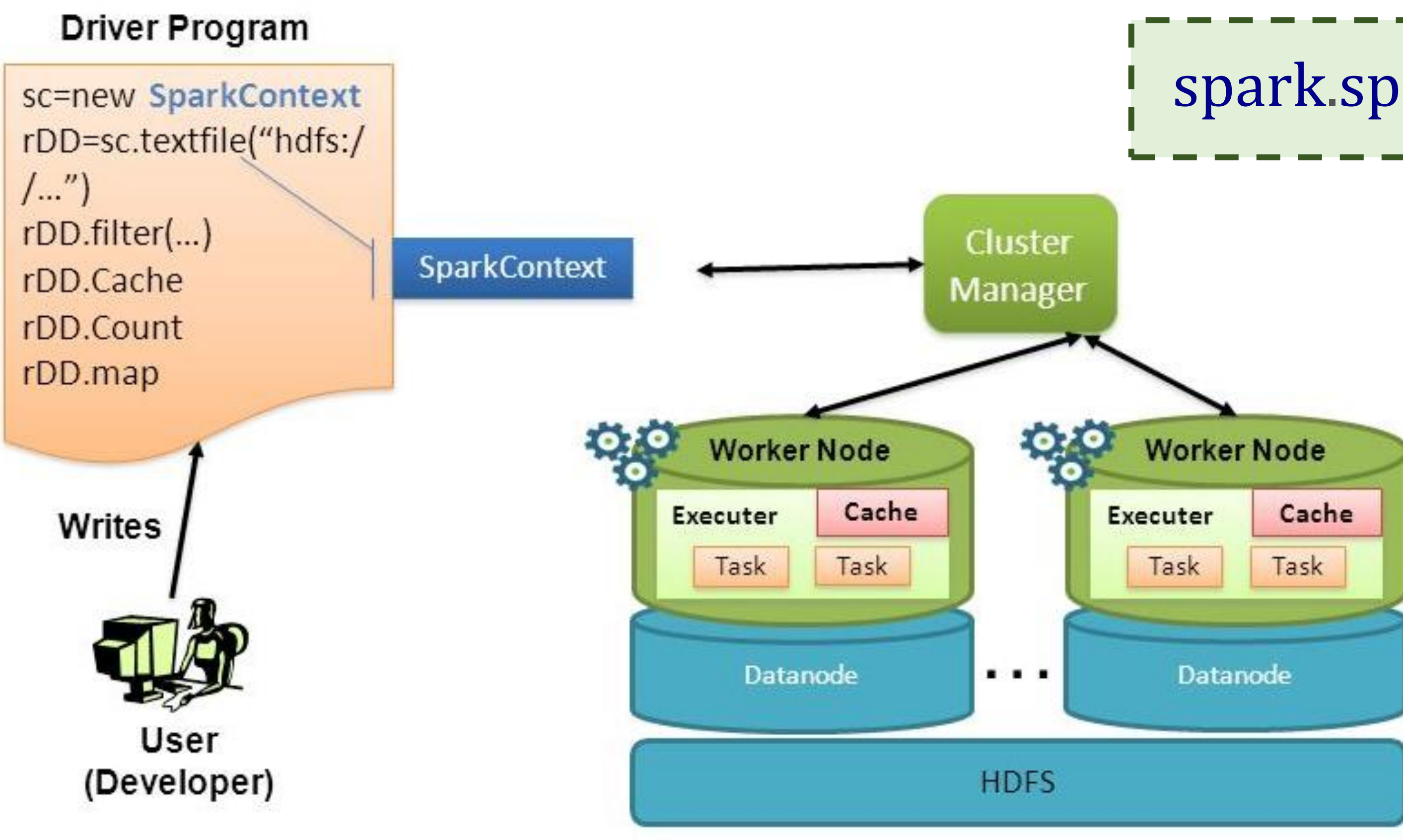
Why to understand low-level APIs?

- All Spark workloads compile down to fundamental primitives.
- A `DataFrame` transformation is converted into a set of `RDD` transformations.



How to use the low-level APIs?

- **`SparkContext`** is the entry point for low-level API functionality
  - It can be accessed through `SparkSession`, the tool for computation across a Spark cluster.



`spark.sparkContext`

Resilient Distributed Datasets

- The building blocks of any Spark application
- A layer of abstracted data over the distributed collection

- **Great power** but not without **potential issues**.

✓ These objects can store anything in any format.

✗ Every manipulation and interaction between values must be defined by hand.

✗ Optimizations are going to require much more manual work.

  - Spark does not understand the inner structure of the records as it does with the Structured APIs

Resilient Distributed Datasets

- Immutable partitioned collection distributed on different nodes
  - A partition is a basic unit of parallelism in Spark, distributing the workload to different nodes in the cluster .
  - The immutability helps to achieve fault tolerance and consistency.
- Lazy evaluation
  - The defined transformations do not get evaluated until an action is called so that they can be generally optimized in one go.
- Fault tolerant
  - `RDD` can be recomputed in case of any failure using DAG of transformations defined for that `RDD`.
- Multi-language support: Available in Python, R, Scala, and Java
- No in-built optimization engine
  - Programmers need to write their own code to minimize the memory usage and improve execution performance.

RDDs properties

- These properties determine all of Spark's ability to schedule and execute the user program.

Required	• A list of partitions
	• A function for computing each split
	• A list of dependencies on other **RDDs**
Optional	• A Partitioner for key-value **RDDs**
	• A list of preferred locations on which to compute each split (e.g., block locations of a HDFS file)

- Different kinds of `RDDs` implement their own versions of `RDD` properties, allowing to define new data sources.

Types of RDDs

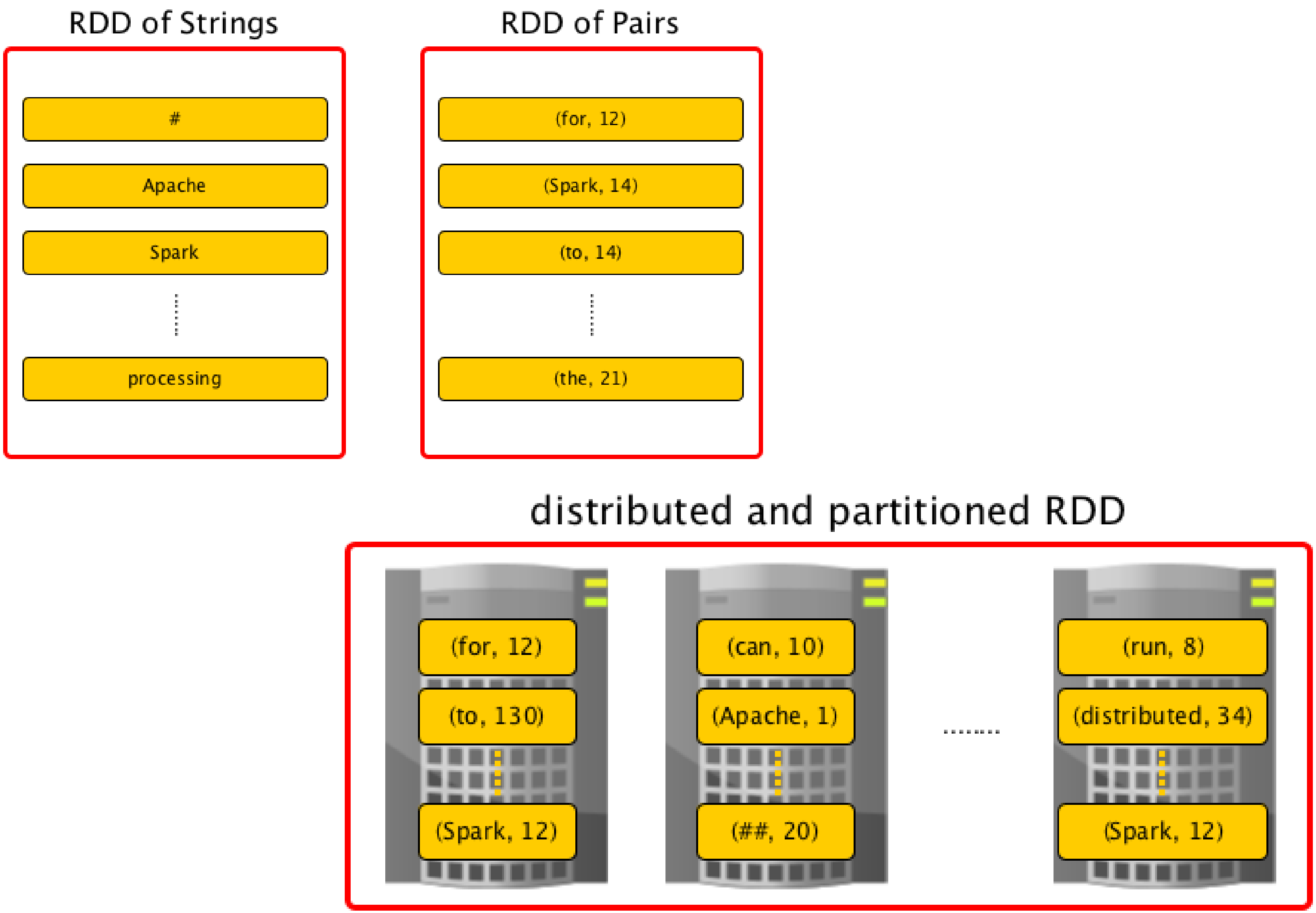
- There are lots of subclasses of `RDDs`, most of which are for `DataFrame` APIs to create optimized physical execution plans.

- Users will likely only be creating two types of `RDDs`

Generic RDD | **Key-value RDD**

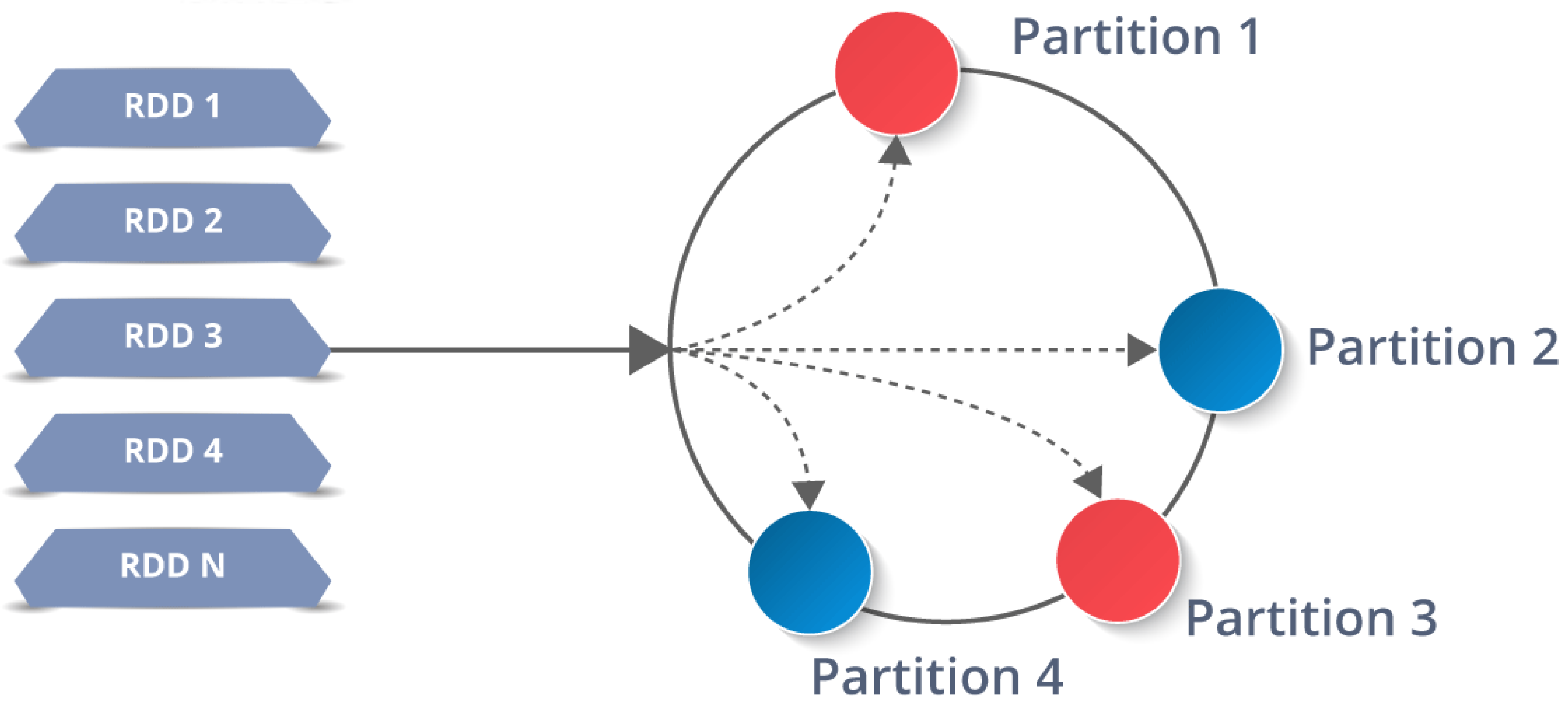
- Key-value `RDDs` have special operations and a concept of custom partitioning by key.
  - A correct customized `Partitioner` gives significant improvements on performance and stability.

Types of RDDs



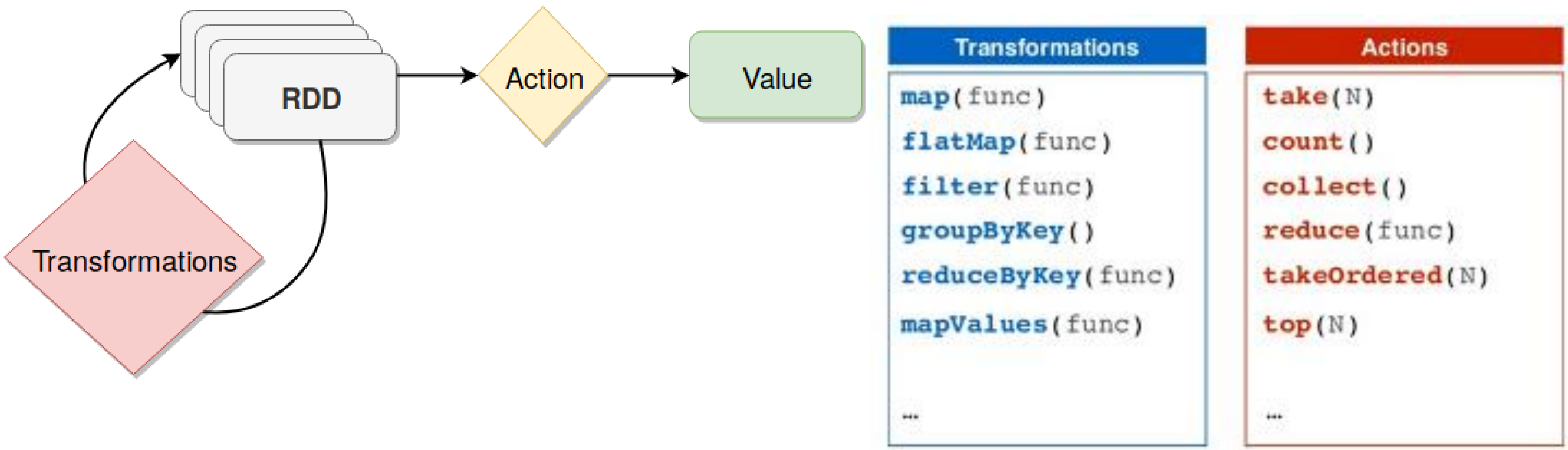
RDD: Logical partitions

- Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.



RDD: Transformations and Actions

- Transformations and actions in `RDDs` work similarly to those in `DataFrames` and `DataSets`.



- Transformations: lazily evaluated, actions: eagerly evaluated
  - Spark remembers the transformations applied to some base dataset and computes them when an action requires a result to be returned.

Basic RDD operations

Create an RDD from a DataFrame

- Create an RDD from an existing `DataFrame` or `Dataset`
  - Scala and Java: from `Dataset[T]` to `RDD[T]`
    - E.g., `Dataset[Long]` → `RDD[Long]`
  - Python: only from `DataFrames` to `RDDs` of type `Row`
  - To operate on this data, you need to convert the `Row` object to the correct data type or extract values out of it.

```
rdd = spark.range(10).rdd
# extract the value
rdd = rdd.map(lambda row: row[0])
```

- **`toDF`**: create a `DataFrame` from an `RDD`

```
df = rdd.toDF()
```

- **`name`**: give a name to the `RDD` to display in Spark UI

```
rdd.setName("myRdd")
rdd.name()
```

Create RDD from a local collection

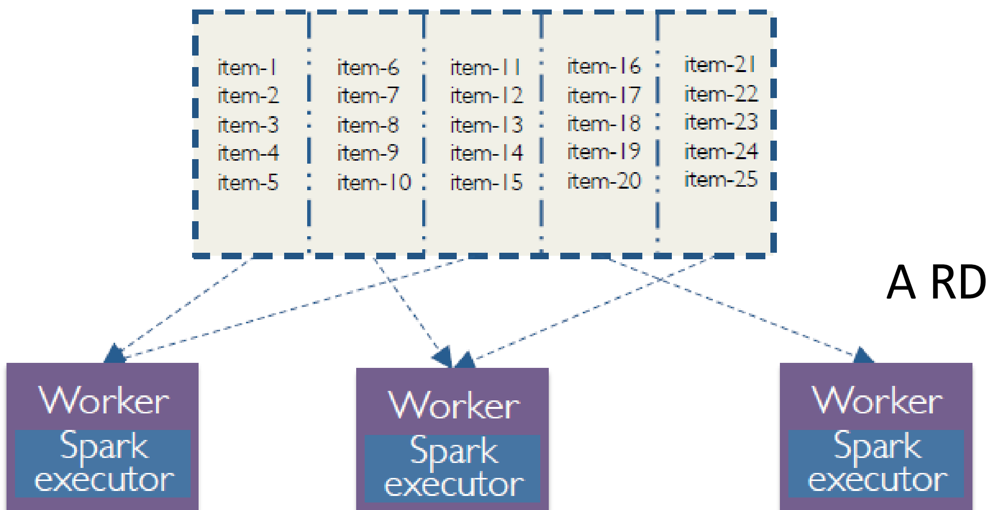
- **parallelize**: turn a single-node collection into a parallel one

```
# turn a string into a list of words, from which a new RDD is created
myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple".split(" ")
words = spark.sparkContext.parallelize(myCollection, 2)
```

- **glom**: show the content of every partition in an RDD

```
words.glom().collect()
```

```
[['Spark', 'The', 'Definitive', 'Guide', ':'],
 ['Big', 'Data', 'Processing', 'Made', 'Simple']]
```



A RDD is split into 5 partitions

Transformations: filter

- **`filter`**: work on each record individually and see which ones match some predicate function

```python
# define the filter function, which could be in pure Python
def startsWithS(individual):
        return individual.startswith("S")
# apply the defined function onto each record of the rdd
words.filter(lambda word: startsWithS(word))
```

```
# ['Spark', 'Simple']
```

- **`map`**: define functions completely inline using `lambda` syntax

```python
words2 = words.map(lambda word:\
        (word, word[0], word.startswith("S")))
```

```
[('Spark', 'S', True),
('The', 'T', False),
('Definitive', 'D', False),
……
('Made', 'M', False),
('Simple', 'S', True)]
```

Transformations: flatMap

- **`flatMap`**: map an element of source `RDD` to one or more elements of target `RDD`.
  - The object in consideration must be **`iterable`** to be expanded.

```
# perform on a collection of words
rdd = spark.sparkContext.parallelize(['Hello', 'Spark', 'Hello', 'Python'], 2)
rddFlatten = rdd.flatMap(lambda word: list(word))
```

```
['H’, 'e’,  'l’, 'l’, ‘o’,  ‘S’,  ‘p’,  ‘a’, ’r’,  ‘k’,  'H’,  'e’,  'l’,  'l’,  ‘o’,  ‘P’,  ‘y’,  't’,  'h’,  'o’,  'n']
```

```
# perform on a collection of lists
rdd = spark.sparkContext.parallelize([['Hello', 'Spark'], ['Hello', 'Python']], 2)
rddFlatten = rdd.flatMap(lambda word: list(word))
```

```
['Hello', 'Spark', 'Hello', 'Python’]
```

Transformations: Some other functions

- Many RDD transformations mirror the functionality found in the Structured APIs.
- **distinct**: remove duplicates from the RDD

```
words.distinct()
```

- **sortBy**: specify a function to extract a value from objects in the RDD and then sort based on that

```
# sort by the word's length in decreasing order
words.sortBy(lambda word: len(word) * -1).take(5)
```

```
['Definitive', 'Processing', 'Simple', 'Spark', 'Guide']
```

- **randomSplit**: split an RDD into an array of RDDs

```
# split the original RDD into two subsets with ratio 1:1
fiftyFiftySplit = words.randomSplit([0.5, 0.5])
```

Actions: reduce

- **`reduce`**: collect an `RDD` of any kind of value to one value.

```
# sum up the value in every entry of the RDD
spark.sparkContext.parallelize(range(1, 21)).reduce(lambda x, y: x + y)
```

210

```
# udf: return the word whose length is greater
def wordLengthReducer(leftWord, rightWord):
    if len(leftWord) > len(rightWord):
        return leftWord
    else:
        return rightWord
# return the word whose length is greatest (undeterministic)
words.reduce(wordLengthReducer)
```

'Processing'
or
'Definitive'

Actions: count and countByValue

- **`count`**: give the number of rows in the RDD

```
words.count()
```

```
10
```

  - **`countApprox`** returns a potentially incomplete result within a timeout, even if not all tasks have finished.
  - **`countApproxDistinct`**
- **`countByValue`**: give the number of values in an RDD.
  - The entire thing is loaded into the driver's memory
  - Use in scenarios that either the total number of rows or the number of distinct items is low
  - **`countByValueApprox`**

```
words.countByValue()
```

```
defaultdict(int,
    {'Spark': 1,
    'The': 1,
    'Definitive': 1,
    ......
    'Processing': 1,
    'Made': 1,
    'Simple': 1})
```

Actions: first, max, min and take

- **max** and **min**: return the maximum and minimum values, respectively

```
rdd = spark.sparkContext.parallelize([11, 1, 20, 19, 21], 2)
rdd.first()
rdd.min()
rdd.max()
```

```
first: 11, min: 1, max: 21
```

- **first**: return the first value in the dataset
- **takeSample**: specify a fixed-size random sample

```
# sample 6 elements with replacement
withReplacement = True
numberToTake = 6
randomSeed = 100
words.takeSample(withReplacement, numberToTake, randomSeed)
```

```
['Data', 'Definitive', 'Data', 'The', 'Definitive', 'Spark’]
or ['The', ':', 'Simple', 'Spark', 'Data', 'Big']
```

Actions: take, takeOrdered, and top

- `take` and its variants take several values from the RDD.
  - They first scan one partition and then use the results to estimate the number of additional partitions needed to satisfy the limit.

- `take`: grab the first N items from the RDD
- `takeOrdered`: grab the first N items from the RDD, following the lexicographic order
- `top`: grab the first N items from the RDD, following the *reversed* lexicographic order

```
words.take(5)
words.takeOrdered(5)
words.top(5)
```

```
['Spark', 'The', 'Definitive', 'Guide', ':’]
[':', 'Big', 'Data', 'Definitive', 'Guide']
['The', 'Spark', 'Simple', 'Processing', 'Made']
```

Saving RDD to files

- A RDD cannot be conventionally “saved” to a data source.
  - It must iterate over the partitions to save the contents of each partition to some external database.
- Write to a plain-text file

```
words.saveAsTextFile("file:/tmp/bookTitle")
```

  - To set a compression codec: import the proper codec from Hadoop – org.apache.hadoop.io.compress library

```
codec = "org.apache.hadoop.io.compress.GzipCodec"
words.saveAsTextFile("file:/tmp/bookTitle“, codec)
```

- Write to a sequence file

```
words.saveAsObjectFile("/tmp/my/sequenceFilePath")
```

- A variety of different Hadoop file formats are also supported.

Cache and Persist

- The same principles for `RDDs`, `DataFrames` and `Datasets`.
- **`cache`**: persist the RDD with the default storage level (MEMORY_ONLY)

```
words.cache()
```

```
words.getStorageLevel()
```

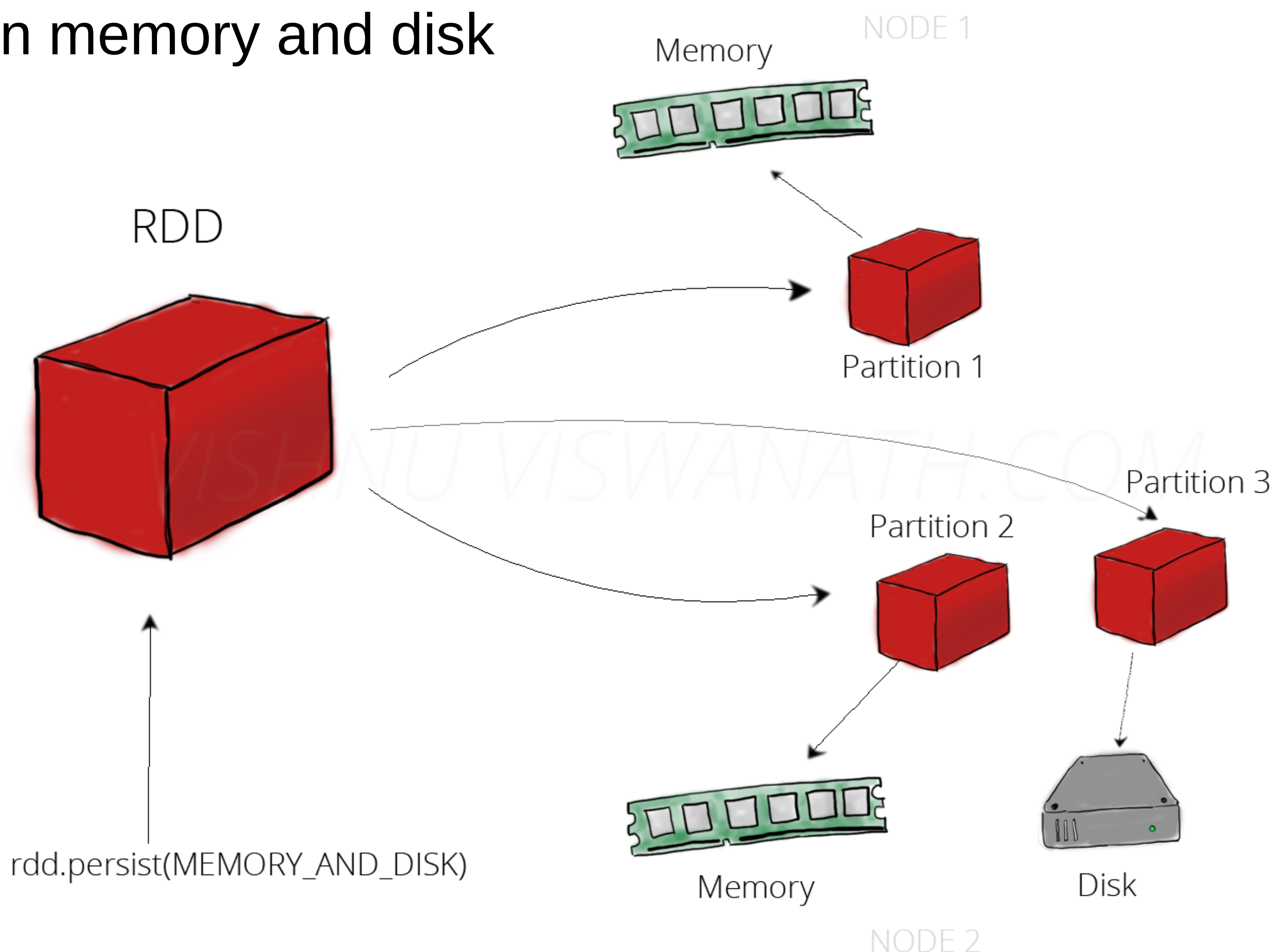
```
words.persist()
```

```
StorageLevel(False, False, False, False, 1)
```

- **`persist`**: set the RDD's storage level to persist its values across operations after the first time it is computed
- **`getStorageLevel`**: get the information of the storage level
  - pyspark.StorageLevel(useDisk, useMemory, useOffHeap, deserialized, replication=1)

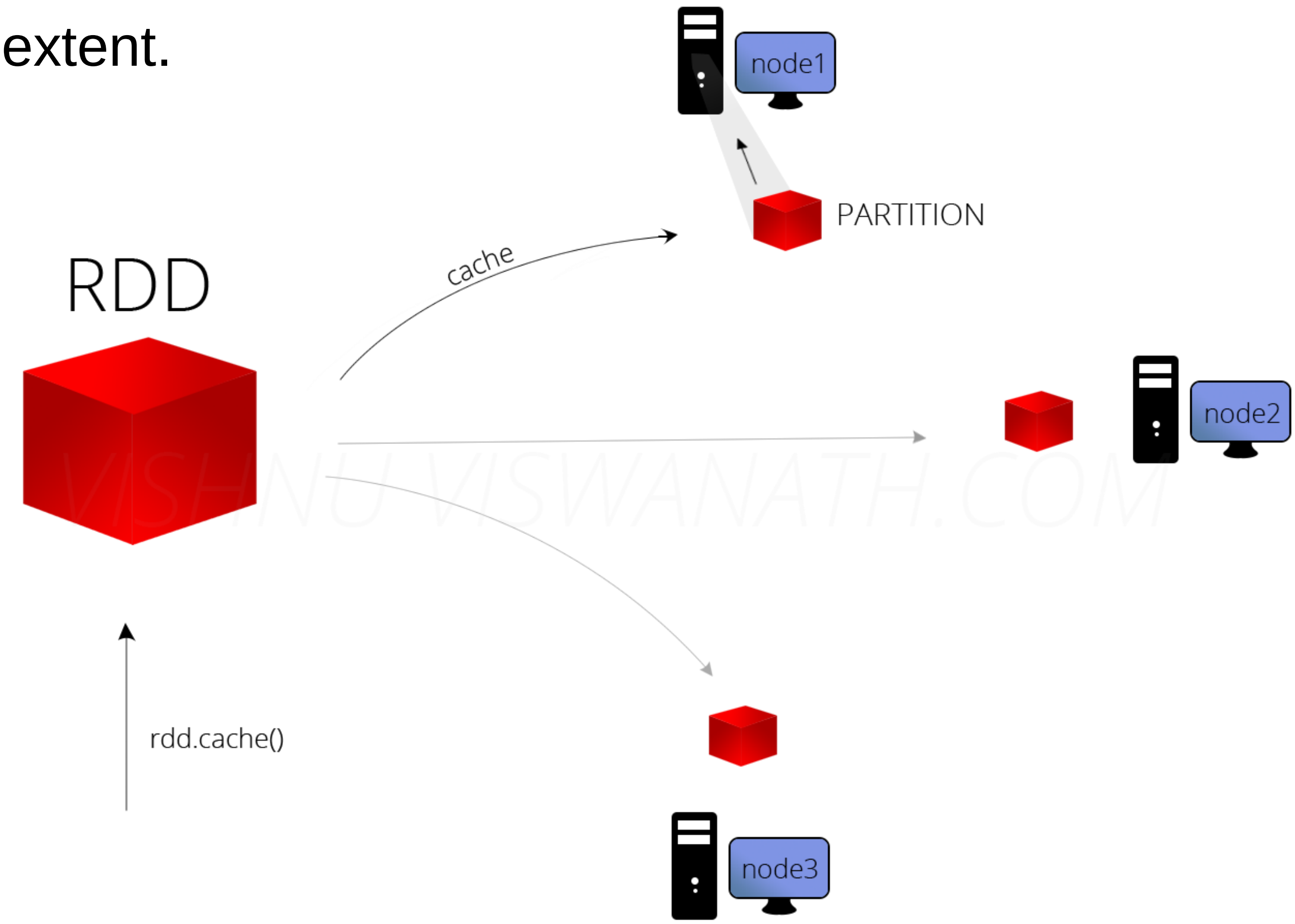
Cache and Persist

- Persist in memory and disk



Cache and Persist

- Caching can improve the performance of an application to a great extent.



Checkpointing

- Save an **`RDD`** to disk → future references point to those intermediate partitions rather than recomputing the **`RDD`** from its original source

```
# set a checkpoint of the rdd to the designated location
spark.sparkContext.setCheckpointDir("/some/path/for/checkpointing")
words.checkpoint()
```

- Helpful optimization
- Similar to caching except that it is stored only in disk
- Feature not available in the DataFrame API

Finally, our Word Count!

```
linesRdd = sc.textFile(“ppap.txt")
wordsRdd = linesRdd.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)\
    .sortBy(lambda pair:-1*pair[1])
wordsRdd.collect()
```

VS.

```
linesDF = spark.read.text("ppap.txt")
wordsDF = linesDF.withColumn("word", f.explode(f.split(f.col("value"), " ")))\
    .groupBy("word")\
    .count()\
    .sort("count", ascending = False)
wordsDF.show()
```

Advanced RDD operations

Key-value RDDs

- There are many cases that require data in key–value format.
- `<some-operation>ByKey` can only perform on `PairRDD`.
- A key–value structure can be achieved by **mapping**.

```
# map each entry in the RDD to a tuple (key, 1)
words.map(lambda word: (word.lower(), 1))
```

```
[('spark', 1), ('the', 1), ('definitive', 1), ('guide', 1), ..., ('made', 1), ('simple', 1)]
```

- **keyBy** provides similar results by specifying a function that creates the key from a current value.

```
# create a key-value RDD by keying by the first letter in the word
keyword = words.keyBy(lambda word: word.lower()[0])
```

```
[('s', 'Spark'), ('t', 'The'), ('d', 'Definitive'), ..., ('m', 'Made'), ('s', 'Simple')]
```

Transformations: keys, values, lookup

- `keys` and `values` extract the keys and values, respectively

```
keyword.values()
```

```
['Spark', 'The', 'Definitive', 'Guide', ':', 'Big', 'Data', 'Processing', 'Made', 'Simple']
```

```
keyword.keys()
```

```
['s', 't', 'd', 'g', ':', 'b', 'd', 'p', 'm', 's']
```

- `lookup`: provide the result for a particular key in the RDD
  - There is no enforcement mechanism with respect to there being only one key for each input

```
keyword.lookup('s')
```

```
['Spark', 'Simple']
```

Transformations: Map over values

- In a tuple, the first element is considered as a key and the second as the corresponding value.
- **`mapValues`**: explicitly map over the values and ignore the individual keys

```
keyword.mapValues(lambda word: word.upper())
```

```
[('s', 'SPARK'), ('t', 'THE'), ('d', 'DEFINITIVE'),..., ('m', 'MADE'), ('s', 'SIMPLE')]
```

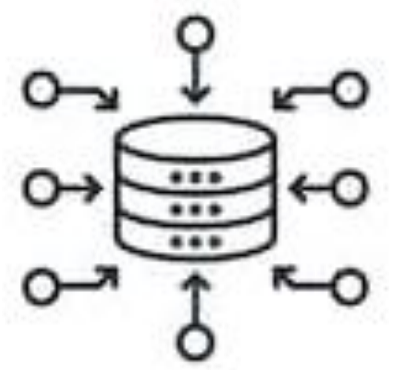
- **`flatMapValues`**: expand the number of rows so that each row represents a character.

```
keyword.flatMapValues(lambda word: word.upper())
```

```
[('s', 'S'), ('s', 'P'), ('s', 'A'), ('s', 'R'), ('s', 'K'),..., (‘s', 'P'), ('s', 'L'), (‘s', 'E')]
```

Aggregations functions

- Aggregations can be done on plain `RDDs` or on `PairRDDs`, depending on the method being used.



- Let's define some preliminaries

```
chars = words.flatMap(lambda word: word.lower())
KVcharacters = chars.map(lambda letter: (letter, 1))

def maxFunc(left, right):
        return max(left, right)
def addFunc(left, right):
        return left + right

nums = spark.sparkContext.parallelize(range(1,31), 5)
```

Aggregations by keys

- **countByKey**: give the number of elements for each key, collecting the results to a local Map.

```
KVcharacters.countByKey()
```

```
defaultdict(int,
    {':': 1, 'a': 4, 'b': 1, ...,
    's':4, 't': 3, 'u': 1, 'v': 1})
```

- There are several ways to create key–value PairRDD.
- The implementation is important for job stability.
- **groupByKey** and **reduceByKey** are similar functions.

```
KVcharacters.groupByKey().map(lambda row: (row[0], reduce(addFunc, row[1])))
```

```
KVcharacters.reduceByKey(addFunc)
```

```
[('s', 4), ('p', 3), ('r', 2), ('h', 1), ..., (':', 1), ('o', 1), ('m', 2)]
```

groupByKey vs. reduceByKey

- `groupByKey` is a wrong approach for most cases.
  - Each executor must hold all values for a given key in memory before applying the function to them.
  - With massive key skew, some partitions might be overloaded.
  - *When to use? Each key has consistent value sizes and fits in the memory of a given executor*
- `reduceByKey` is preferred for additive use cases.
  - Within each partition, everything in memory not required
  - No incurred shuffle during this operation, everything at each worker individually before performing a final reduce
  - *When to use? The workload is associative*
  - *When not to use? The order matters*

Aggregations: aggregate

- **`aggregate`** requires a function that aggregate within partitions and another for across partitions.
  - It performs the final aggregation on the driver → failed if the results from the executors are too large.

```
# sum the maximum values of every partition
nums.aggregate(0, maxFunc, addFunc)
```

90

- **`treeAggregate`** “pushes down” some sub-aggregations before performing the final aggregation on the driver.
  - It helps to ensure that the driver does not run out of memory in the aggregation process.

```
depth = 3
nums.treeAggregate(0, maxFunc, addFunc, depth)
```

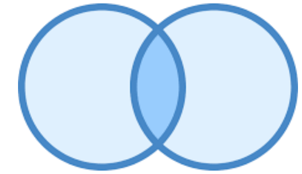
Aggregations: combineByKey

- A combiner operates on a given key and merges the values according to some function
- It then merges different outputs of combiners for result.

```
# user-defined functions for the combiner
def valToCombiner(value):
    return [value]
def mergeValuesFunc(vals, valToAppend):
    vals.append(valToAppend)
    return vals
def mergeCombinerFunc(vals1, vals2):
    return vals1 + vals2

# trigger the aggregation with given udf
outputPartitions = 6
KVcharacters.combineByKey(valToCombiner, mergeValuesFunc,\
            mergeCombinerFunc, outputPartitions)
```

```
[  [('s', [1, 1, 1, 1]), ('d', [1, 1, 1, 1]), …
   [('v', [1]), (':', [1])],
   [('p', [1, 1, 1]), ('r', [1, 1]), ('c', [1])],
   [('k', [1]), ('t', [1, 1, 1]), …
   [('h', [1]), ('i', [1, 1, 1, 1, 1, 1, 1]), …
   [('a', [1, 1, 1, 1]), …
]
```

Aggregations: inner join



- **`join`**: perform an inner join.
  - The number of output partitions can be specified by outputPartitions.

```
# sum the maximum values of every partition
x = spark.sparkContext.parallelize([("a", 1), ("b", 4), ("c", 6)])
y = spark.sparkContext.parallelize([("a", 2), ("a", 3), ("b", 5)])
x.join(y).collect()
```

```
[('b', (4, 5)), ('a', (1, 2)), ('a', (1, 3))]
```

- Other joins all follow the same basic format
  - fullOuterJoin, leftOuterJoin, rightOuterJoin
  - cartesian (This is very dangerous! It does not accept a join key and can have a massive output.)
- Basic format: the two RDDs to join and (optionally) either the number of output partitions or the customer partition function

Aggregations: cogroup and zip

- **`cogroup`**: result in a group with the given key on one side, and all the relevant values on the other side.

```
# perform cogroup on two given RDDs
x = spark.sparkContext.parallelize([("foo", 1), ("bar", 4)])
y = spark.sparkContext.parallelize([("foo", -1)])
grp = x.cogroup(y)
# printing the content of a co-grouped RDD is not straightforward
grp.mapValues(lambda val: [i for e in val for i in e]).collect()
```

```
[('foo', [1, -1]), ('bar', [4])]
```

- **`zip`**: combine two `RDDs` of the same length
  - The two RDDs must further have the same number of partitions

```
numRange = sc.parallelize(range(10), 2)
words.zip(numRange).collect()
```

```
[('Spark', 0),  ('The', 1),  ('Definitive', 2),…, ('Processing', 7),  ('Made', 8), ('Simple', 9)]
```

Controlling partitions

- Control over how data is exactly physically distributed across the cluster
  - Same as in the Structured APIs, but additionally specify a partitioning function (formally a custom Partitioner)
- **`coalesce`**: collapse partitions on the same worker in order to avoid a shuffle of the data when repartitioning

```
# this gives us 1 partition
words.coalesce(1).getNumPartitions()
```

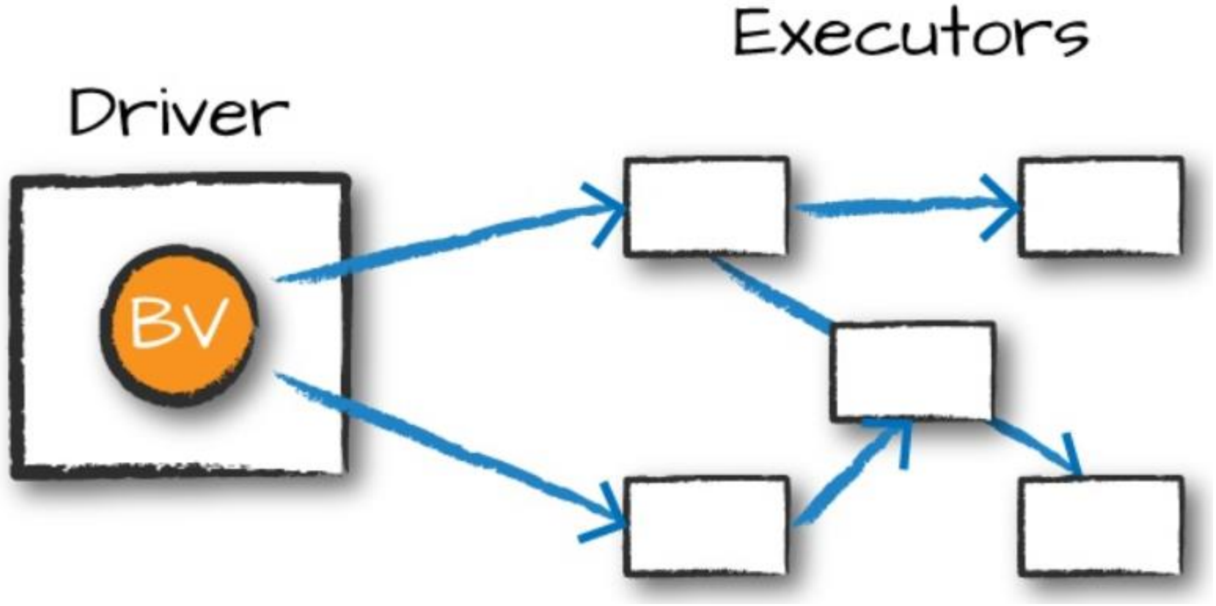
```
# this gives us 1 partition
words.repartition(10)
```

- **`repartition`**: partition the data up or down but performs a shuffle across nodes in the process.

Distributed shared variables

Broadcast variables

- Shared and immutable variables that are cached on every machine in the cluster
  - E.g., pass around a large lookup table that fits in memory on the executors and use that in a function



Broadcast variables

- Share immutable values efficiently around the cluster without encapsulating it in a function closure
  - Depend on the amount of data and the number of executors
  - For very small data (low KBs) on small clusters, it might not be
- Broadcasting can be used in the context of an RDD, a UDF or a Dataset and achieve the same result.

Broadcast variables: An example

- Consider the `RDD` of words, **`words`**, shown in previous example.
- Let's supplement the list of words with other information (right join)
  - This may be many KBs, MBs, or potentially even GBs in size.

```
supplementalData = {"Spark":1000, "Definitive":200,"Big":-300, "Simple":100}
```

- Broadcast the structure across Spark and reference it
  - This value is immutable and is lazily replicated across all nodes when trigger an

```
suppBroadcast = spark.sparkContext.broadcast(supplementalData)}
```

- Reference this variable via

```
suppBroadcast.value
```

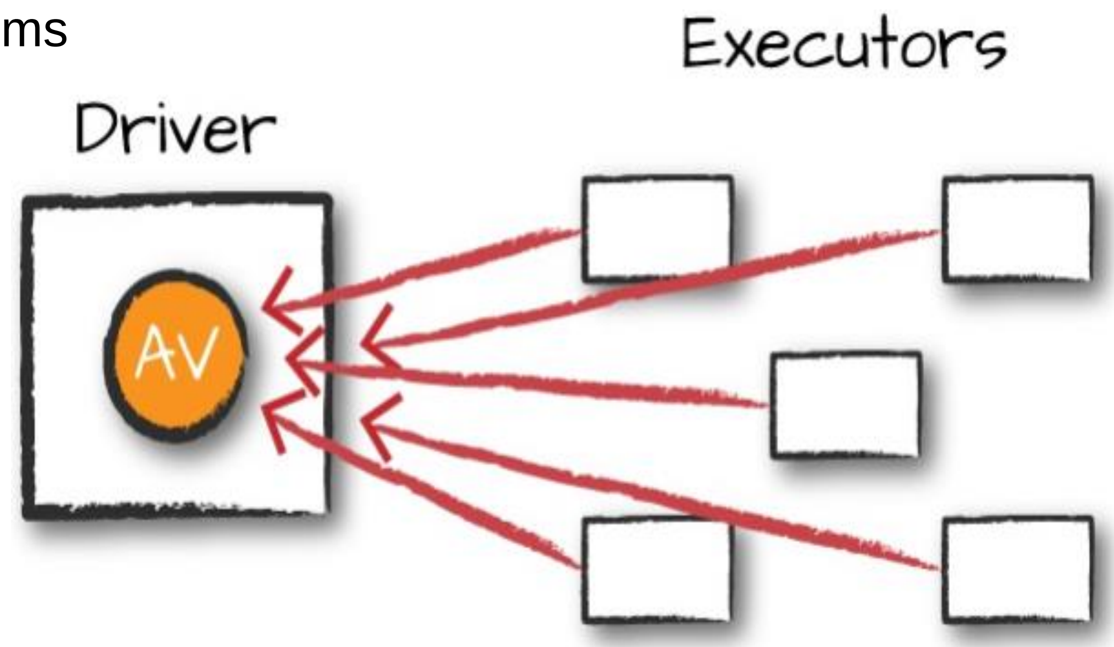
- Transform the given RDD using this value

```
words.map(lambda word: (word,\
      suppBroadcast.value.get(word, 0)))\
      .sortBy(lambda wordPair: wordPair[1])
```

```
[('Big', -300),
('The', 0),
...
('Definitive', 200),
('Spark', 1000)]
```

Accumulators

- A mutable variable that a Spark cluster can safely update on a per-row basis
  - E.g., debugging, low-level aggregation, implement counters (as in MapReduce) or sums



- Spark natively supports accumulators of numeric types, and programmers can add support for new types.

Accumulators: An example

- Read the flight data stored in a Parquet-formatted file

```
flights = spark.read.parquet("/data/flight-data/parquet/2010-summary.parquet")
```

- Create an accumulator to count the number of flights to or from China

```
accChina = spark.sparkContext.accumulator(0)
```

- Define how to add to the accumulator

```
def accChinaFunc(flight_row):
    destination = flight_row["DEST_COUNTRY_NAME"]
    origin = flight_row["ORIGIN_COUNTRY_NAME"]
    if destination == "China":
        accChina.add(flight_row["count"])
    if origin == "China":
        accChina.add(flight_row["count"])
flights.foreach(lambda flight_row: accChinaFunc(flight_row))
```

Accumulators: An example

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.5 s	0.5 s	0.5 s	0.5 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks
driver	10.172.238.229:44026	0.5 s	1	0	1

Accumulators

Accumulable	Value
China	953

Tasks (1)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	210	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/01/17 21:33:27	0.5 s		China: 953	


...the end.