STRUCTURED APIs

Presented by: Le Ngoc Thanh

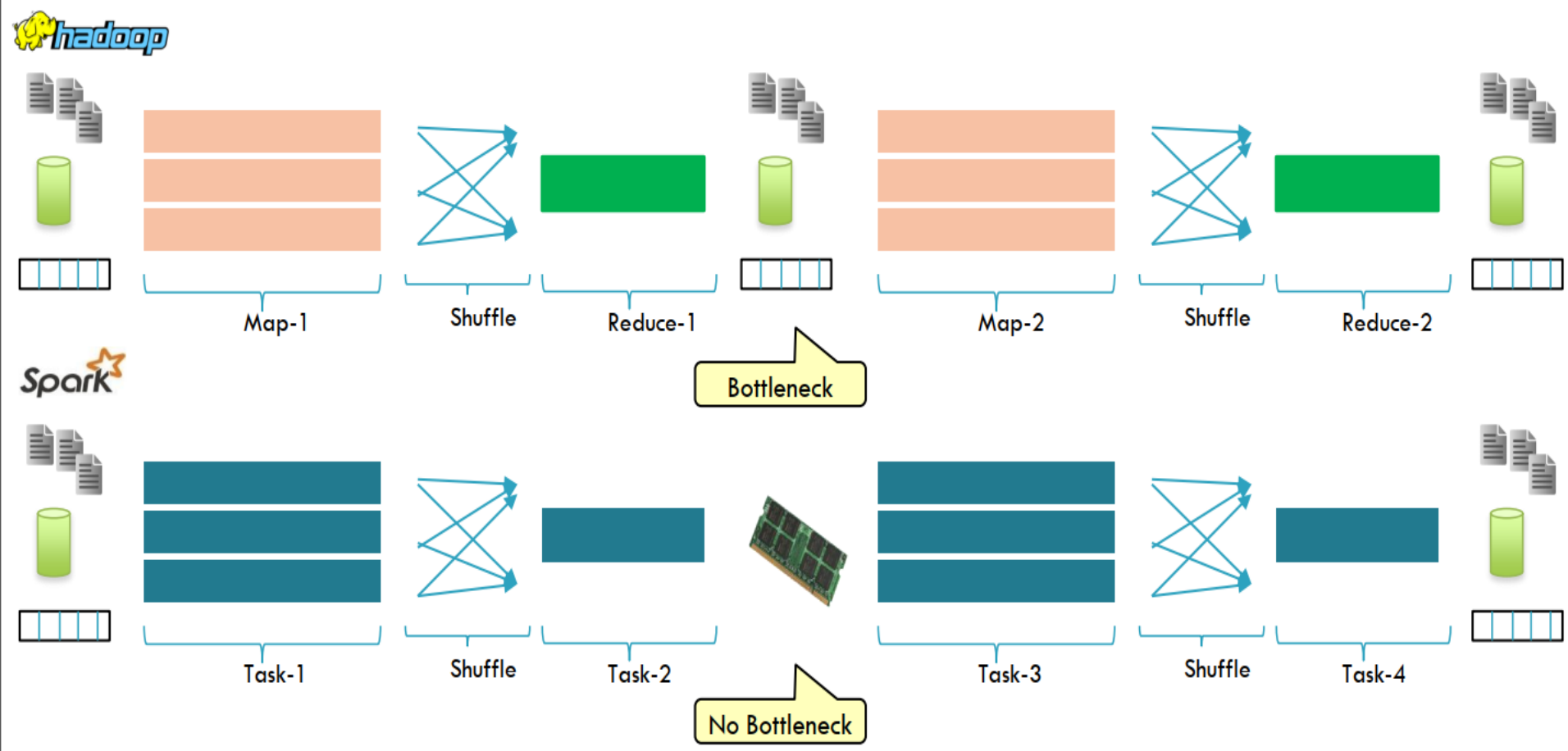
Outline

- **Review Spark Architecture**
  - Overall Architecture
  - RDD
  - DAG
  - Stage, Task
  - Example
- DataFrames and Datasets
- Structured API Execution
- Basic Structured Operations

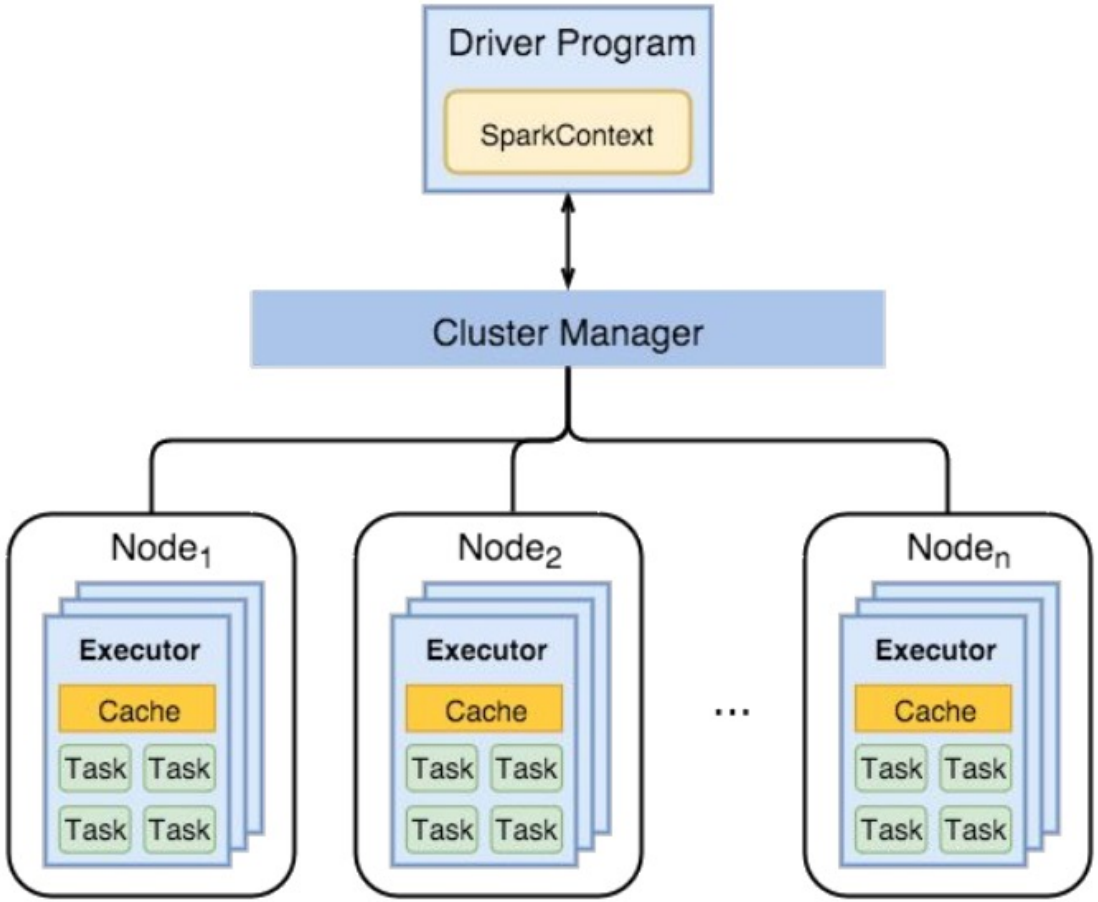
@lnthanh

MapReduce vs Spark

fit@hcmus



Spark Architecture

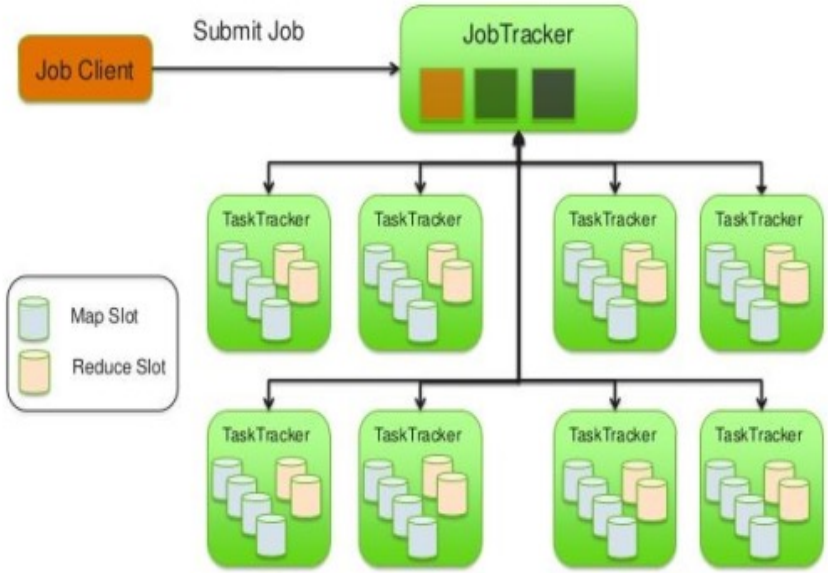


- Spark Driver
  - separate process to execute user applications
  - creates *SparkContext* to schedule jobs execution and negotiate with cluster manager
- Executors
  - run tasks scheduled by driver
  - store computation results in memory, on disk or off-heap
  - interact with storage systems
- Cluster Manager
  - Mesos
  - YARN
  - Spark Standalone

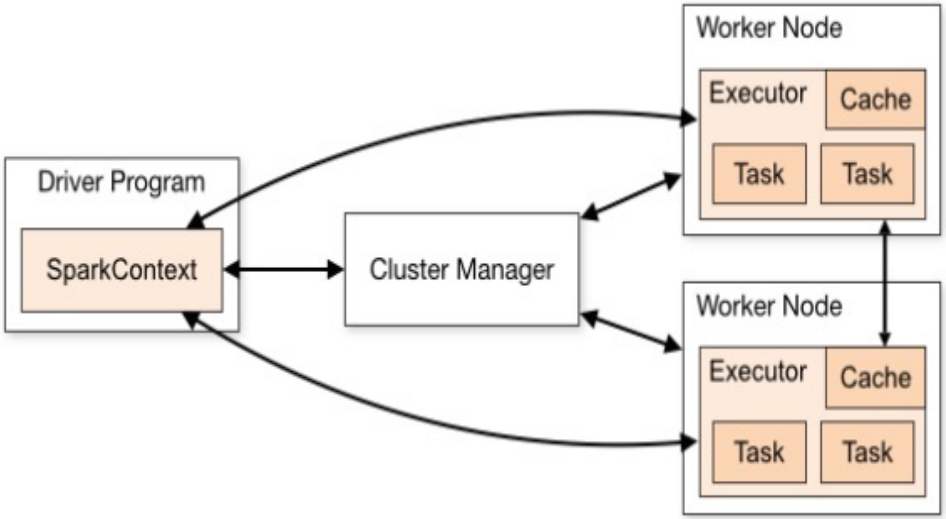
MapReduce vs Spark

fit@hcmus

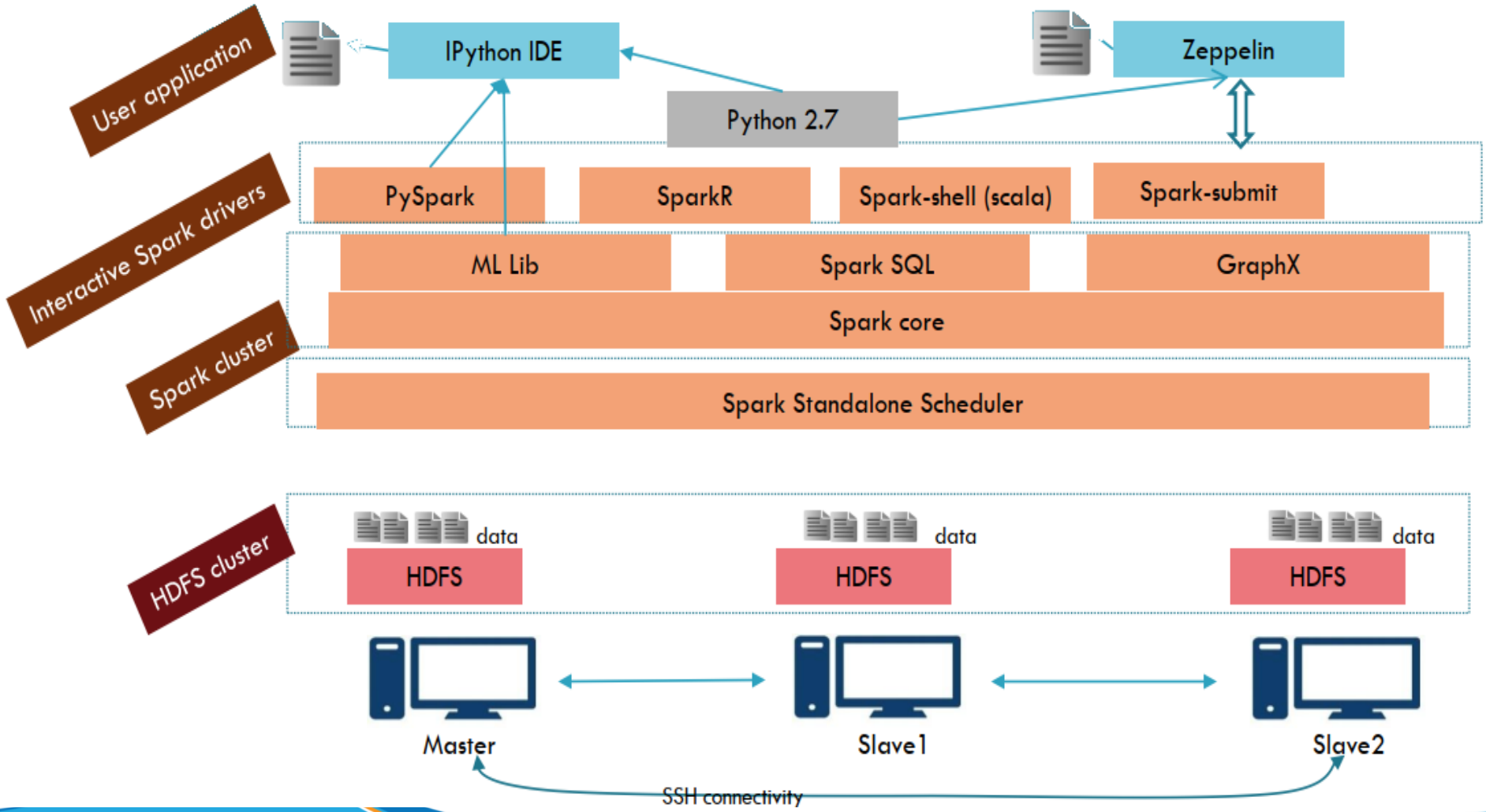
Apache Hadoop v1



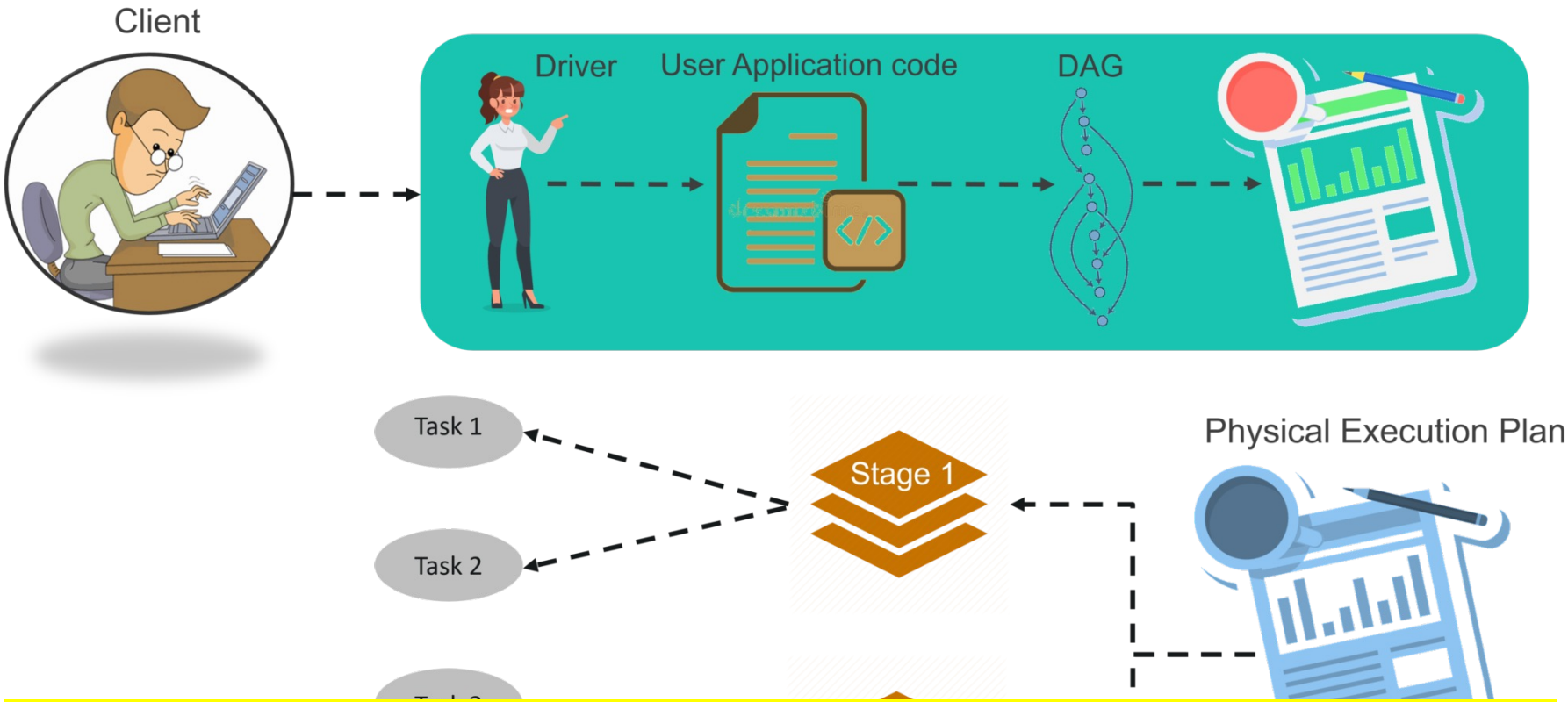
Apache Spark



Spark

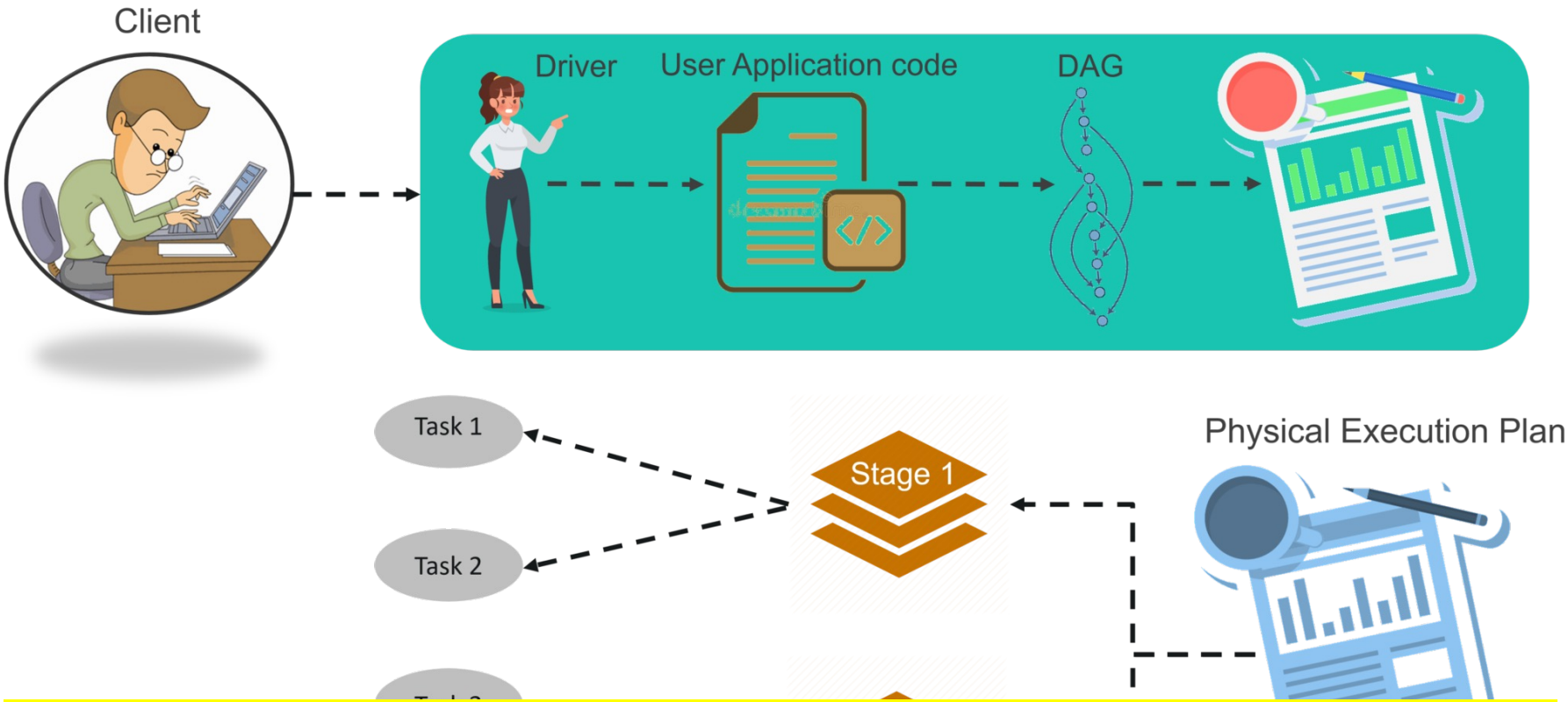

User application
IPython IDE
Zeppelin
Python 2.7
Interactive Spark drivers
PySpark
SparkR
Spark-shell (scala)
Spark-submit
Spark cluster
ML Lib
Spark SQL
GraphX
Spark core
Spark Standalone Scheduler
HDFS cluster
data
HDFS
data
HDFS
data
HDFS
Master
Slave1
Slave2
SSH connectivity

The architecture of Spark



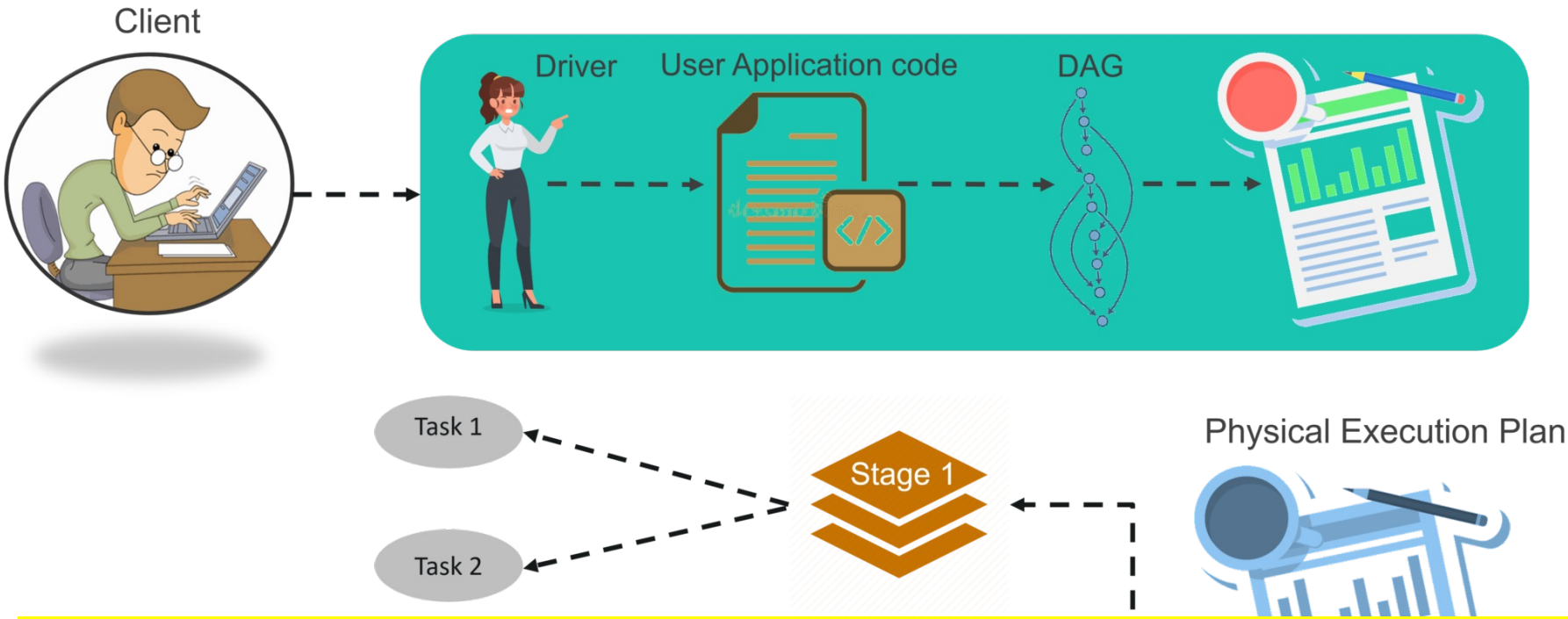
STEP 1:
- The client submits spark user application code.
- Next, the driver implicitly converts user code that contains transformations and actions into a logically directed acyclic graph called DAG.
- At this stage, it also performs optimizations such as pipelining transformations.

The architecture of Spark



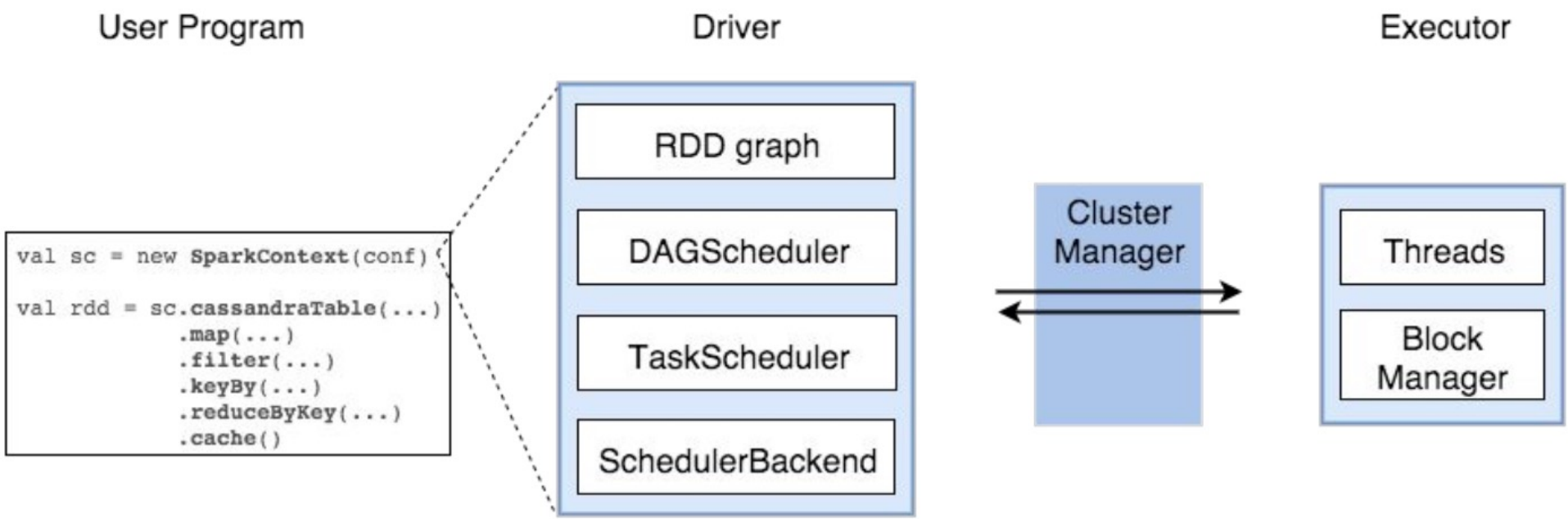
STEP 2:
- It converts DAG into physical execution plan with many stages.
- Next, it creates physical execution units called tasks under each stage.
- Then the tasks are bundled and sent to the cluster.

The architecture of Spark

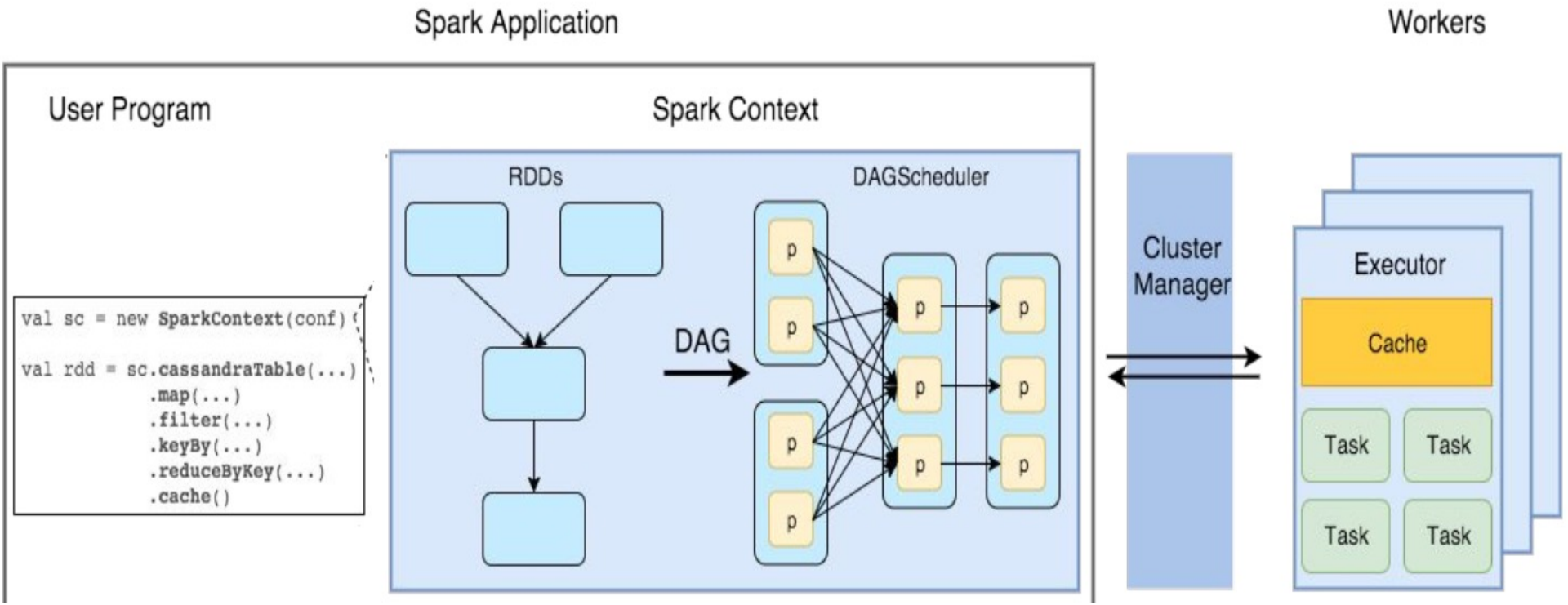


STEP 3:
- The driver talks to the cluster manager and negotiates the resources.
- Cluster manager launches executors in worker nodes on behalf of the driver.
- At this point, the driver will send the tasks to the executors based on data placement. When executors start, they register themselves with drivers. So, the driver will have a complete view of executors that are executing the task.

Spark Components



Spark Components



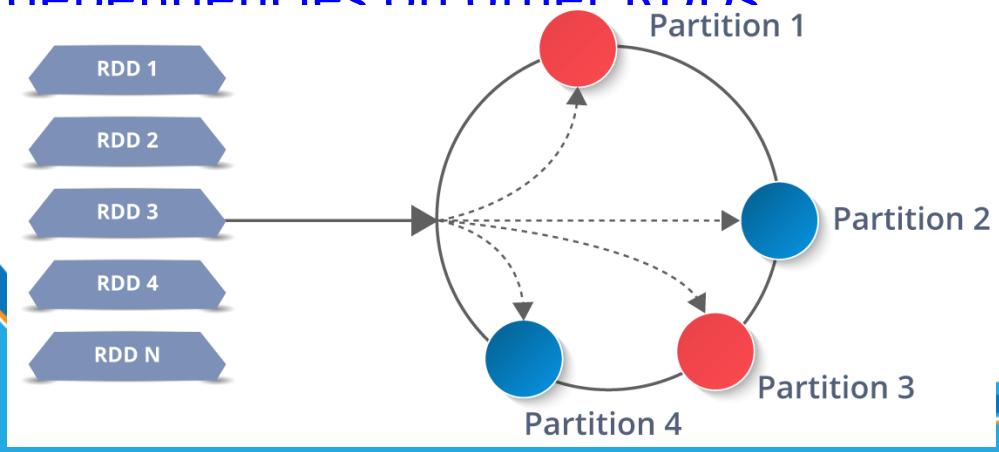
Outline

- **Review Spark Architecture**
  - Overall Architecture
  - RDD
  - DAG
  - Stage, Task
  - Example
- DataFrames and Datasets
- Structured API Execution
- Basic Structured Operations

@lnthanh

Resilient Distributed Dataset (RDD)

- **Resilient Distributed Dataset (RDD)** represents an immutable, partitioned collection of elements that can be operated on in parallel.
- Each RDD is characterized by the following main properties
  - A list of partitions
  - A function for computing each split
  - A list of dependencies on other RDDs

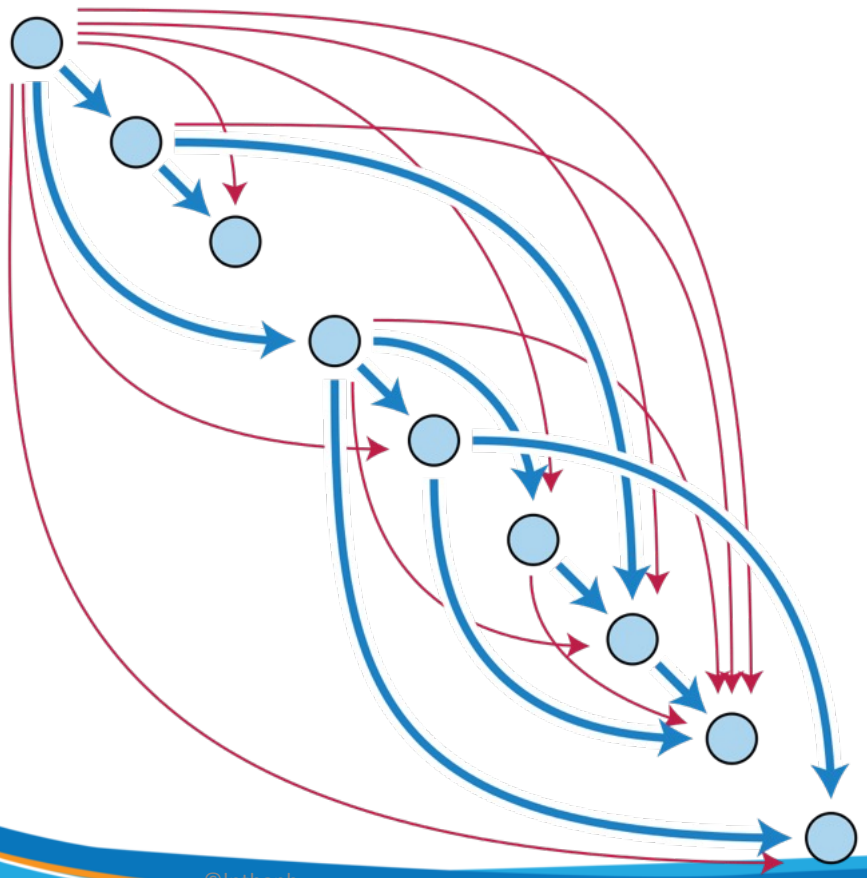


RDD

- Provides API for
  - manipulating the collection of elements (transformations and materialization)
  - persisting intermediate results in memory for later reuse
  - controlling partitioning to optimize data placement
- Can be created through deterministic operation
  - from storage (distributed file system, database, plain file)
  - from another RDD
- Stores information about parent RDDs
  - for execution optimization and operations pipelining
  - to recompute the data in case of failure

Spark applications: DAG

- DAG is a graph denoting the sequence of operations that are performed on the target RDD

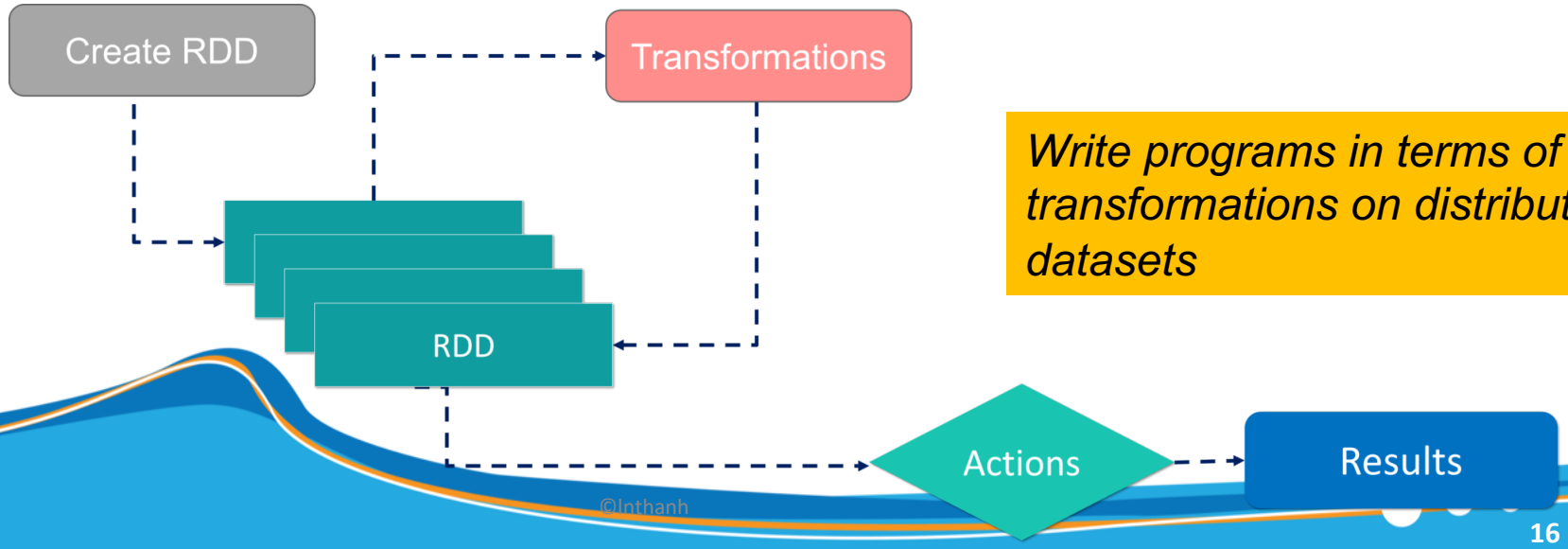


@lnthanh

Life of a Spark Program

fit@hcmus

1) Create some input RDDs from external data or parallelize a collection in your driver program.
2) Lazily ***transform*** them to define new RDDs using transformations like filter() or map()
3) Ask Spark to cache() any intermediate RDDs that will need to be reused.
4) Launch ***actions*** such as count() and collect() to kick off a parallel computation, which is then optimized and executed by Spark



Write programs in terms of transformations on distributed datasets

@lnthanh

RDD Operations

fit@hcmus

- **Transformations**
  - *apply* user function to every element in a partition (or to the whole partition)
  - *apply* aggregation function to the whole dataset (groupBy, sortBy)
  - *introduce* dependencies between RDDs to form DAG
  - provide functionality for repartitioning (repartition, partitionBy)
- **Actions**
  - trigger job execution
  - used to materialize computation results
- **Extra**: persistence
  - explicitly store RDDs in memory, on disk or off-heap (cache, persist)
  - checkpointing for truncating RDD lineage

@lnthanh

Example Transformations

fit@hcmus

map()	intersection()	cartesion()
flatMap()	distinct()	pipe()
filter()	groupByKey()	coalesce()
mapPartitions()	reduceByKey()	repartition()
mapPartitionsWithIndex()	sortByKey()	partitionBy()
sample()	join()	...
union()	cogroup()	...

Example Actions

fit@hcmus

```
reduce()
collect()
count()
first()
take()
takeSample()
saveToCassandra()
```

```
takeOrdered()
saveAsTextFile()
saveAsSequenceFile()
saveAsObjectFile()
countByKey()
foreach()
...
```

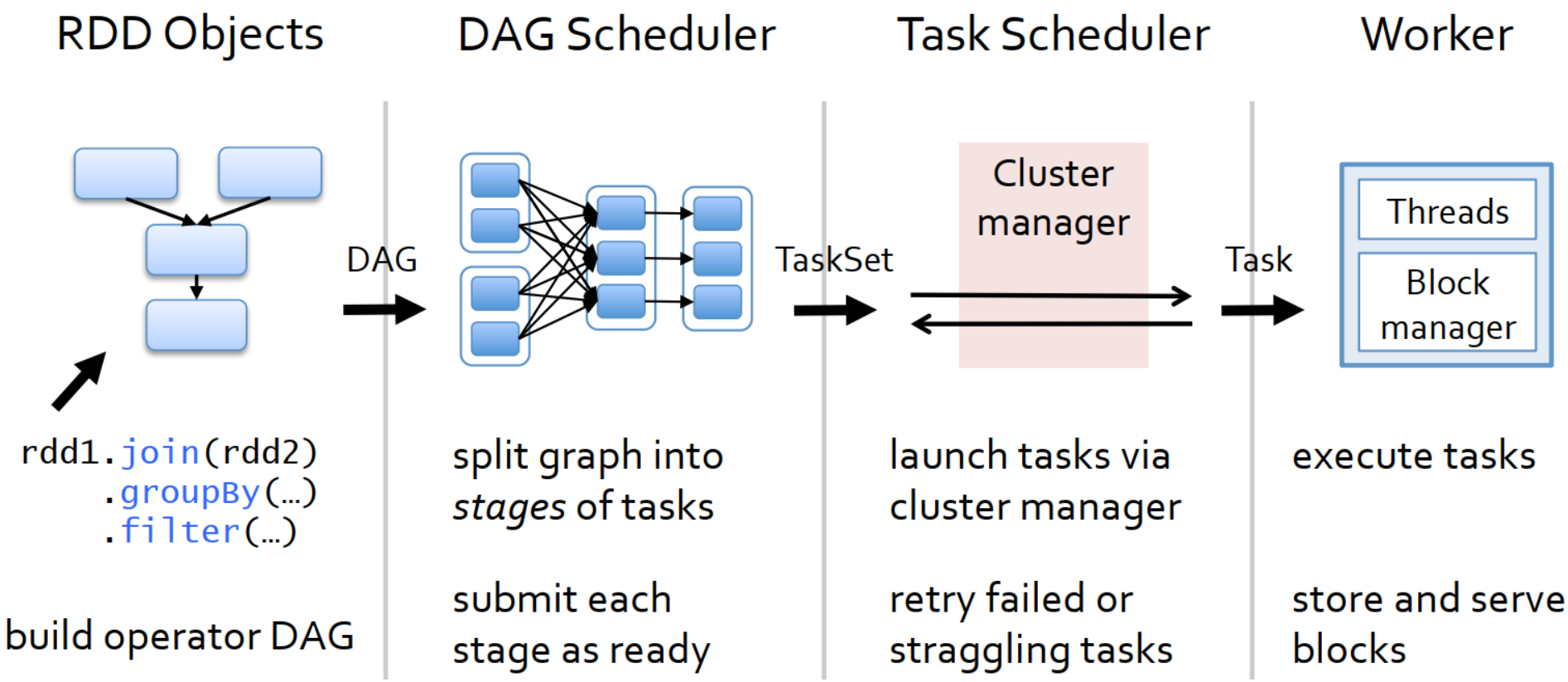
RDD Methods

fit@hcmus

Transformations (applies some logic and creates a new RDD)		Actions (return something to driver)
map filter sample cache union	**Map like**	reduce collect count saveAsX take …
groupByKey reduceByKey sortBy Repartition	Shuffle **Reduce-like,** though not always exactly	

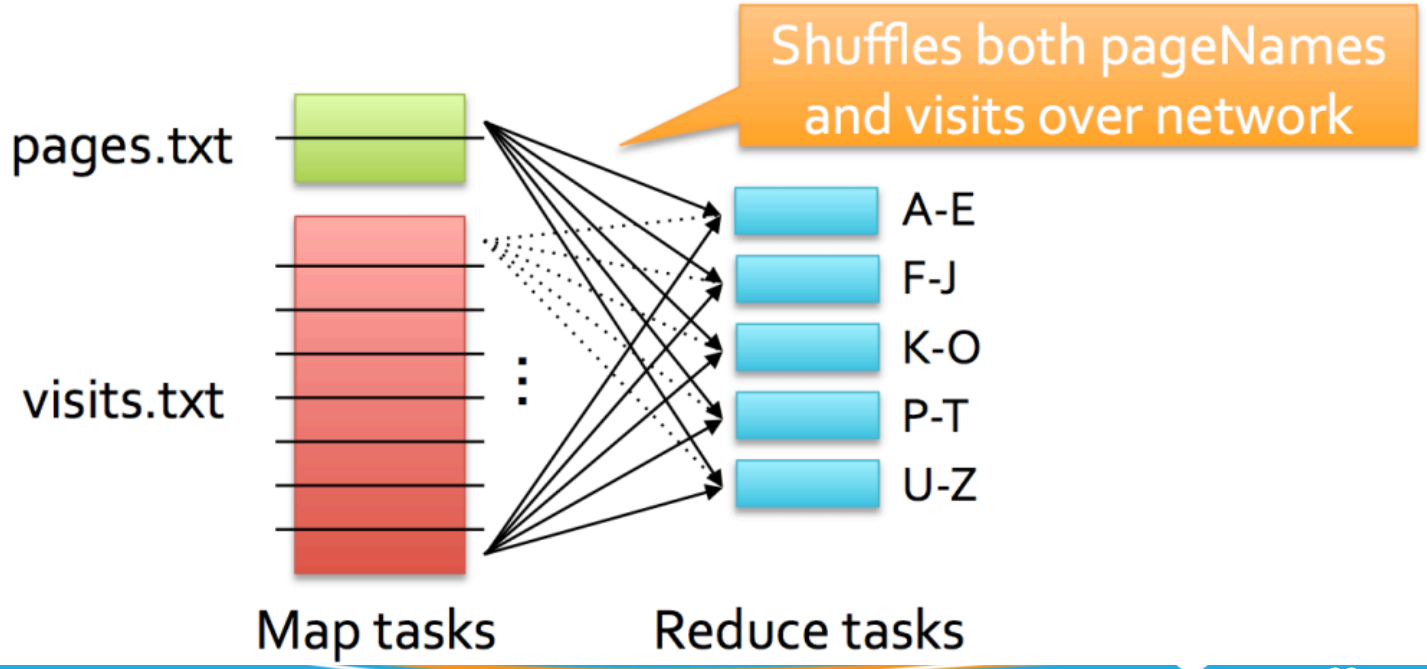
RDD -> Stages -> Tasks

fit@hcmus



Simple Example

```
// Load RDD of (URL, name) pairs
val pageNames = sc.textFile("pages.txt").map(...)

// Load RDD of (URL, visit) pairs
val visits = sc.textFile("visits.txt").map(...)

val joined = visits.join(pageNames)
```

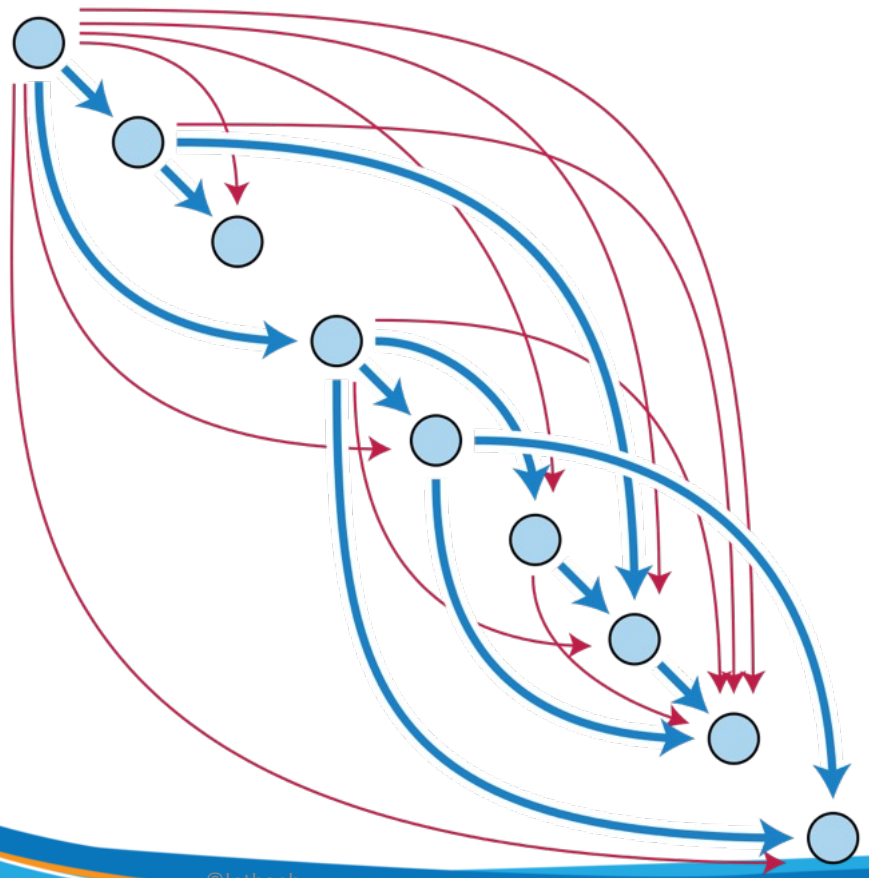


Outline

- **Review Spark Architecture**
  - Overall Architecture
  - RDD
  - DAG
  - Stage, Task
  - Example
- DataFrames and Datasets
- Structured API Execution
- Basic Structured Operations

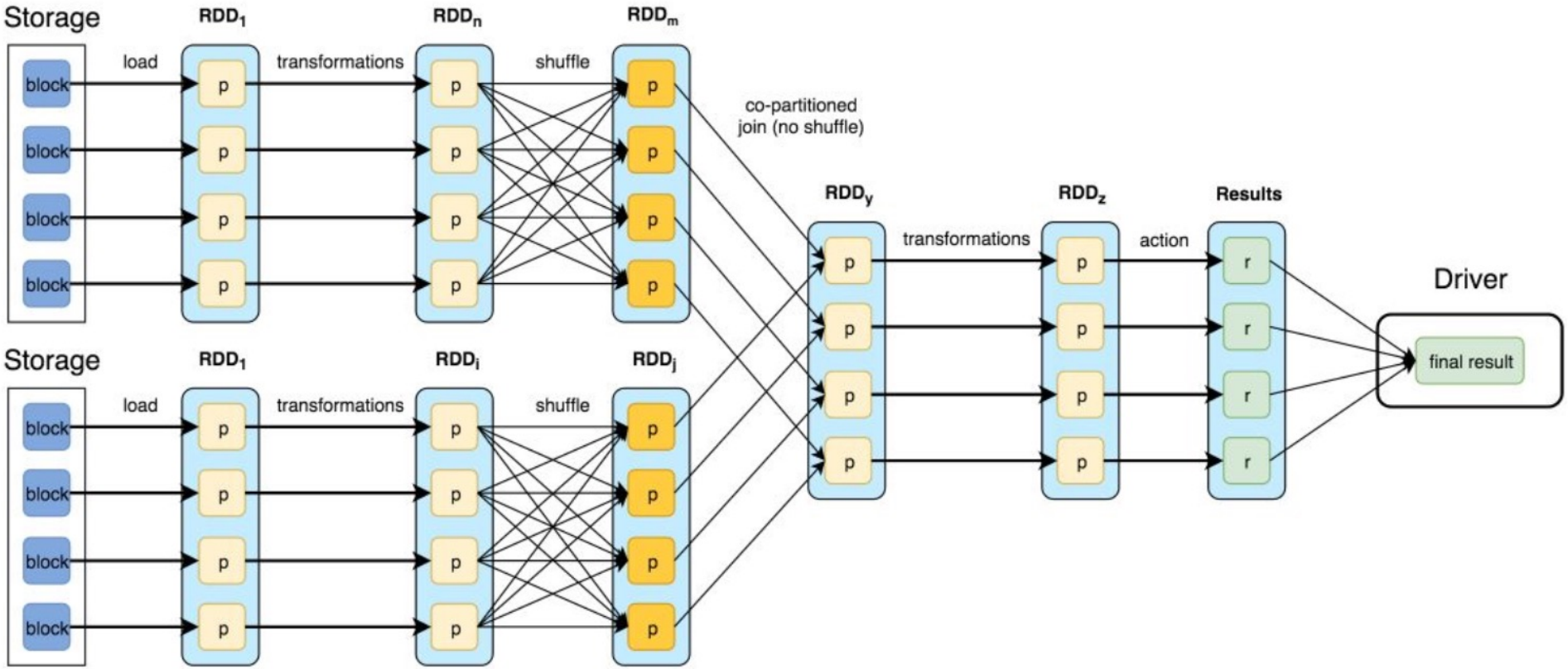
Spark applications: DAG

- DAG is a graph denoting the sequence of operations that are performed on the target RDD

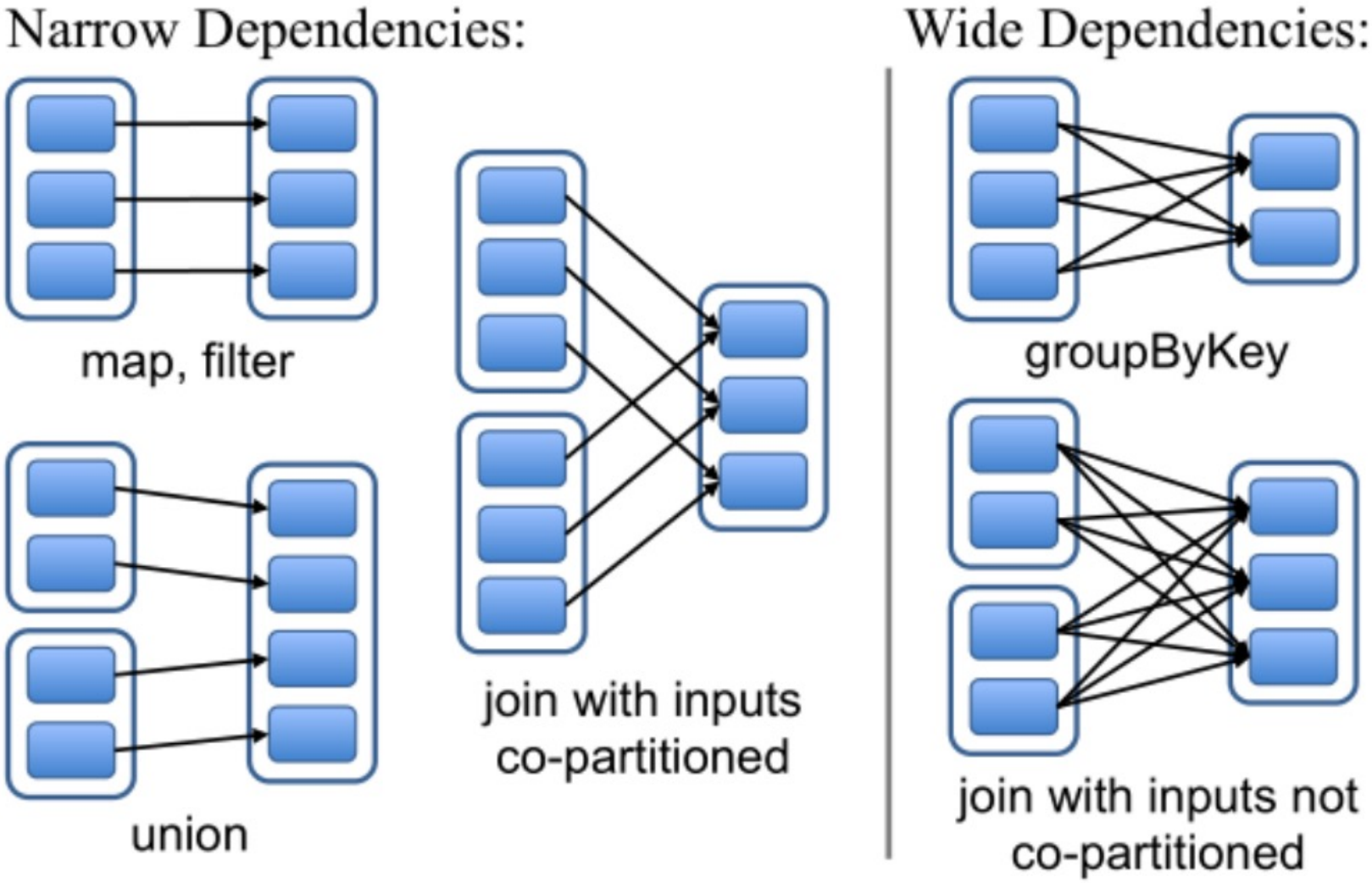


@lnthanh

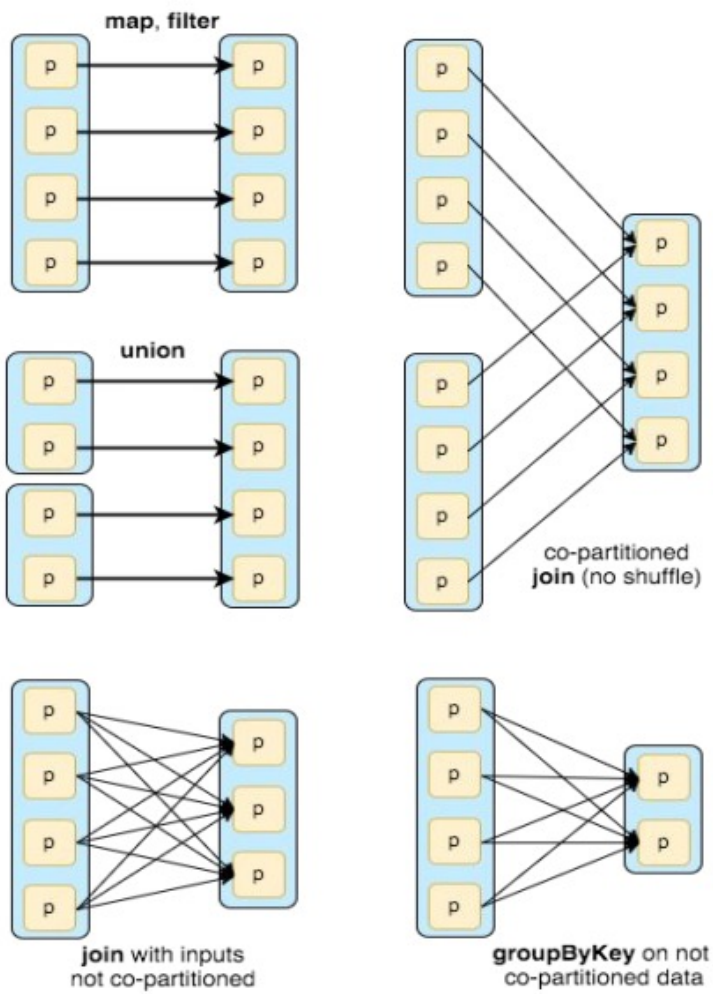
DAG


Storage
RDD1
RDDn
RDDm
load
transformations
shuffle
block
p
co-partitioned
join (no shuffle)
RDDy
RDDz
Results
transformations
action
r
Driver
final result
Storage
RDD1
RDDi
RDDj
load
transformations
shuffle

DAG Scheduler



Dependency Types



- Narrow (pipelineable)
  - each partition of the parent RDD is used by at most one partition of the child RDD
  - allow for pipelined execution on one cluster node
  - failure recovery is more efficient as only lost parent partitions need to be recomputed
- Wide (shuffle)
  - multiple child partitions may depend on one parent partition
  - require data from all parent partitions to be available and to be shuffled across the nodes
  - if some partition is lost from all the ancestors a complete recomputation is needed

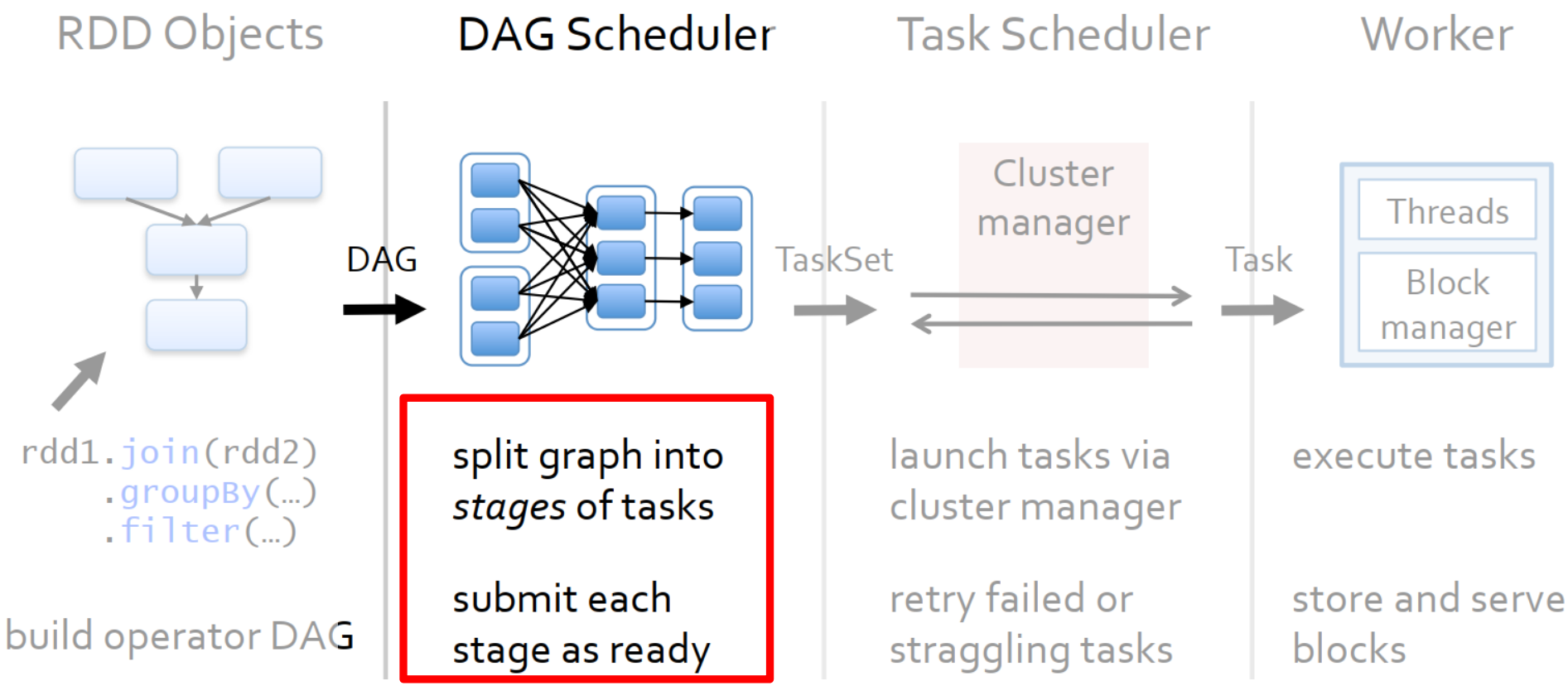
Outline

- **Review Spark Architecture**
  - Overall Architecture
  - RDD
  - DAG
  - Stage, Task
  - Example
- DataFrames and Datasets
- Structured API Execution
- Basic Structured Operations

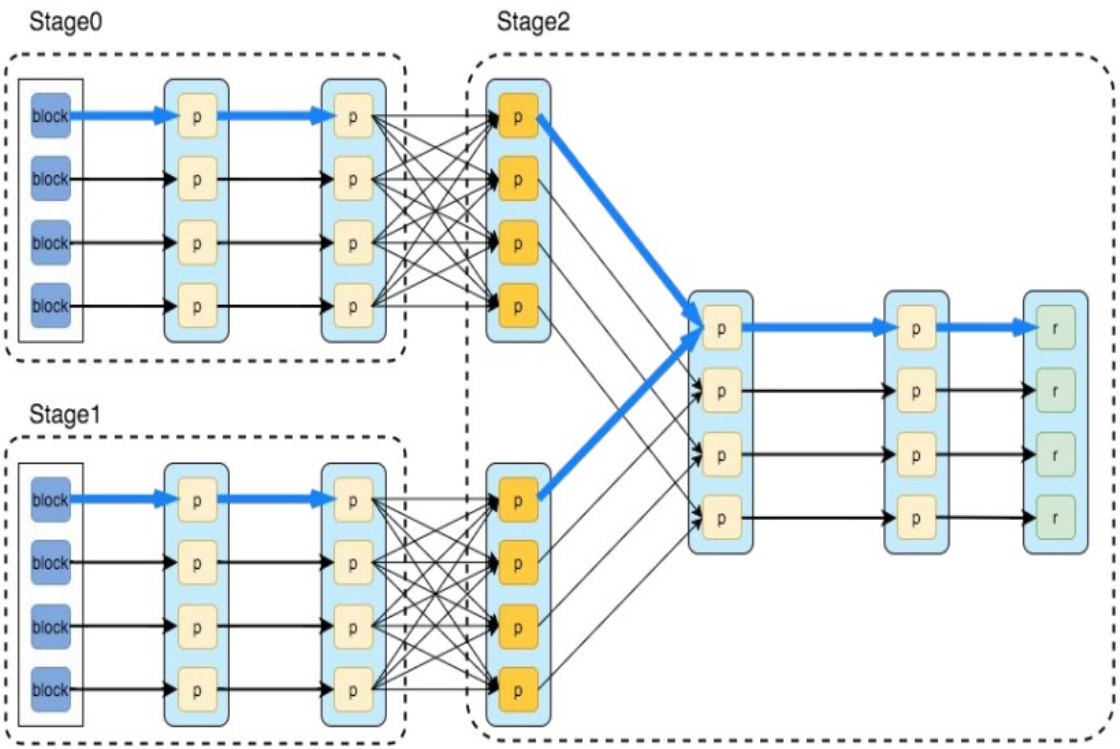
@lnthanh

RDD -> Stages -> Tasks

fit@hcmus

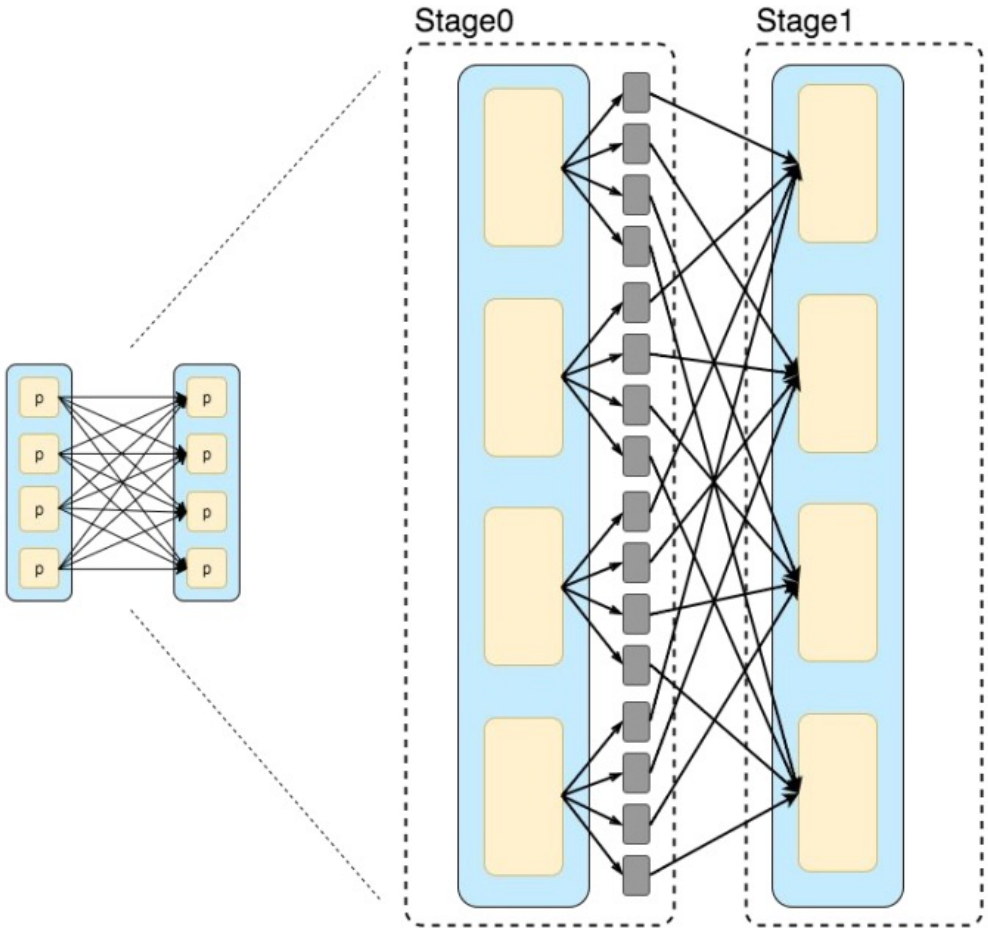


Stages and Tasks



- Stages breakdown strategy
  - check backwards from final RDD
  - add each "narrow" dependency to the current stage
  - create new stage when there's a shuffle dependency
- Tasks
  - *ShuffleMapTask* partitions its input for shuffle
  - *ResultTask* sends its output to the driver

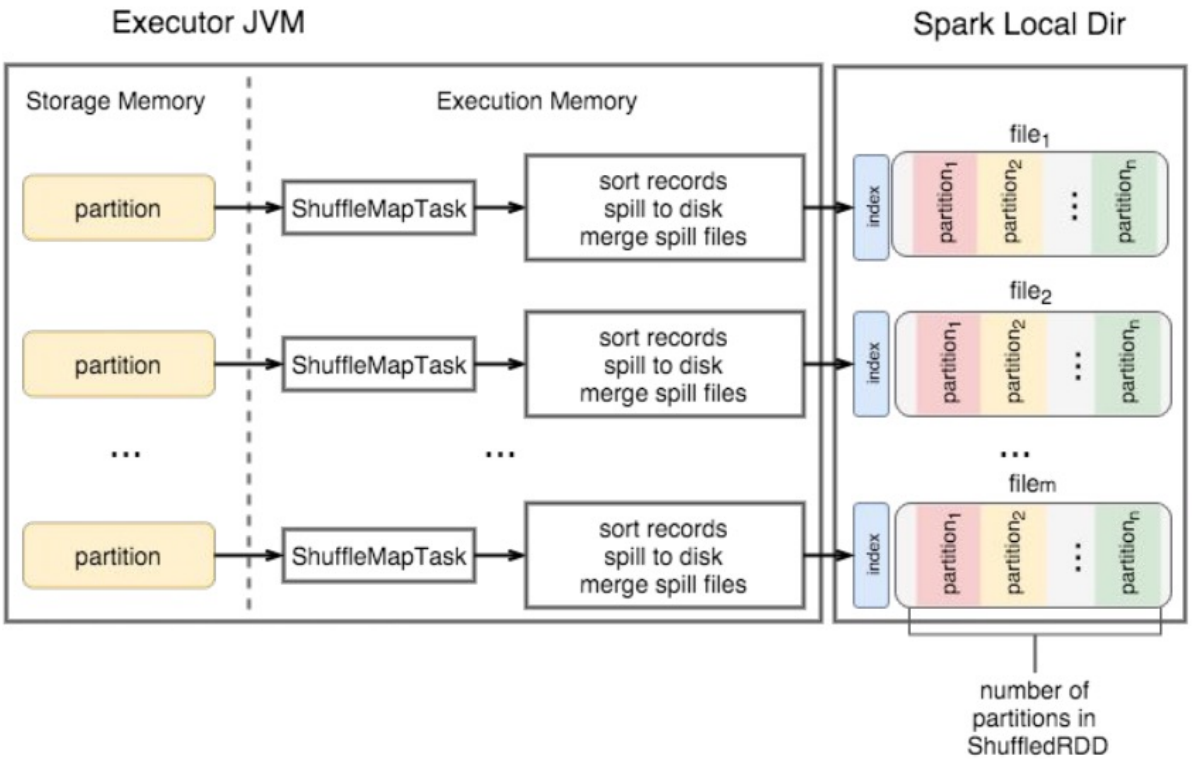
Stages and Tasks



- Shuffle Write
  - redistributes data among partitions and writes files to disk
  - each *hash shuffle* task creates one file per "reduce" task (total = MxR)
  - *sort shuffle* task creates one file with regions assigned to reducer
  - *sort shuffle* uses in-memory sorting with spillover to disk to get final result
- Shuffle Read
  - fetches the files and applies *reduce()* logic
  - if data ordering is needed then it is sorted on "reducer" side for any type of shuffle (SPARK-2926)

Stages and Tasks

- Sort Shuffle



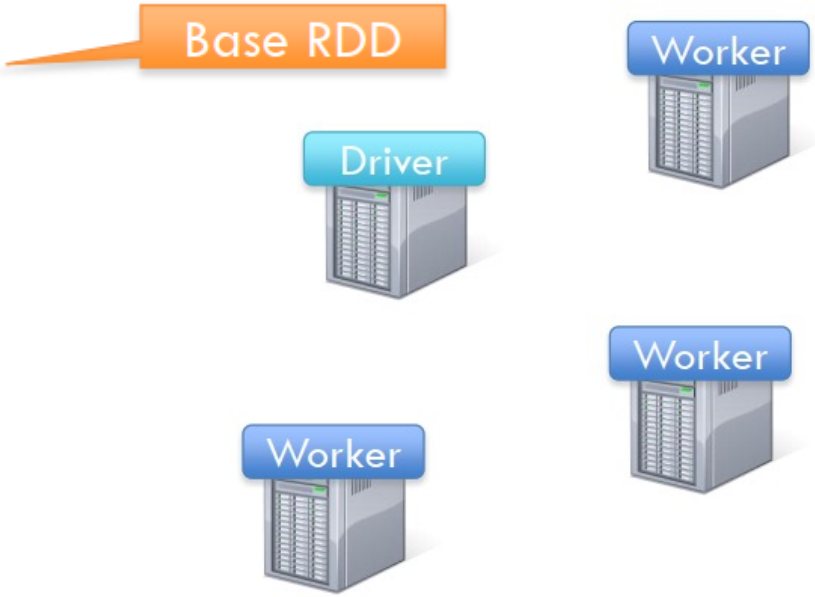
- Incoming records accumulated and sorted in memory according their target partition ids
- Sorted records are written to file or multiple files if spilled and then merged
- *index* file stores offsets of the data blocks in the data file
- Sorting without deserialization is possible under certain conditions (SPARK-7081)

Outline

- **Review Spark Architecture**
  - Overall Architecture
  - RDD
  - DAG
  - Stage, Task
  - Example
- DataFrames and Datasets
- Structured API Execution
- Basic Structured Operations

Example 1: Log Mining Example

lines = spark.textFile("hdfs://...")



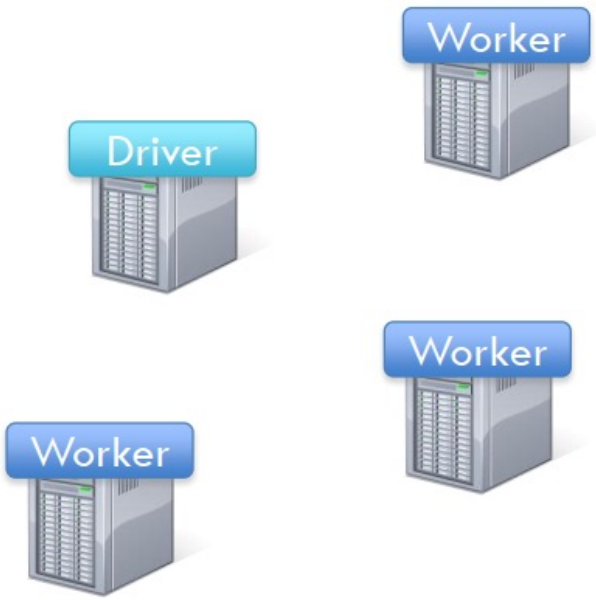
Example 1: Log Mining Example

fit@hcmus

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR
```

Base RDD

Transformed RDD



Example 1: Log Mining Example

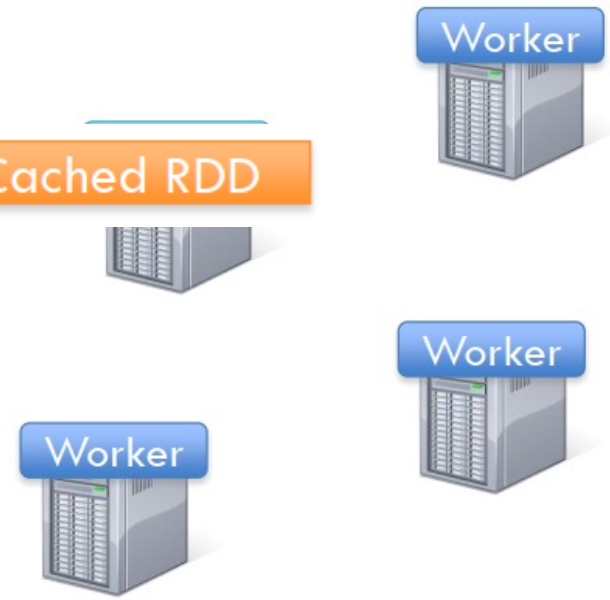
fit@hcmus

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
```

Base RDD

Transformed RDD

Cached RDD



Example 1: Log Mining Example

fit@hcmus

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ER
messages = errors.map(_.split('\t'
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo",).count
```

Base RDD

Transformed RDD

Action

Cached RDD

Worker

Driver

Parallel operation

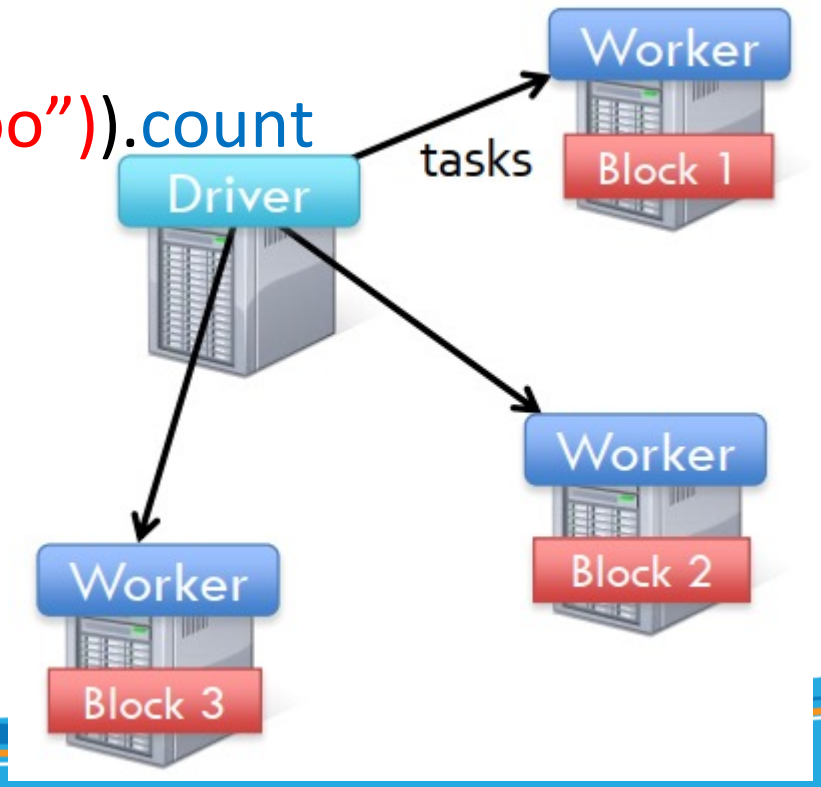
Worker

Worker

Example 1: Log Mining Example

fit@hcmus

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo")).count
```

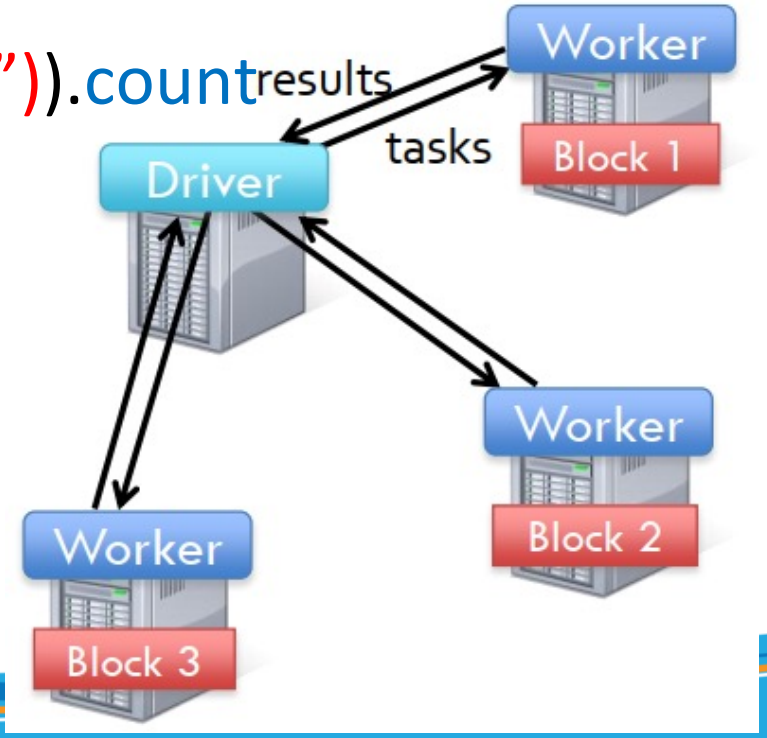


@lnthanh

Example 1: Log Mining Example

fit@hcmus

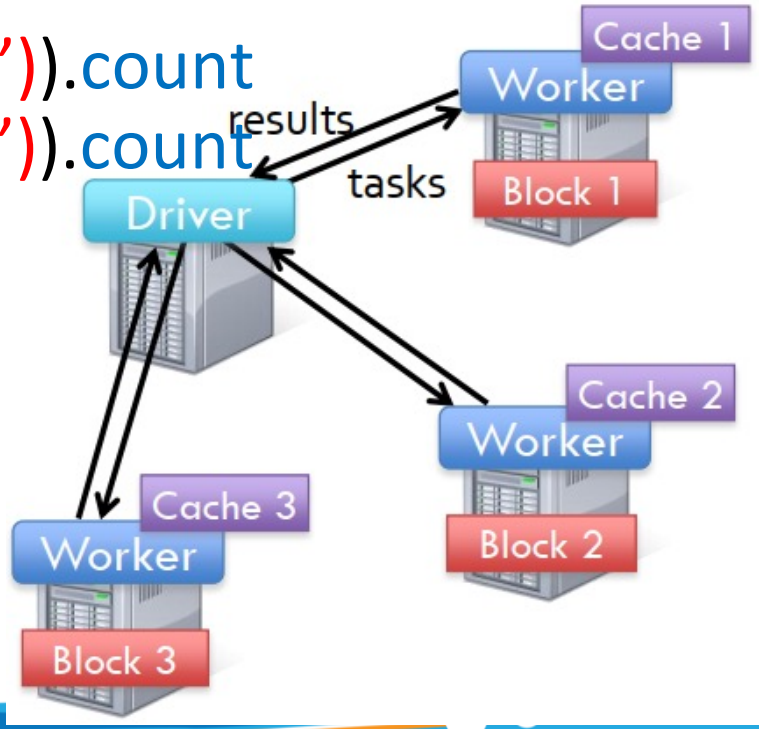
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo")).count
```



Example 1: Log Mining Example

fit@hcmus

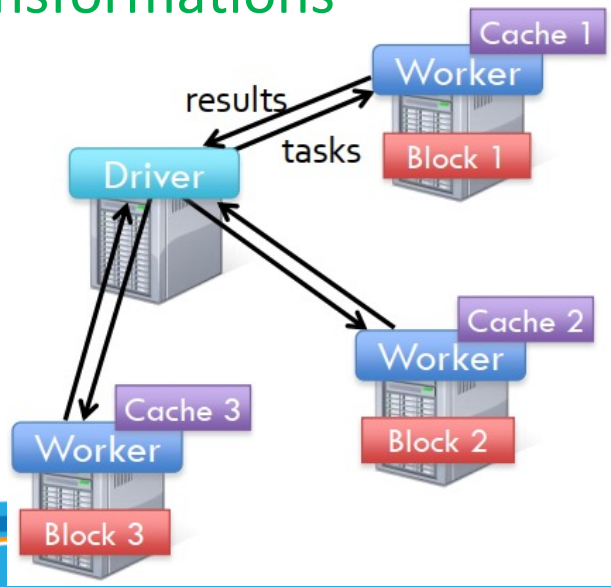
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
```



Example 1: Log Mining Example

fit@hcmus

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
different = cachedMsgs.aDifferentChainOfTransformations
count = different.count
```

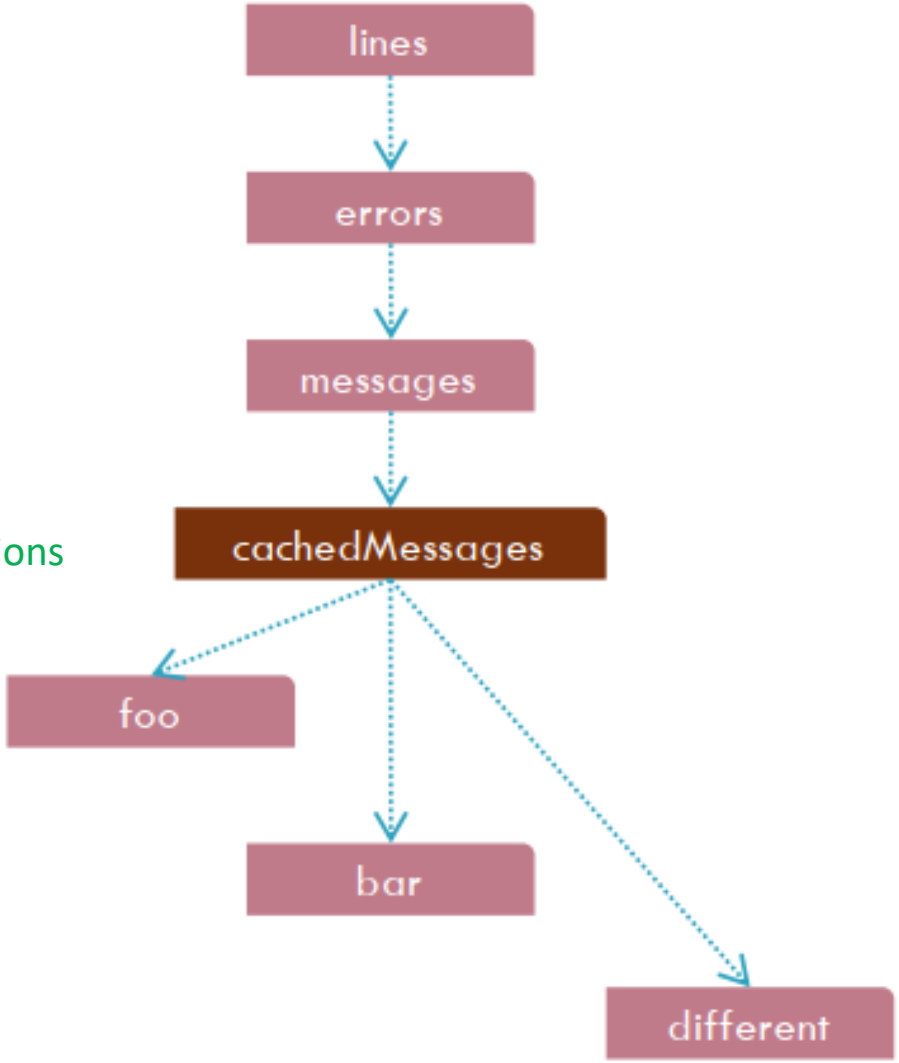


@lnthanh

Example 1: Log Mining Example

fit@hcmus

```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

messages = errors.map(_.split('\t')(2))

cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count

. . .
different = cachedMsgs.aDifferentChainOfTransformations
count = different.count
```



Action

©lnthanh

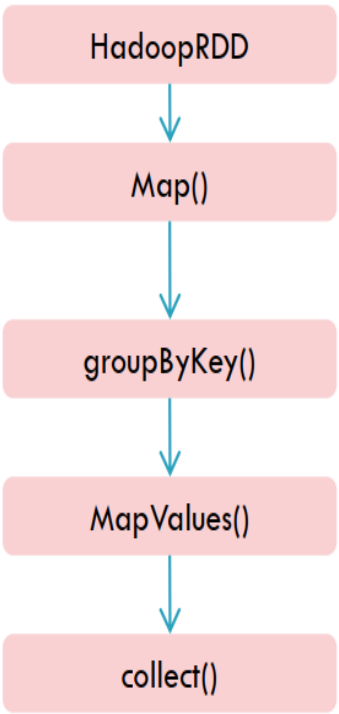
Example 2: Initials Count

fit@hcmus

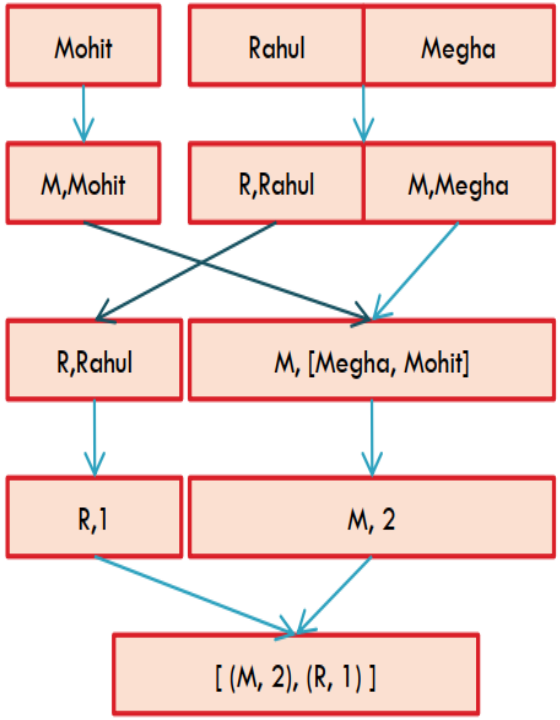
From spark-shell:

```
Val textFile =
sc.textFile("hdfs:.//names.txt", 2)
        .map(name =>
        (name.charAt(0), name) )

        .groupByKey()

        .mapValues(names =>
        names.toArray.size)

        .collect
```

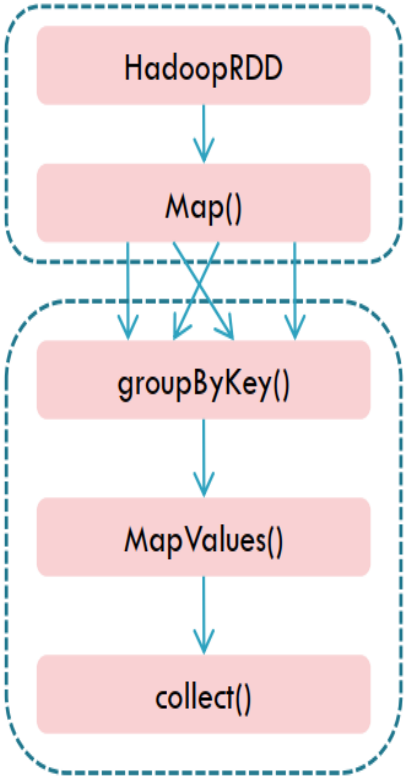
spark-shell: it gives sc as context



What happens to real data

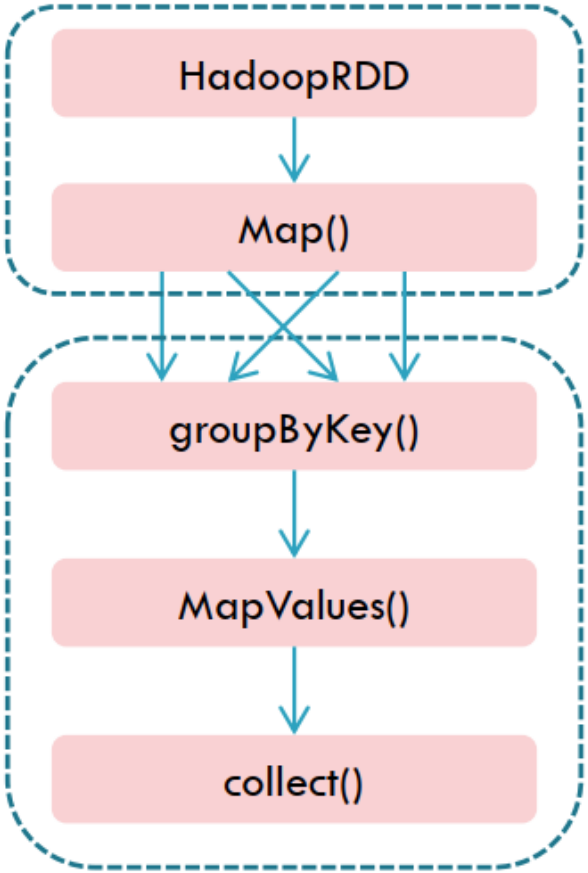


How tasks are created? Only 2



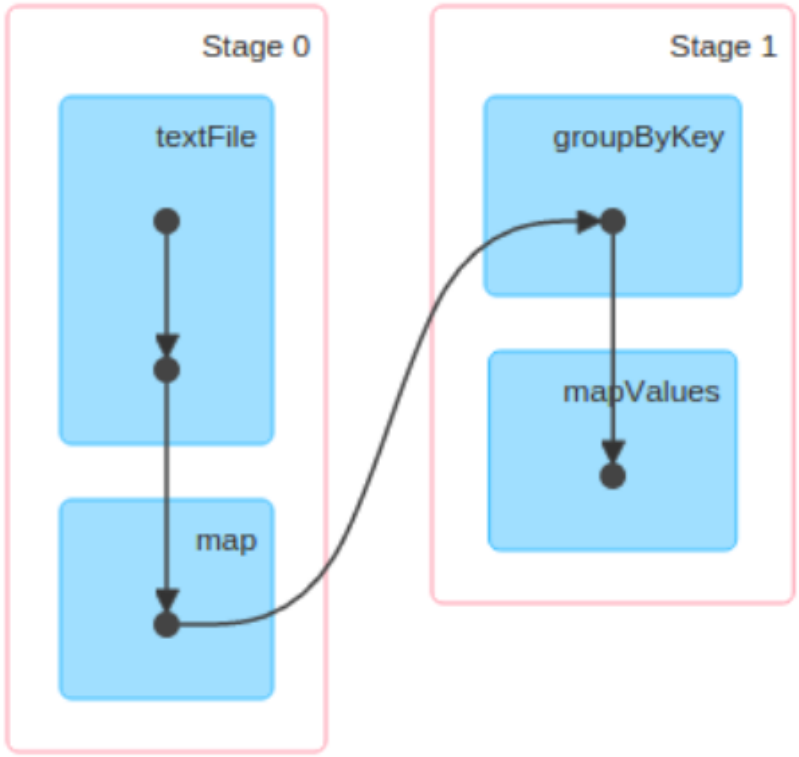
@lnthanh

Example 2: Initials Count



Stage - 1

Stage - 2

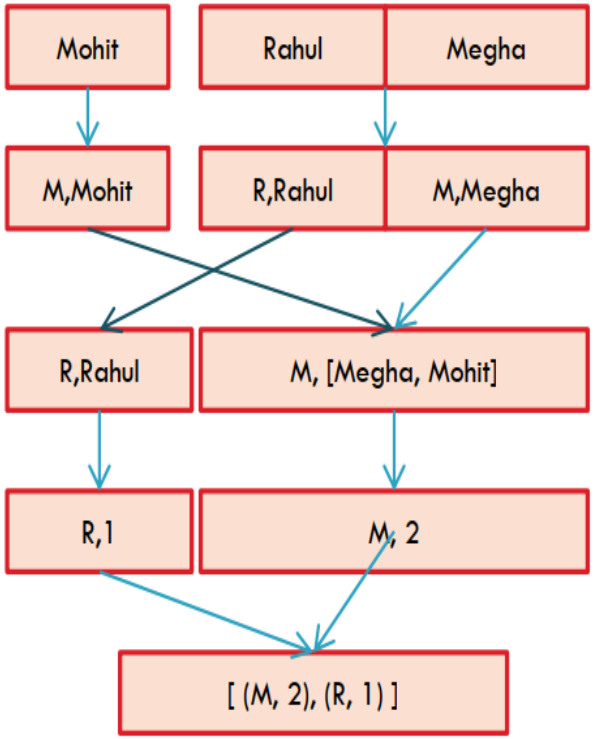


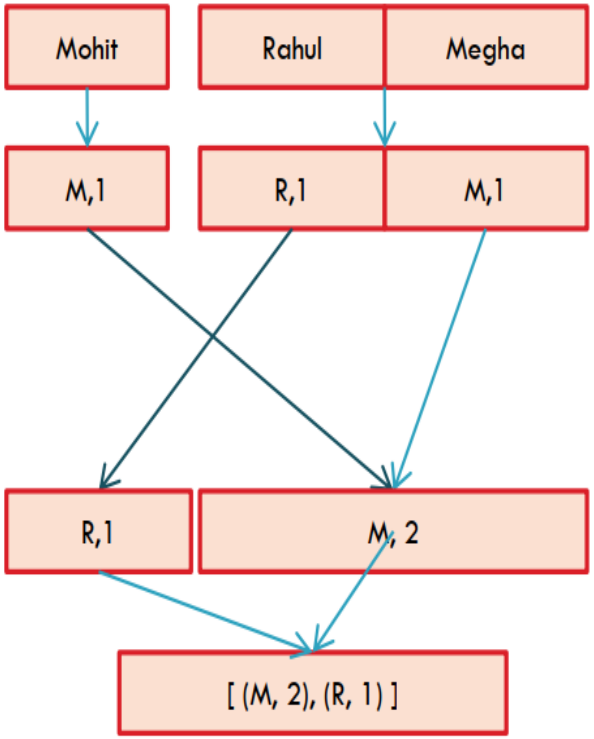
Example 2: Initials Count

fit@hcmus

- Reduce data movement

```
Val textFile =
sc.textFile("hdfs:.//names.txt", 2)
        .map(name =>
        (name.charAt(0), name) )

        .groupByKey()

        .mapValues(names =>
        names.toArray.size)

        .collect
```



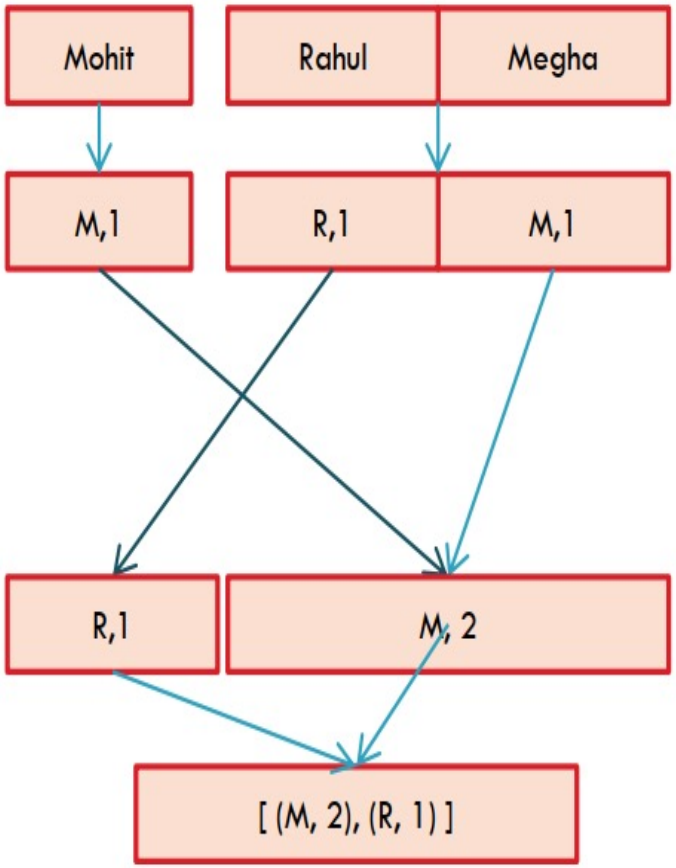


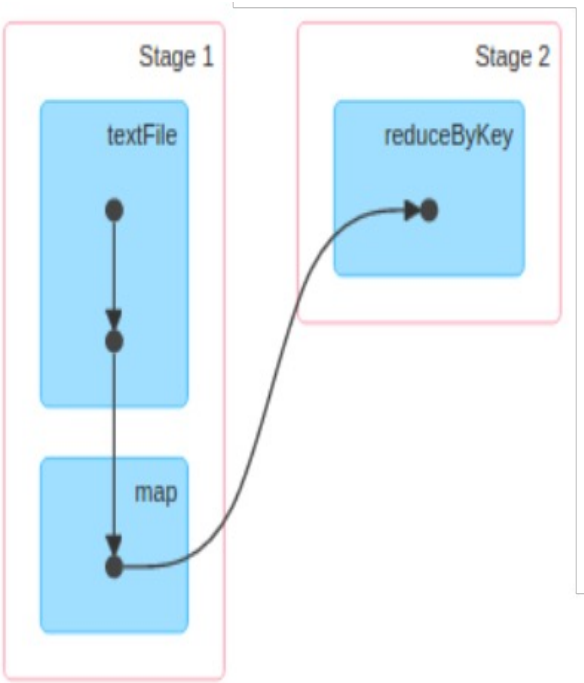
@lnthanh

Example 2: Initials Count

fit@hcmus

```
val textFile =
sc.textFile("names.txt", 2)

val kVal = textFile.map(name =>
(name.charAt(0), 1) )

val kValGrouped =
kVal.reduceByKey(_ + _)

keyCount.collect
```





Outline

- Review Spark Architecture
- **DataFrames and Datasets**
- Structured API Execution
- Basic Structured Operations

@lnthanh

Spark Structured APIs

fit@hcmus

- A tool for manipulating all sorts of data
  - From unstructured log files to semi-structured CSV files and highly structured Parquet files.
- Three core types of distributed collection APIs include

Datasets | DataFrames | SQL Tables and Views

- Most APIs apply to both batch and streaming computation.

©lnthanh

Review of fundamental concepts

fit@hcmus

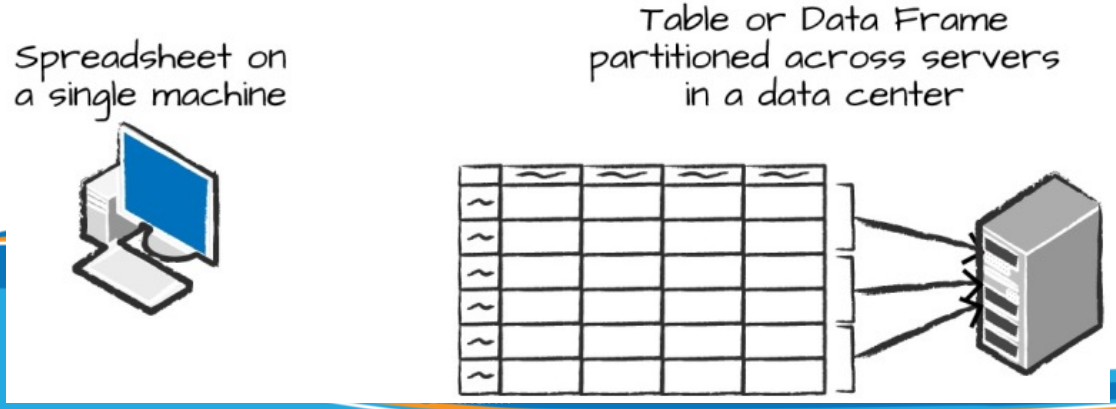
- The user specifies transformations, which together build up a DAG of instructions.
- An action begins the process of executing that graph, as a single job, breaking it down into stages and tasks to execute across the cluster
- The logical structures are DataFrames and Datasets
  - Transformations: create a new DataFrame or Dataset
  - Actions: start computation or convert to native language types

DataFrames and Datasets

Logical structures that manipulate with transformations and actions

DataFrames and Datasets

- Distributed table-like collections with well-defined rows and columns.
  - Each column must have the same number of rows and its type information must be consistent for every row in the collection.
- Represent immutable and lazily evaluated plans
  - Specify what operations to apply to data residing at a location to generate some output



Schemas

- Define the column names and types of a DataFrame

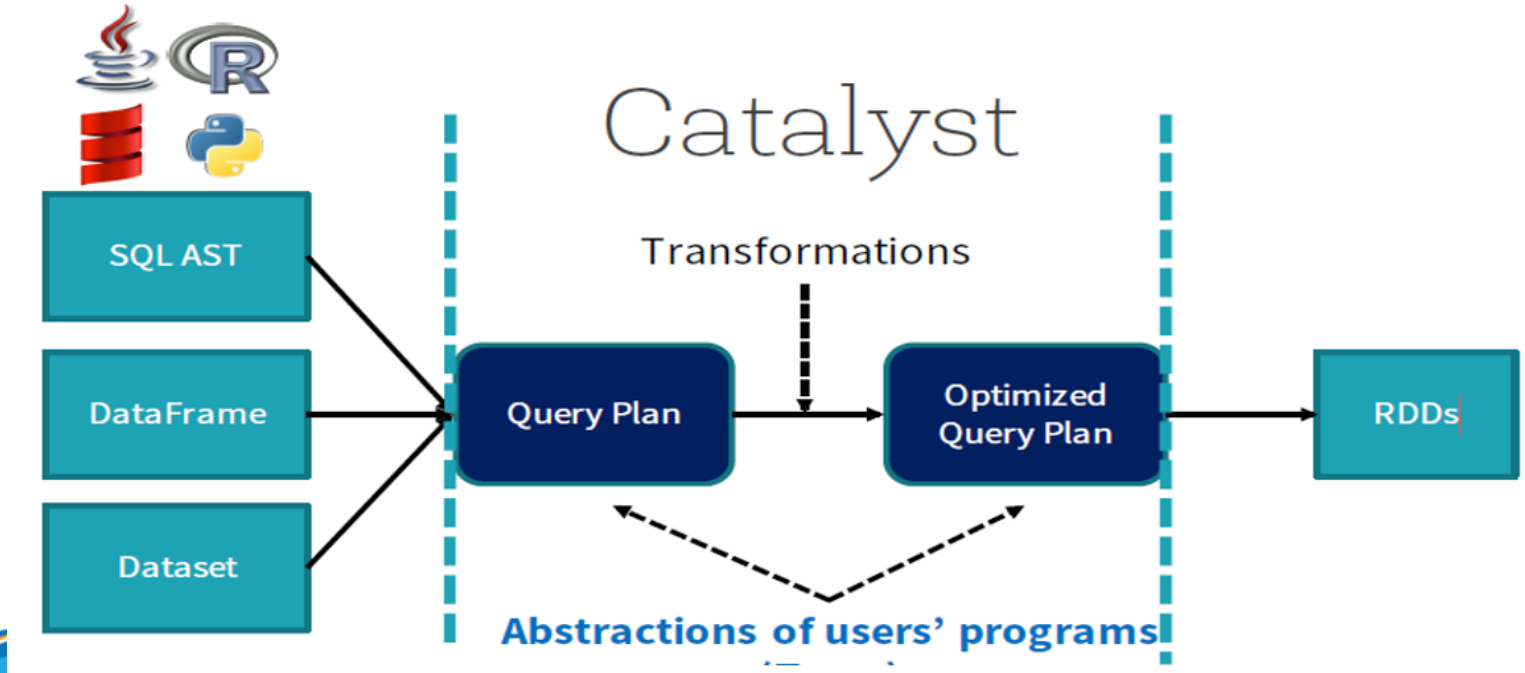
```
[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Ireland', count=344),
 Row(DEST_COUNTRY_NAME='Egypt', ORIGIN_COUNTRY_NAME='United States', count=15),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='India', count=62),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Singapore', count=1),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Grenada', count=62),
 Row(DEST_COUNTRY_NAME='Costa Rica', ORIGIN_COUNTRY_NAME='United States', count=588),
 Row(DEST_COUNTRY_NAME='Senegal', ORIGIN_COUNTRY_NAME='United States', count=40),
 Row(DEST_COUNTRY_NAME='Moldova', ORIGIN_COUNTRY_NAME='United States', count=1)]
```

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

- Schemas can be defined manually or from a data source.

Structured Spark types

- Spark is effectively a programming language of its own.
- An expression written in an input language is converted to its corresponding internal Catalyst representation.



Structured Spark types

fit@hcmus

- Spark types map directly to the different language APIs that Spark maintains, using a lookup table for each of these.
  - Available for Scala, Java, Python, SQL, and R.
  - Even if we use from Python or R, the majority of our manipulations will operate strictly on Spark types, not Python types.

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

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

- This opens up a wide variety of execution optimizations that make significant differences.

DataFrames vs. Datasets

fit@hcmus

- When to check whether types conform to the specification?

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

- Datasets are only available to JVM–based languages
  - Specify types with case classes (Scala) or Java beans (Java)

@lnthanh

DataFrames vs. Datasets

fit@hcmus

- DataFrames are simply Datasets of Type Row.
  - The `Row` type: Spark's internal representation of its optimized in-memory format → highly specialized and efficient computation
- DataFrames provide users the same efficiency gains to all of Spark's language APIs.

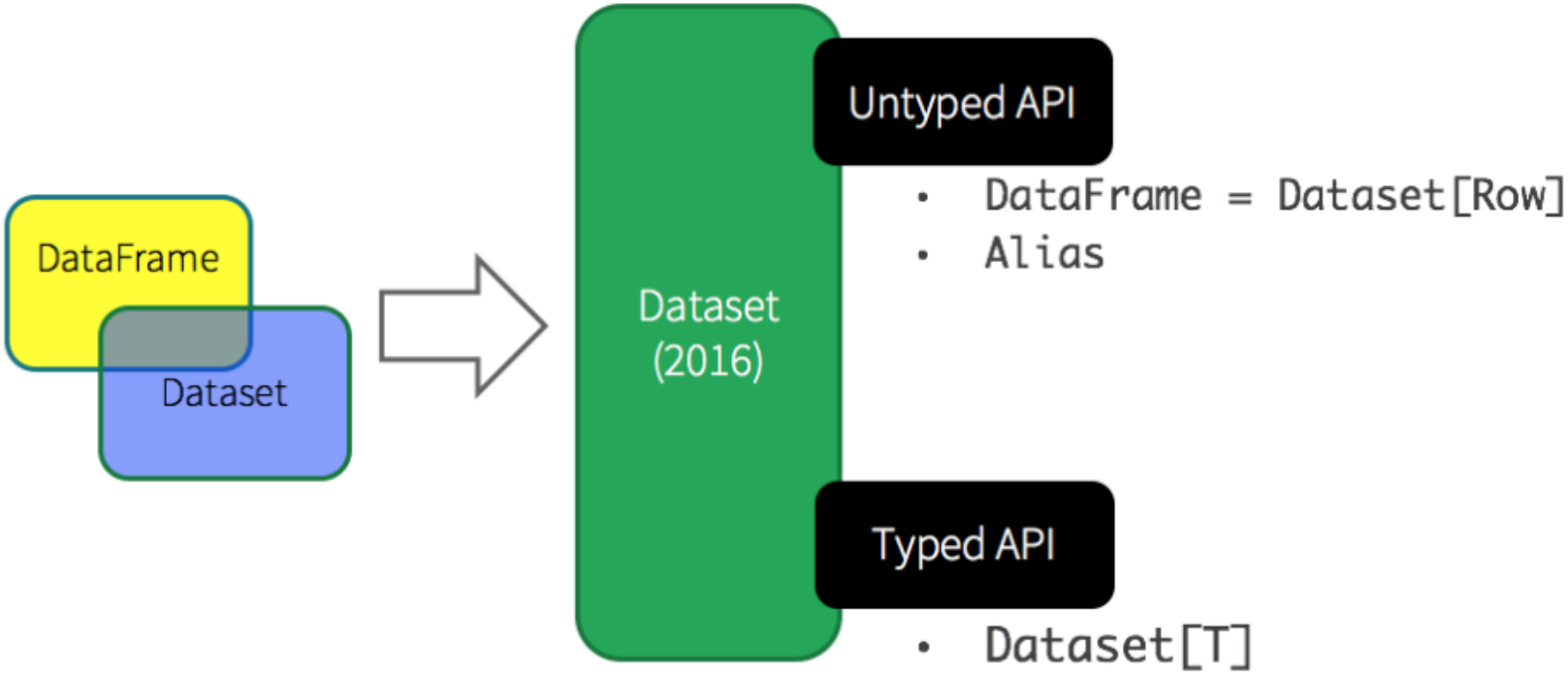
- *If you need strict compile-time checking, read Chapter 11 to learn more about it.*

@lnthanh

DataFrames vs. Datasets

fit@hcmus

Unified Apache Spark 2.0 API



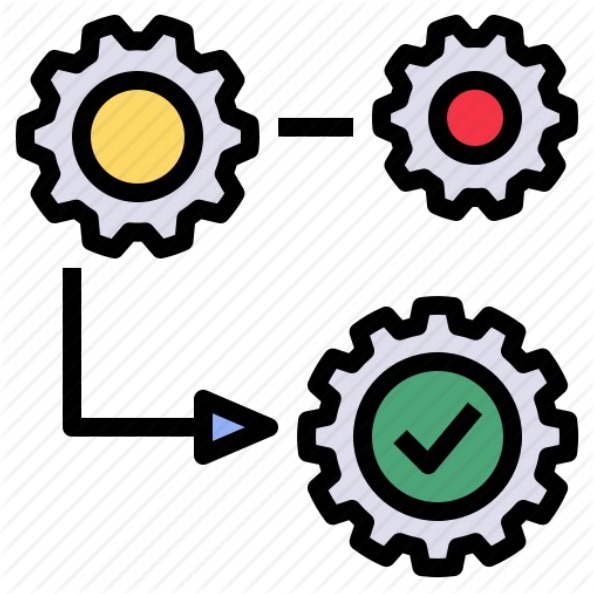
databricks

Columns and Rows

- Column represent a simple type like an *integer* or *string*, a complex type like an *array* or *map*, or a *null value*.
  - Spark `Column` types as columns in a table
- A row is nothing more than a record of data.
  - Created manually from scratch, SQL, RDDs, or data sources

▸ (2) Spark Jobs

```
+----------+------+------------+
|      word| count|    uniqueID|
+----------+------+------------+
|Databricks| 20000|317827579904|
|     Spark|100000|635655159808|
|    Hadoop|  3000|953482739712|
+----------+------+------------+
```



Structured API Execution

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

Spark types

fit@hcmus

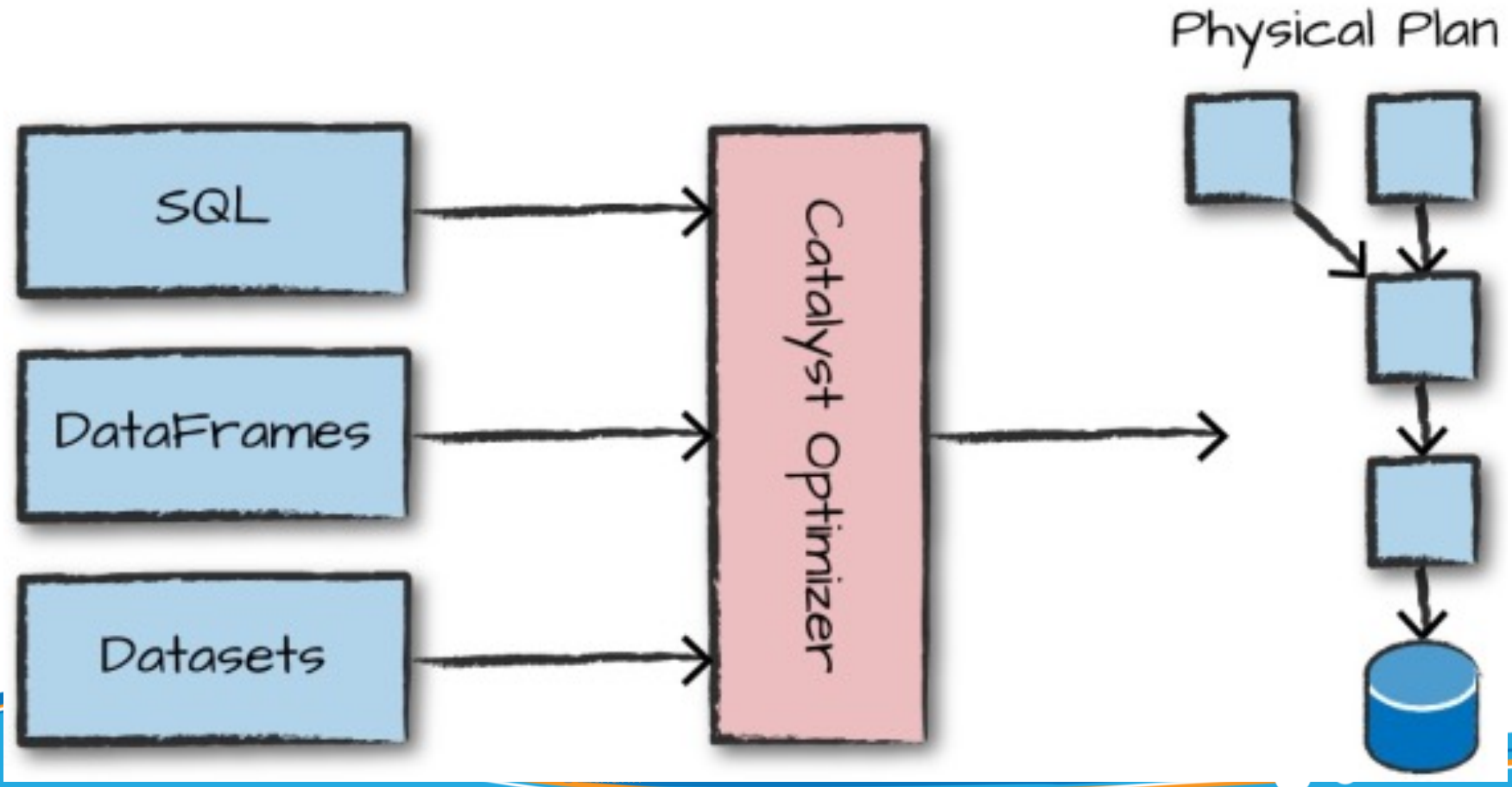
- Spark has a large number of internal type representations.
- To declare a column to be of a certain type, consult the following tables for specific languages
  - *Table 4-1. Python type reference*
    - Note that Python types at times have certain requirements
  - *Table 4-2. Scala type reference*
  - *Table 4-3. Java type reference*

```
# in Python
from pyspark.sql.types import *
b = ByteType()
```

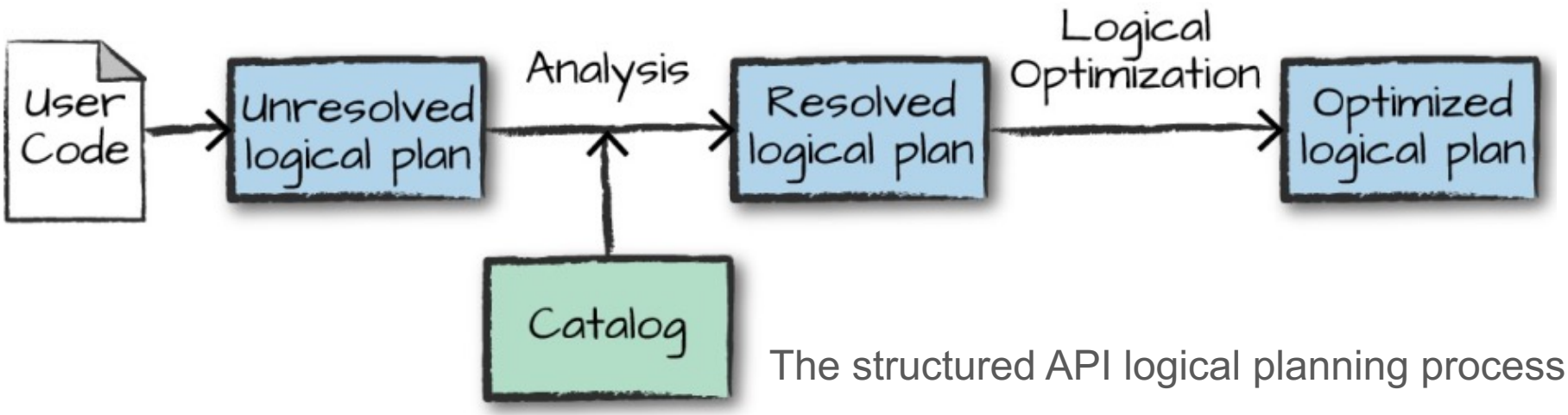
ByteType: int or long, 1-byte signed integer numbers within the range of –128 to 127.

Structured API execution

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



Logical planning



The structured API logical planning process

- Take user code and convert it into a logical plan
- This plan represents a set of abstract transformations that do not refer to executors or drivers
  - Purely convert user's expressions into the most optimized version

Logical planning

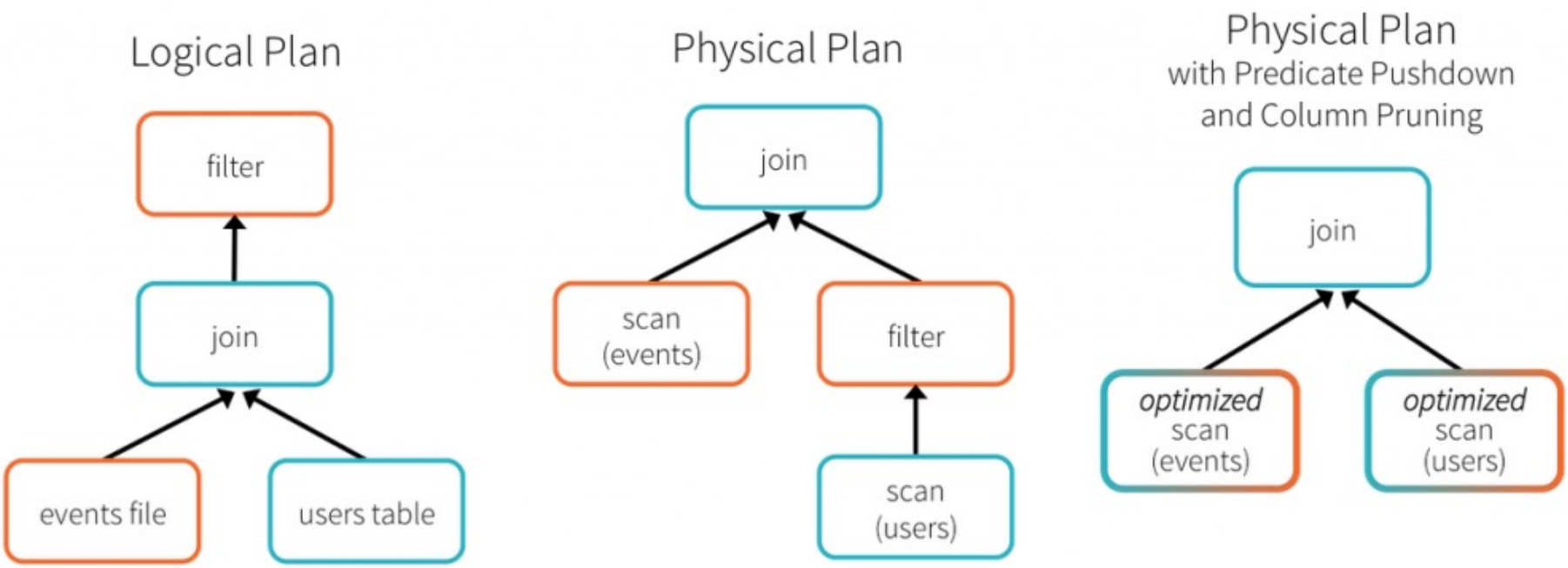
fit@hcmus

- 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
  - If the required table or column name does not exist in the catalog → the analyzer might reject the unresolved logical.
- Pass the resolved plan through the Catalyst Optimizer
  - A collection of rules that attempt to optimize the logical plan by pushing down predicates or selections

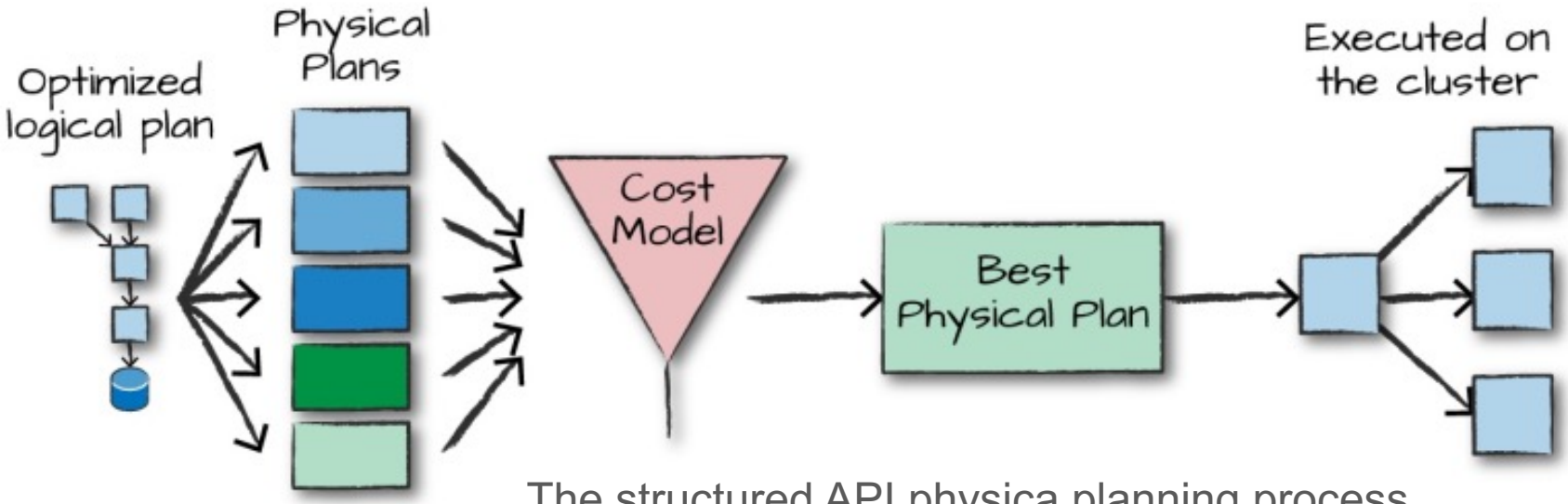
Plan optimization

fit@hcmus

```
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()
```



Physical (Spark) planning



The structured API physica planning process

- Specify how the logical plan will execute on the cluster by generating different physical execution strategies and comparing them through a cost model
  - E.g., choosing 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).
- Resulting 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
- Spark performs further optimizations at runtime
  - Generate native Java bytecode that can remove entire tasks or stages during execution
- Finally, the result is returned to the user.

Basic Structured Operations

Fundamental DataFrame operations while avoiding aggregations and joins

Creating a DataFrame

- A DataFrame consists of a series of records of type Row, and a number of columns.
- Columns represent a computation expression that can be performed on each individual record in the Dataset.

- Creating a DataFrame from a data source

```
# in Python
df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
```

©Inthanh

Looking at the schema

- Schemas define the name and type of data in each column, and thus tie everything together.

```
# in Python
spark.read.format("json").load("/data/flight-data/json/2015-summary.json").schema
```

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

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

Or printSchema ()

©Inthanh

Looking at the schema

- Schema inference can vary depending on the type of data being read in.
  - Schemas should be defined manually when working with untyped data sources

```
# in Python
from pyspark.sql.types import StructField, StructType, StringType, LongType
myManualSchema = StructType([
StructField("DEST_COUNTRY_NAME", StringType(), True),
StructField("ORIGIN_COUNTRY_NAME", StringType(), True),
StructField("count", LongType(), False, metadata={"hello":"world"})
])
df = spark.read.format("json").schema(myManualSchema)\
.load("/data/flight-data/json/2015-summary.json"
```

©Inthanh

Looking at the schema

- A schema is a `StructType` made up of a number of fields of type `StructFields`.
  - It can contain other StructTypes when working with complex types.
- Each field has a name, a type, and a Boolean flag
  - The flag specifies whether missing or null values are allowed.
  - Metadata is optionally associated

©Inthanh

Columns and Expressions

- Columns are logical constructions representing a value computed on a per-record basis by means of an expression.
  - Similar to those in Excel spreadsheet, R dataframe, or pandas DataFrame
- There are different ways to construct and refer to columns

```
# in Python
from pyspark.sql.functions import col, column
col("someColumnName")
column("someColumnName")

# Explicit column references
df["count“]
```

©Inthanh

Columns and Expressions

- An expression is a set of transformations on one or more values in a record in a DataFrame, denoted by expr()
  - A function that takes as input one or more column names, resolves them, and then potentially applies more expressions to create a single value for each record.
- It is basically just a DataFrame column reference.
  - That is, expr("someCol") is equivalent to col("someCol").

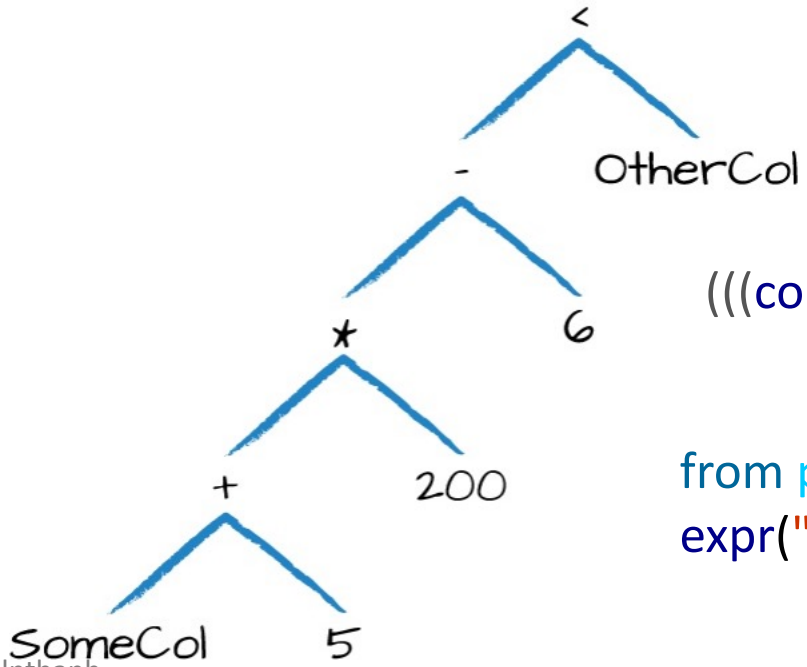
©Inthanh

Columns as expressions

- Columns are just expressions.

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

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




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

```
from pyspark.sql.functions import expr
expr("(((someCol + 5) * 200) - 6) < otherCol")
```

©Inthanh

Records and Rows

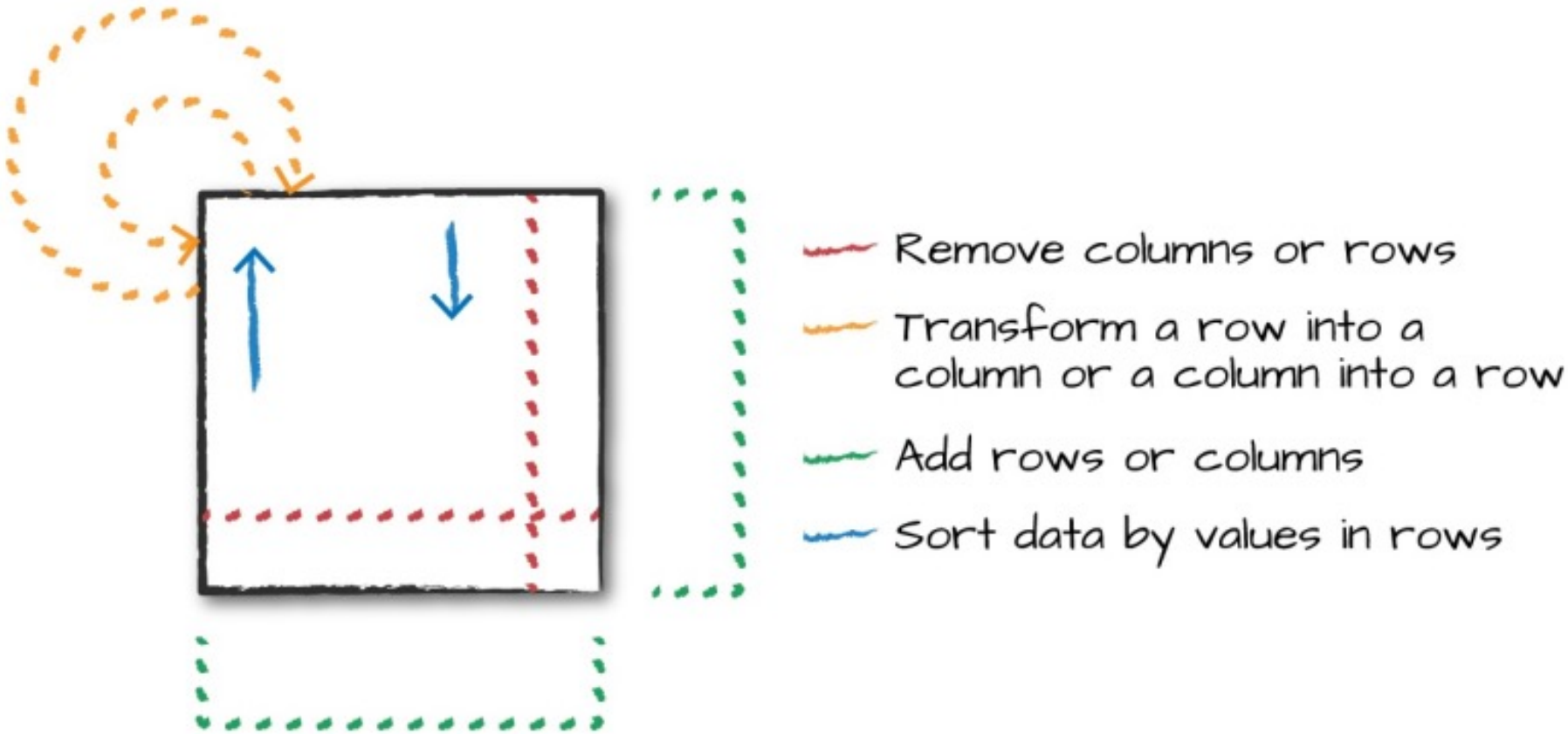
- Each row is a single record, which is an object of type Row.
- DataFrames have schemas but rows themselves do not
- The values must be specified in the same order as the schema of the DataFrame to which they might be appended

```
# in Python
from pyspark.sql import Row
myRow = Row("Hello", None, 1, False)
```

```
# in Python
myRow[0]
myRow[2]
```

©Inthanh

DataFrame transformations



©Inthanh

Creating a DataFrame from rows

- Creation on the fly by taking a set of rows and converting them to a DataFrame

```
# in Python
from pyspark.sql import Row
from pyspark.sql.types import StructField, StructType, StringType, LongType
myManualSchema = StructType([StructField("some", StringType(), True),
StructField("col", StringType(), True),
StructField("names", LongType(), False)])
myRow = Row("Hello", None, 1)
myDf = spark.createDataFrame([myRow], myManualSchema)
myDf.show()
```

```
+-----+----+-----+
| some| col|names|
+-----+----+-----+
|Hello|null|    1|
+-----+----+-----+
```

©Inthanh

select and selectExpr

- Do the DataFrame 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
```

©Inthanh

select and selectExpr

- Columns can be referred in different ways, which user can use them interchangeably.

```
# in Python
from pyspark.sql.functions import expr, col, column
df.select(
expr("DEST_COUNTRY_NAME"),
col("DEST_COUNTRY_NAME"),
column("DEST_COUNTRY_NAME"))\
.show(2)
```

- Attempt to mix Column objects and strings will result in a compiler error.

```
df.select(col("DEST_COUNTRY_NAME"), "DEST_COUNTRY_NAME")
```

©Inthanh

select 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
```

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

```
df.selectExpr("DEST_COUNTRY_NAME as
newColumnName", "DEST_COUNTRY_NAME").show(2)
```

©Inthanh

select and selectExpr

- selectExpr helps build up complex expressions that create new DataFrames.
  - We can add any valid non-aggregating SQL statement, and as long as the columns resolve, it will be valid!

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

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

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

©Inthanh

Converting to Spark Types (Literals)

- Translate from a given programming language's literal value to one that Spark understands
  - This might be a constant value required to compare to later on.

```
# in Python
from pyspark.sql.functions import lit
df.select(expr("*"), lit(1).alias("One")).show(2)
```

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

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

©Inthanh

Columns manipulations

- Add a column

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

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

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

```
# in Python
df.withColumn("withinCountry", expr("ORIGIN_COUNTRY_NAME == DEST_COUNTRY_NAME")).show(2)
```

```
+-----------------+-------------------+-----+-------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|
+-----------------+-------------------+-----+-------------+
|    United States|       Saint Martin|    1|        false|
|    United States|            Romania|   12|        false|
+-----------------+-------------------+-----+-------------+
```

©Inthanh

Columns manipulations

- Rename a column

... dest, ORIGIN_COUNTRY_NAME, count

```
# in Python
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns
```

- Replicate a column

... DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count, Destination

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

- Remove a column

```
# in Python
df.drop("ORIGIN_COUNTRY_NAME")
# or
dfWithLongColName.drop("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME")
```

©Inthanh

Rows manipulation

- Filter rows

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

- 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
```

©Inthanh

Rows manipulation

- Spark automatically performs all filtering operations at the same time regardless of the filter ordering.
  - It is not useful to put multiple filters into the same expression. Just chain them sequentially and let Spark handle the rest

```
# 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
```

©Inthanh

Rows manipulation

- Sort rows: sort in ascending order by default

```
# in Python
df.sort("count").show(5)
df.orderBy("count", "DEST_COUNTRY_NAME").show(5)
df.orderBy(col("count"), col("DEST_COUNTRY_NAME")).show(5)
```

```
# in Python
from pyspark.sql.functions import desc, asc
df.orderBy(expr("count desc")).show(2)
df.orderBy(col("count").desc(), col("DEST_COUNTRY_NAME").asc()).show(2)

-- in SQL
SELECT * FROM dfTable ORDER BY count DESC, DEST_COUNTRY_NAME ASC LIMIT 2
```

©Inthanh

Rows manipulation

- Display some top rows

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|       Saint Martin|    1|
|    United States|            Romania|   12|
|    United States|            Croatia|    2|
|    United States|            Ireland|  291|
|    United States|              India|   62|
+-----------------+-------------------+-----+
```

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

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

```
+--------------------+-------------------+-----+
|   DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+--------------------+-------------------+-----+
|Saint Vincent and...|      United States|    1|
|              Malawi|      United States|    1|
|            Slovakia|      United States|    1|
|          Kazakhstan|      United States|    1|
|       United States|       Saint Martin|    1|
|       United States| Congo (Brazaville)|    1|
+--------------------+-------------------+-----+
```

```
# in Python
df.orderBy(expr("count desc")).limit(6).show()

-- in SQL
SELECT * FROM dfTable ORDER BY count desc LIMIT 6
```

©Inthanh

Finally, our Word Count!

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
linesDF = sqlContext.read.text("input.txt")
linesDF.show(df.count(),truncate=False)

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

```
+---------+-----+
|     word|count|
+---------+-----+
|      pen|   10|
|pineapple|    5|
|    apple|    5|
|        i|    4|
|     have|    4|
|       ah|    3|
|      pie|    2|
|        a|    2|
|       an|    1|
+---------+-----+
```

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

©lnthanh

©lnthanh