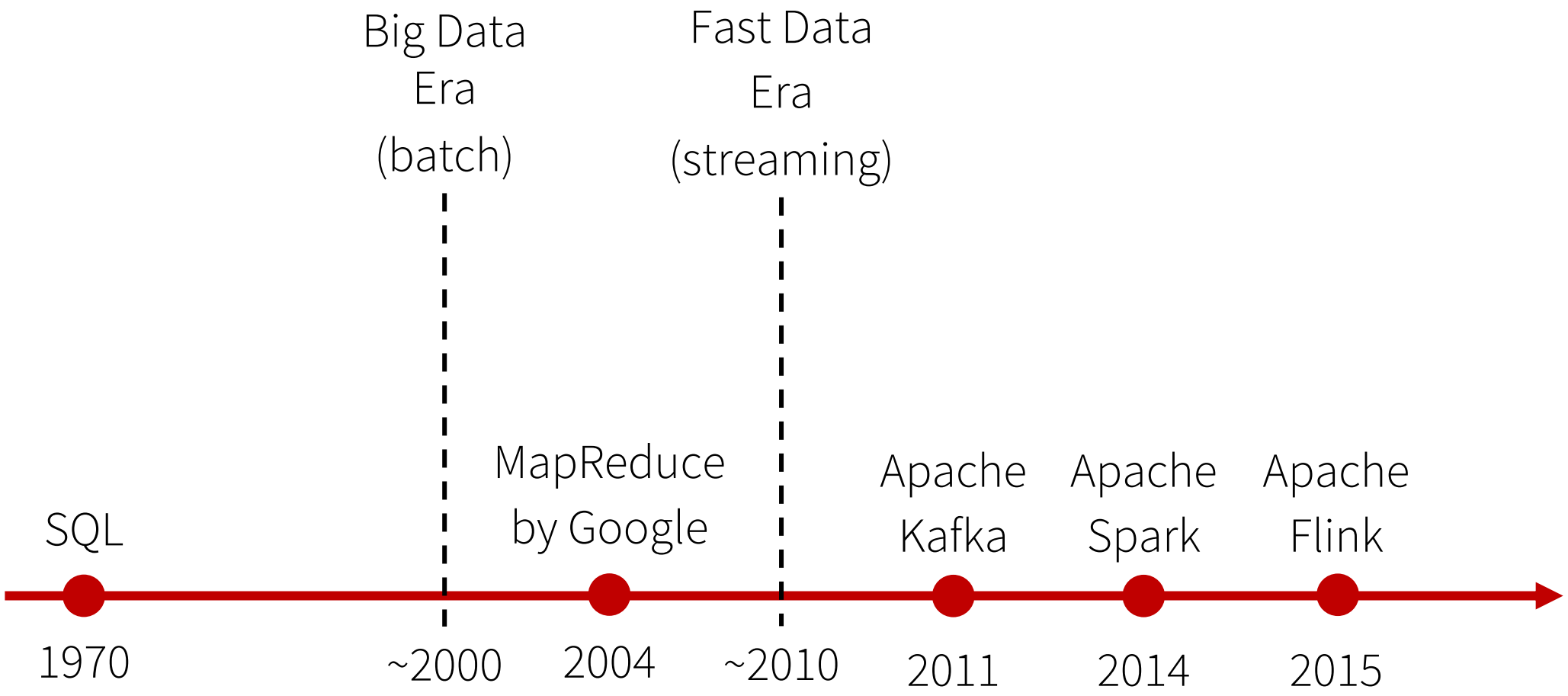
Apache Spark: An Overview

Data Mining Course

Daniele Foroni, December 2019

Data Processing Timeline



Apache Hadoop

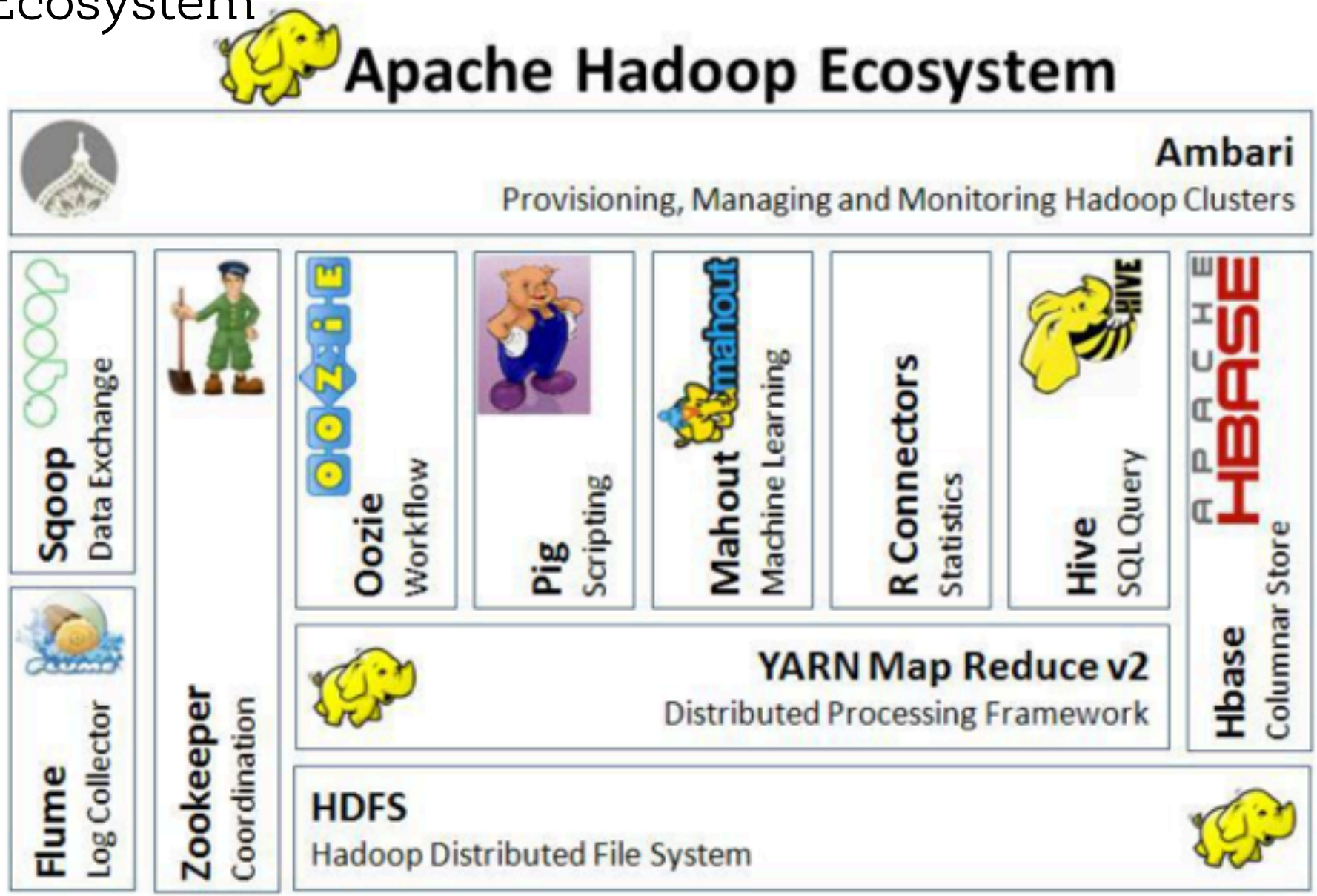
Apache Hadoop: what it is

- An ecosystem of tools for processing “Big Data”.
- Open source project
- Part of the Apache group
- Distributed framework in Java

Apache Hadoop: what it has

- Google's powerful **MapReduce** computation technology
- HDFS (Hadoop Distributed File System) inspired by GFS (Google File System)
- Framework for distributed computation in clusters
- Support from Yahoo!, Facebook, Twitter, IBM, Microsoft (not updated)

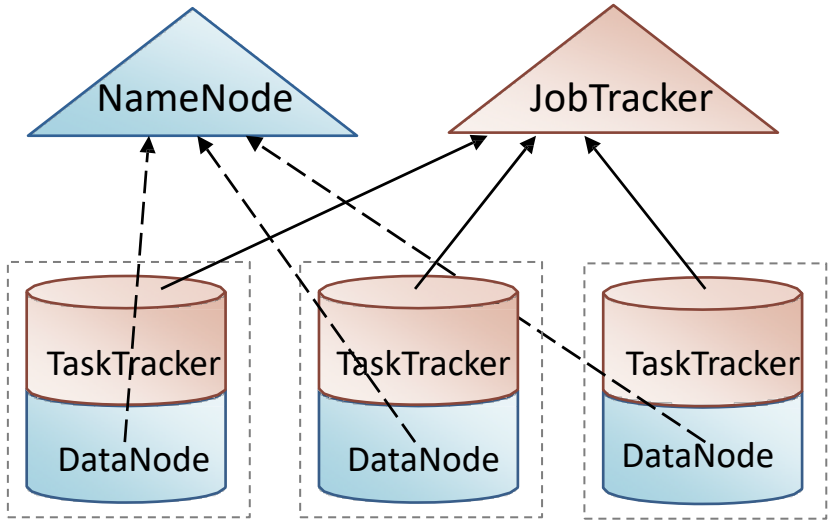
Apache Ecosystem



Apache Ecosystem

Technology	Description
MapReduce	Distributed computation framework (data processing model and execution environment)
HDFS	Distributed file system
HBase	Distributed, column-oriented database
Hive	Distributed data warehouse
Pig	Higher-level data flow language and parallel execution framework
ZooKeeper	Distributed coordination service
Avro	Data serialization system (RPC and persistent data storage)
Sqooq	Tool for bulk data transfer between structured data stores (e.g., RDBMS) and HDFS
Oozie	Complex job workflow service
YARN	(Yet Another Resource Negotiator) Resource Manager
Mahout	Machine learning and data mining library

Apache Hadoop: HDFS and YARN/MapReduce daemons



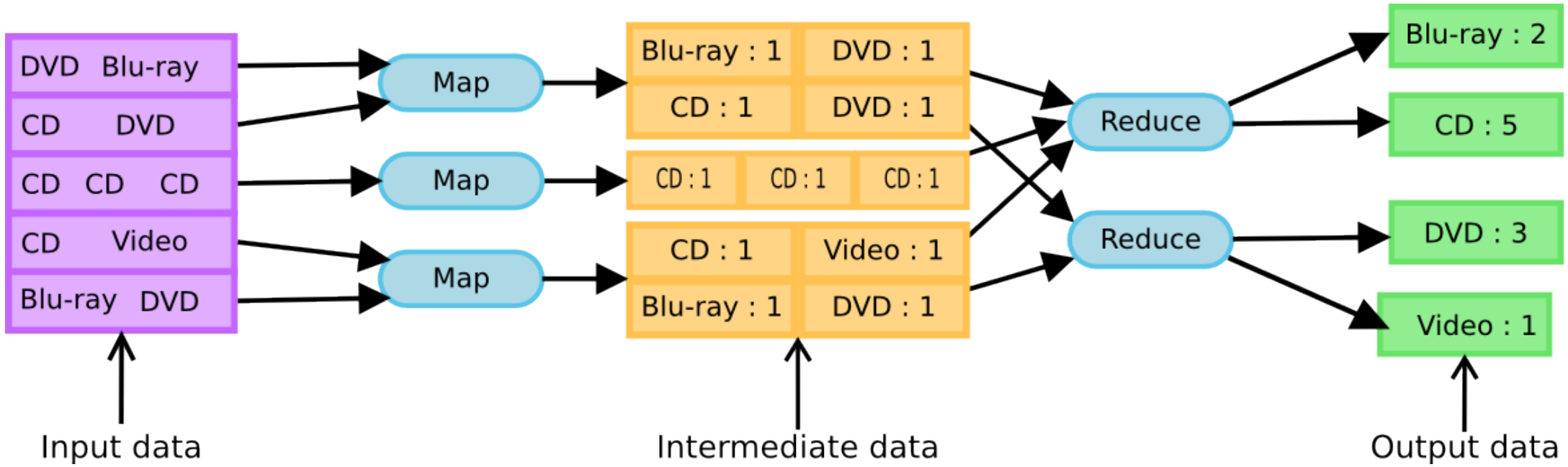
HDFS daemons

- **NameNode:** namespace and data blocks management
- **DataNode:** data replica container (it actually contains the data)

YARN/MapReduce daemons

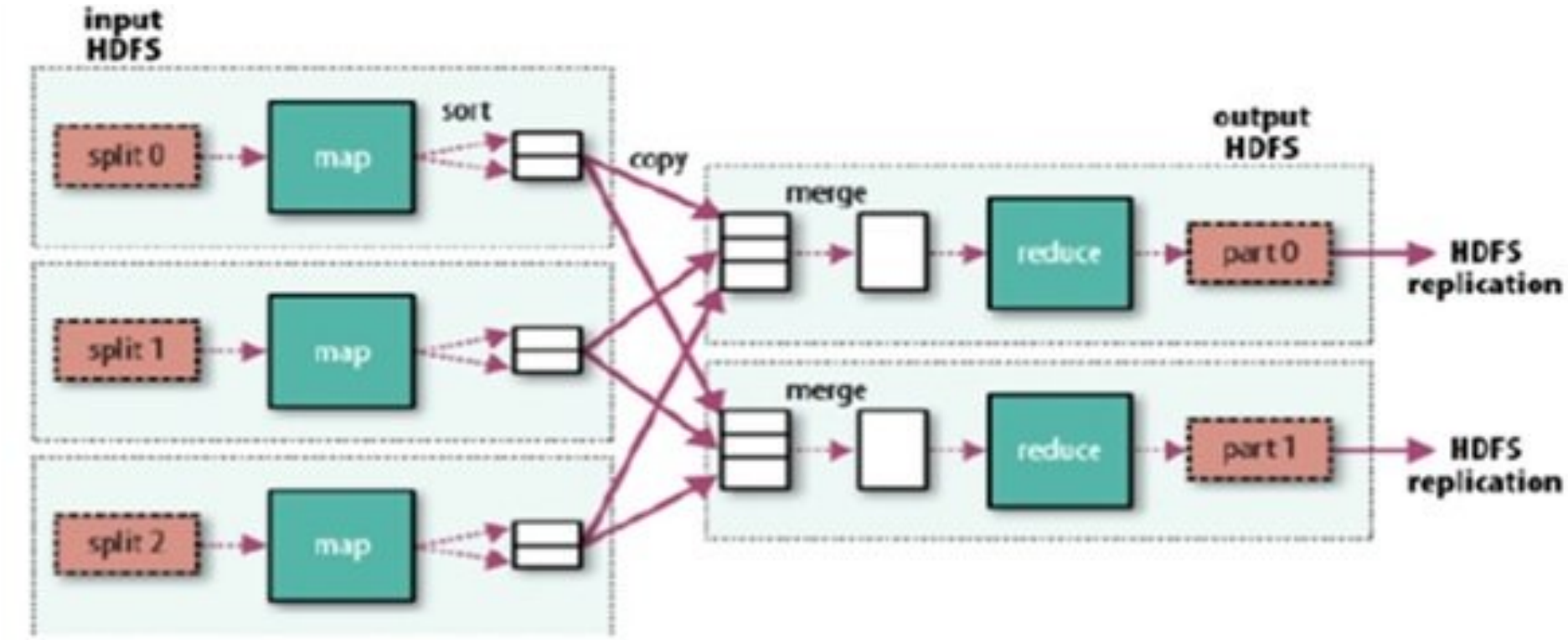
- **JobTracker:** client communication, job scheduling, resource management, lifecycle coordination
- **TaskTracker:** task execution module

MapReduce: an example



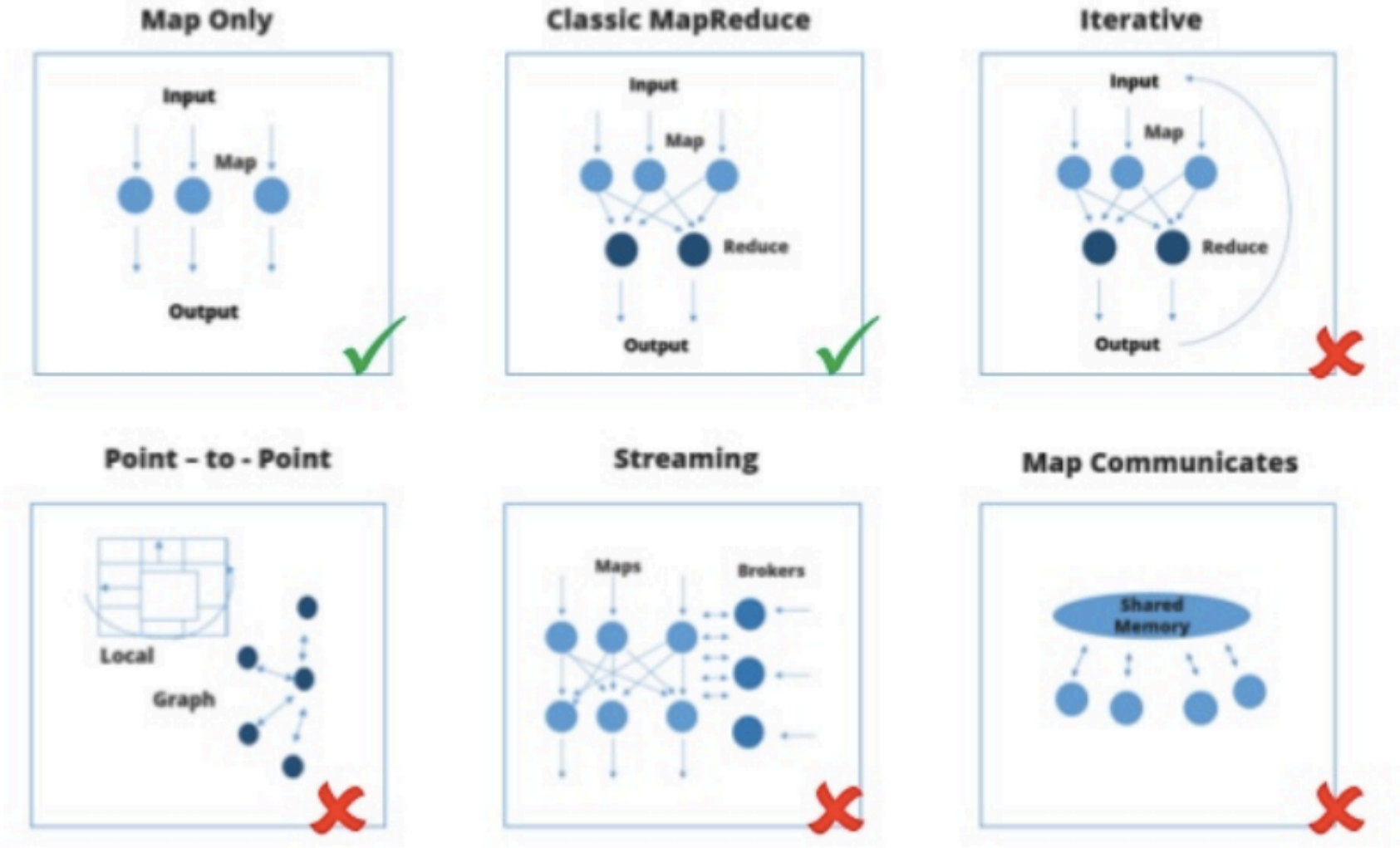
MapReduce means breaking computation in two phases,
the **map** and the **reduce** phase,
both performed in a distributed, parallel way on a cluster of computers.

MapReduce within Hadoop Framework



It represents a scalable solution that fits several use cases,
but **not all of them**

Hadoop use-cases



Hadoop limitations

- Issue with small files
- Slow processing speed (due to replication, I/O operations, serialization)
- Latency
- Lengthy line of codes
- No real time processing
- No iterative processing (need to write on the disk at each iteration)
- Security

Apache Spark

Apache Spark Comparison

Sorting benchmark for the best processing tool

The task is to sort 100 TB in a distributed and parallel way

Apache Spark Comparison

2013 Record: Hadoop

2100 machines

72 minutes

2014 Record: Spark

207 machines

23 minutes

Also sorted 1PB in 4 hours

Apache Spark

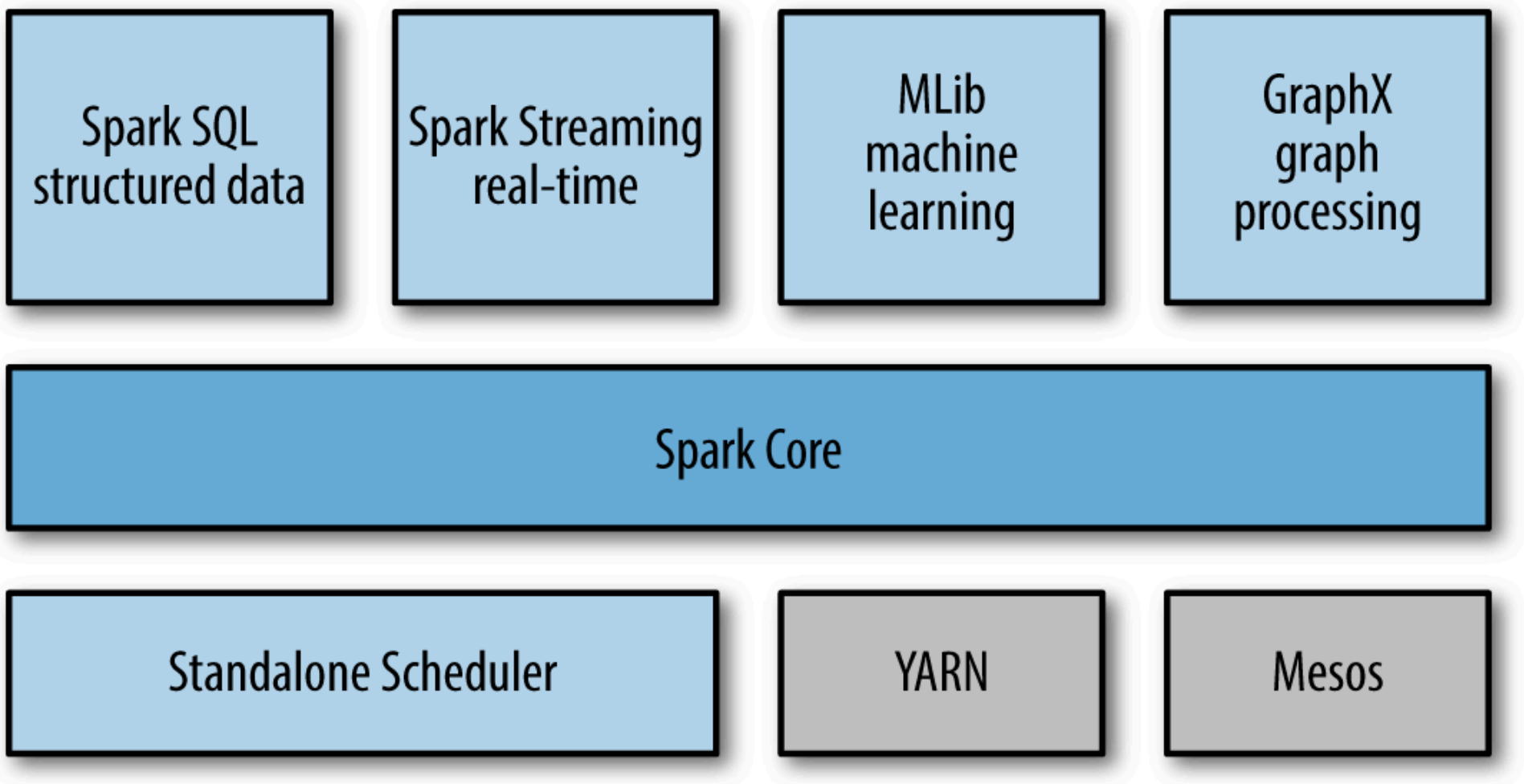
"Efficient, fast, general-purpose cluster computing framework with high-level APIs in Java, Scala, Python, and R

- Originally developed in 2009 at UC Berkley AMPLab
- Open Source since 2010, when it was named Shark

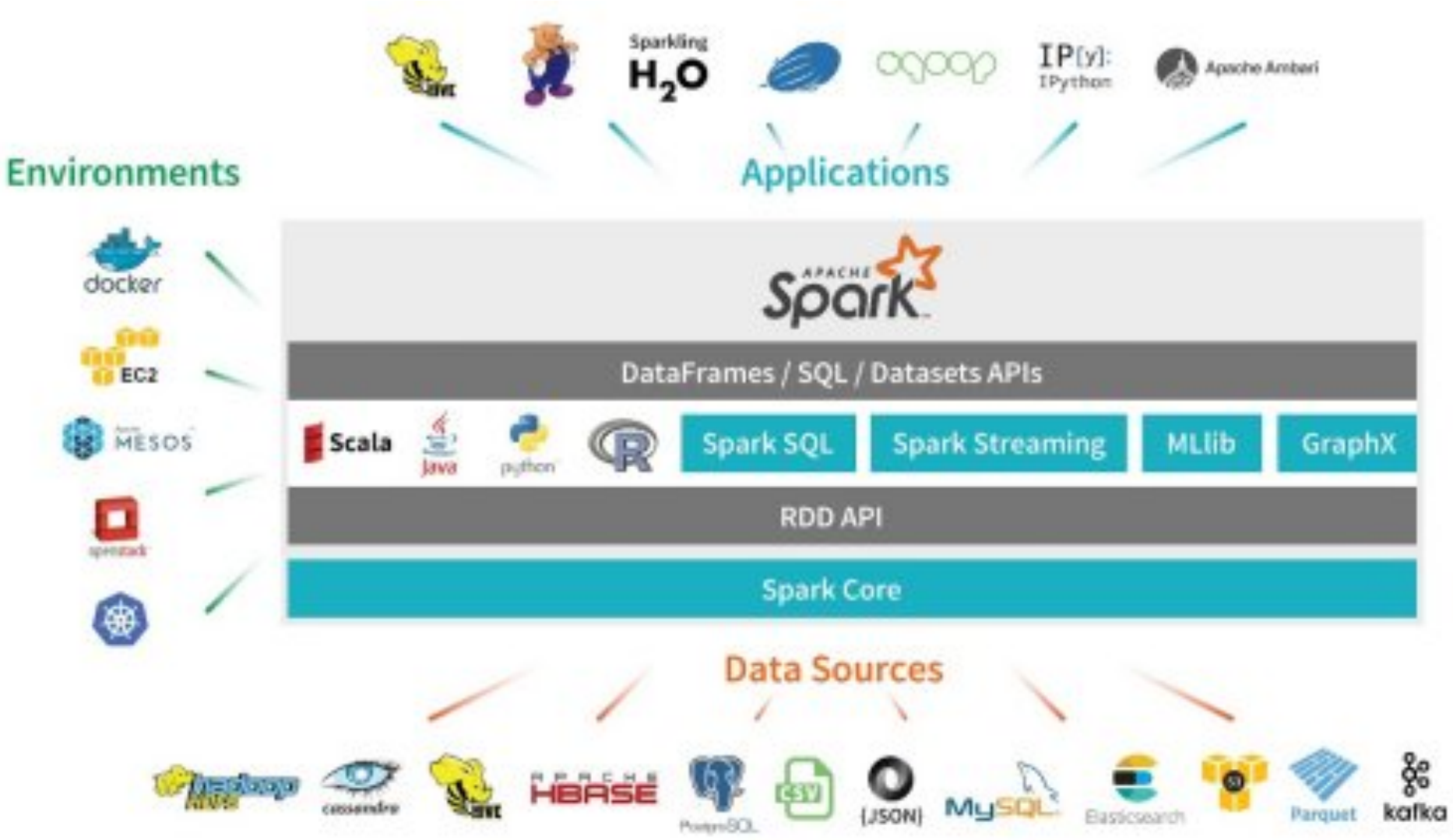
- APIs available in Java, Scala, Python, R
- Creation of an execution graph
- Enable in-memory computation

- Way less code to write

Apache Spark

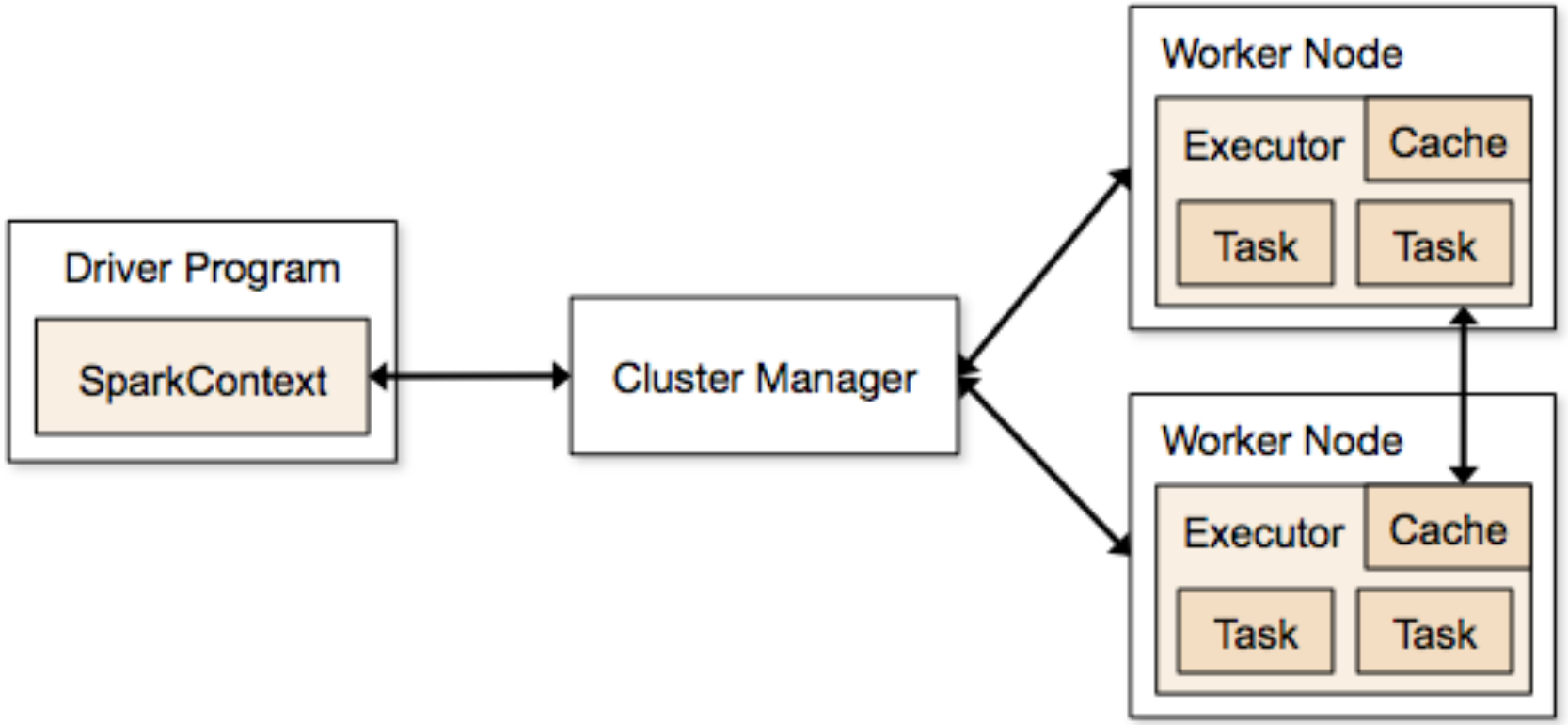


Apache Spark Ecosystem



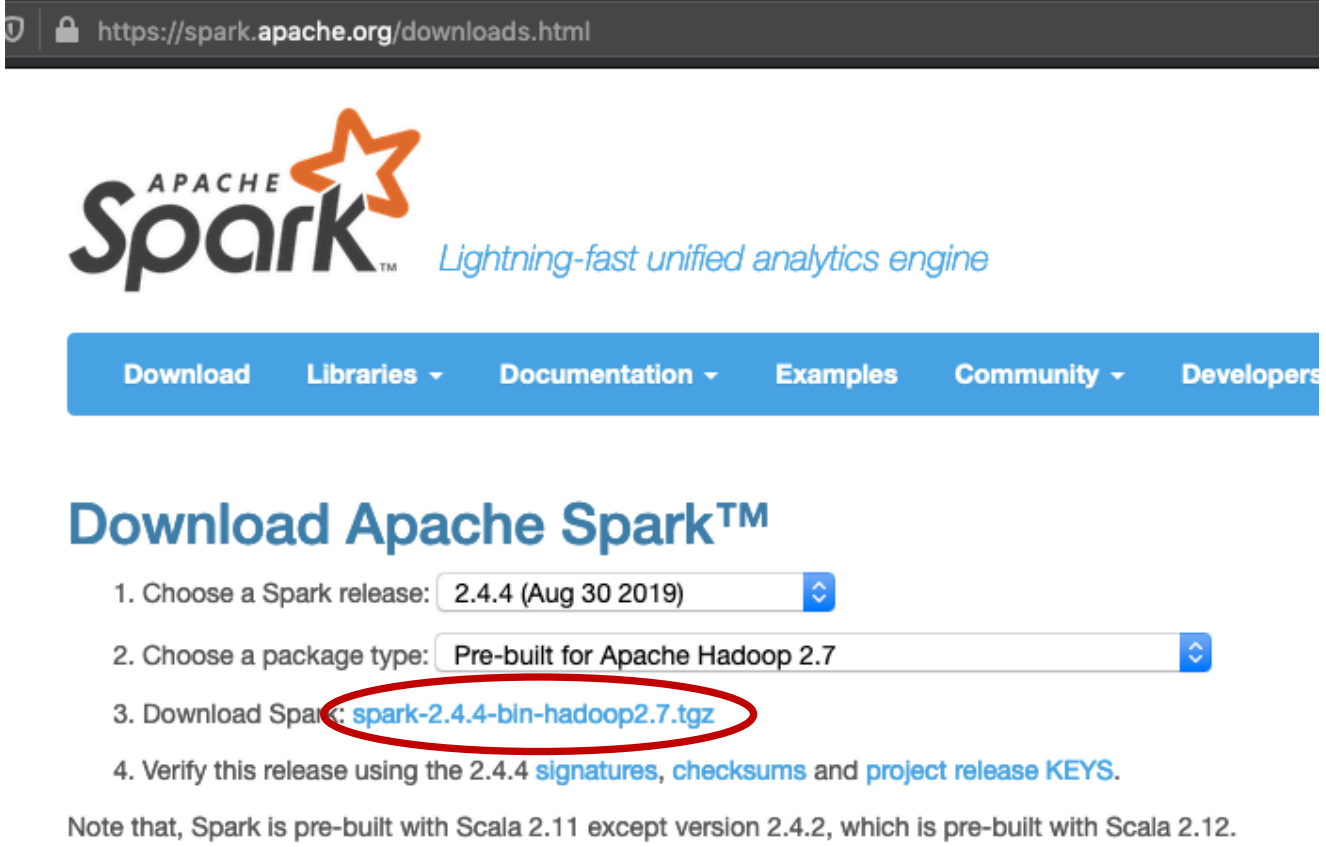
HUAWEI db Trento

Apache Spark: Computation



Spark: How to Install

- Go to the Spark website
- Choose the version you want
- Download it
- Unpack it
- Copy it to all the nodes
- Setup the slave ips in ./conf/slaves
- Setup the master ip and the other configs to ./conf/spark-defaults.conf
- Start Spark by running ./sbin/start-all.sh



Spark: Folders and Content

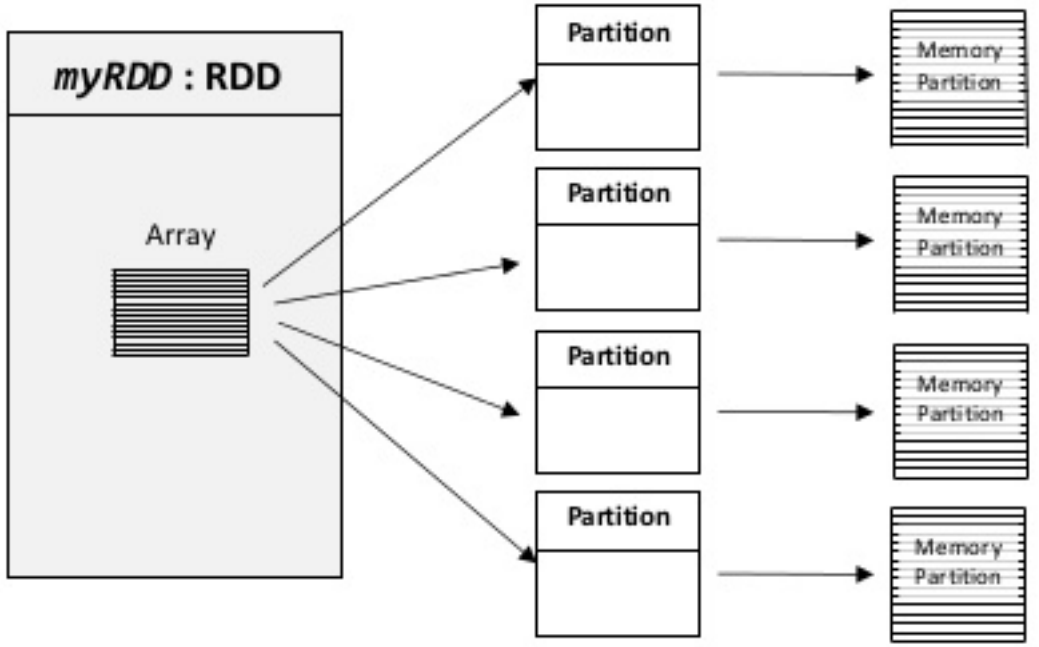
- R: packages and source for R
- bin: shell related scripts
- conf: folder containing all the configuration files
- data: files for running the examples
- examples: code for the examples
- jars: contains all the external libraries needed by Spark
- kubernetes: configurations and code related to docker and kubernetes deployment
- python: packages and source for python
- sbin: scripts to start and stop the distributed framework
- yarn: contains the jar needed by YARN

Spark Core: RDDs

Distributed collection of objects that can be cached in memory across cluster nodes

- Resilient Distributed Dataset

- Immutable
- Resilient
- Distributed
- Lazily evaluated
- Cacheable/Persistent
- Fault-tolerant



Spark Core: Actions and Transformations

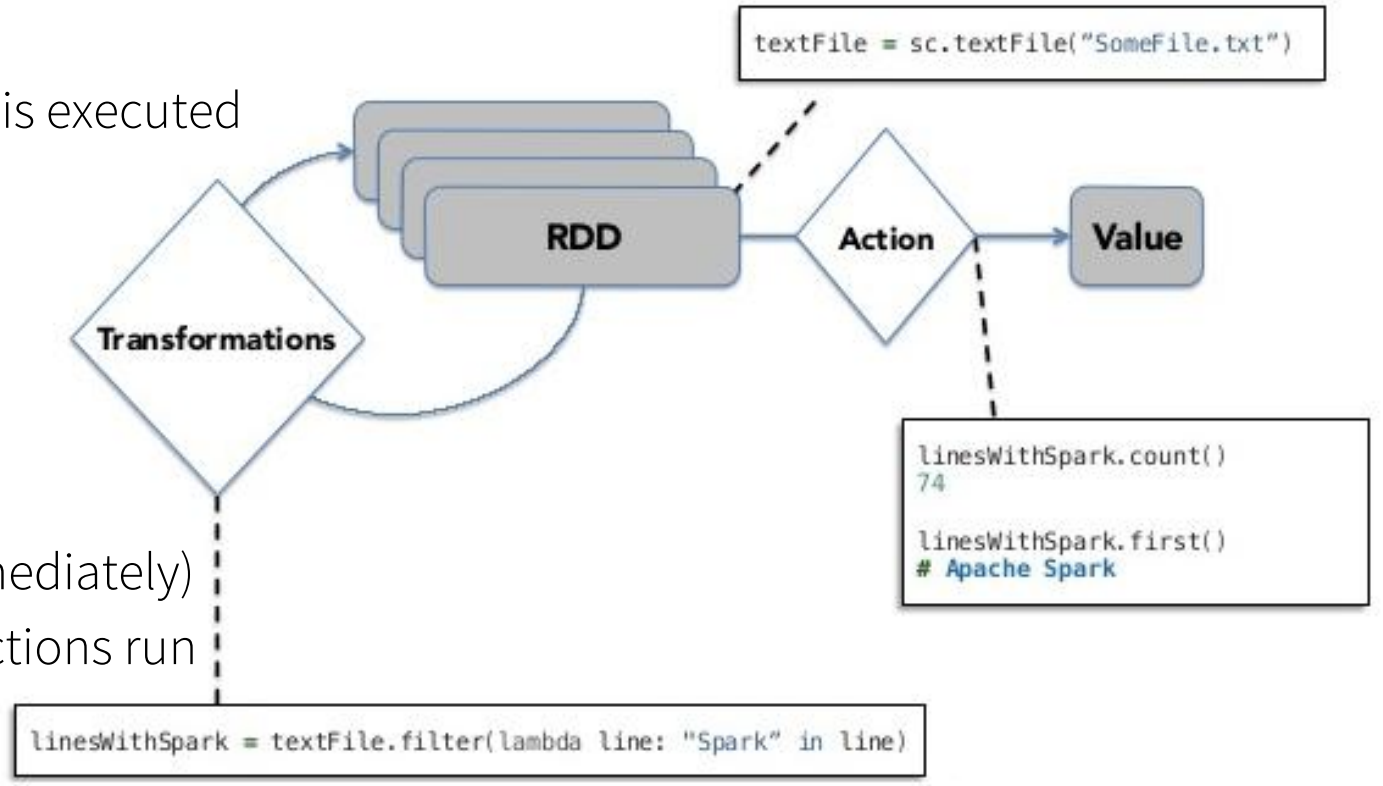
Lazily evaluation due to a dependency chain of RDDs. DAG allows for running consistently more complex operations

Actions

- Return a value
- Result into a DAG of operations
- DAG is compiled into stages where each stage is executed as a series of task
- Task: fundamental unit of work

Transformations

- Return pointers to a new RDD
- Transformations are lazy (not computed immediately)
- Transformed RDDs gets recomputed when actions run on it
- RDD can be persisted in memory or disk



Spark Core: Actions and Transformations

- Transformations create a new dataset from an existing one (*map* is a transformation)
- Actions return a value to the driver program after running a computation on the dataset (*reduce* is an action)
- Transformations are lazy, it means that they are only computed when an action requires a result to be returned to the driver program

```
x = rdd.map(…).map(…)       No computation
print(x.reduce(…))          The computation starts from rdd
print(x.map(…).count())     The computation starts from rdd
```

Spark Core: Lazy evaluation Drawback

Drawbacks

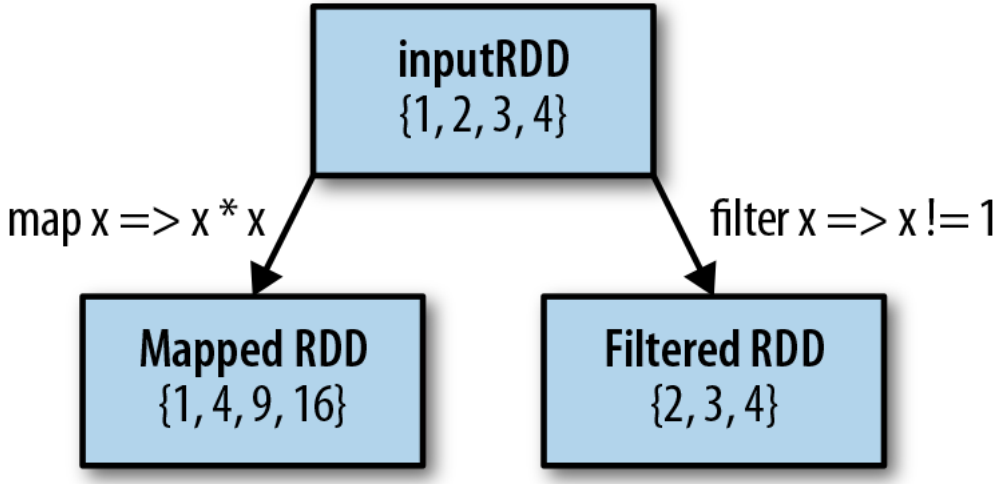
- Many partitions are computed multiple times
- Longer execution time
- Seems so nice from the outside

Solution

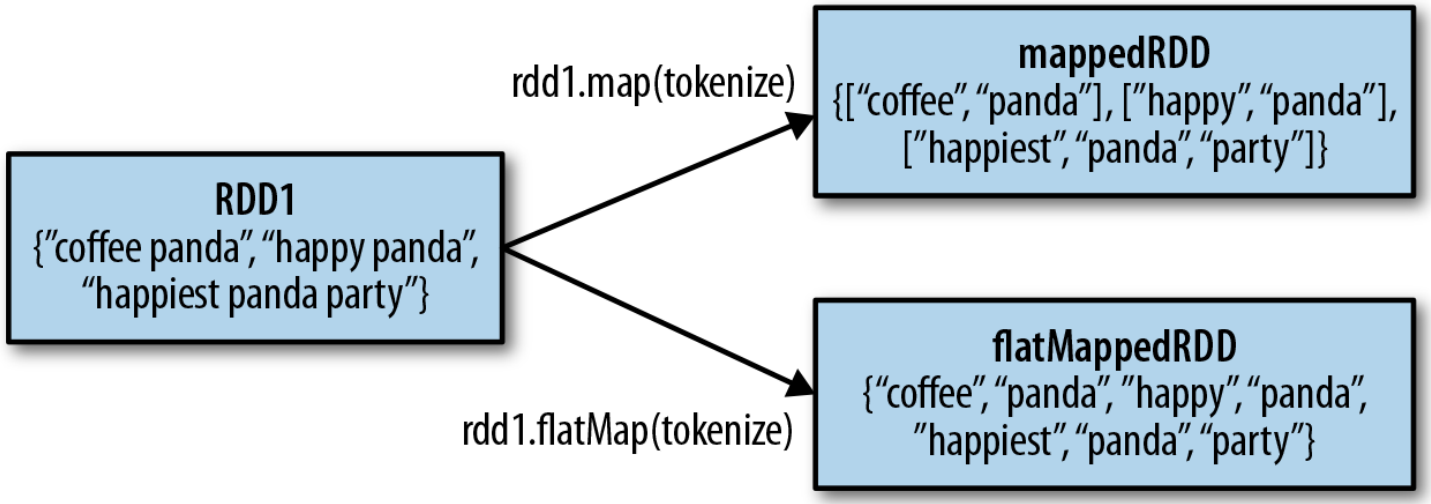
- RDD Persistence
- Caching RDDs in memory across different operations
- Significant speed up (at least 10x)
- Different level of persistence (Main memory, local disk, mixed)

Spark Core: Transformations

Map & Filter

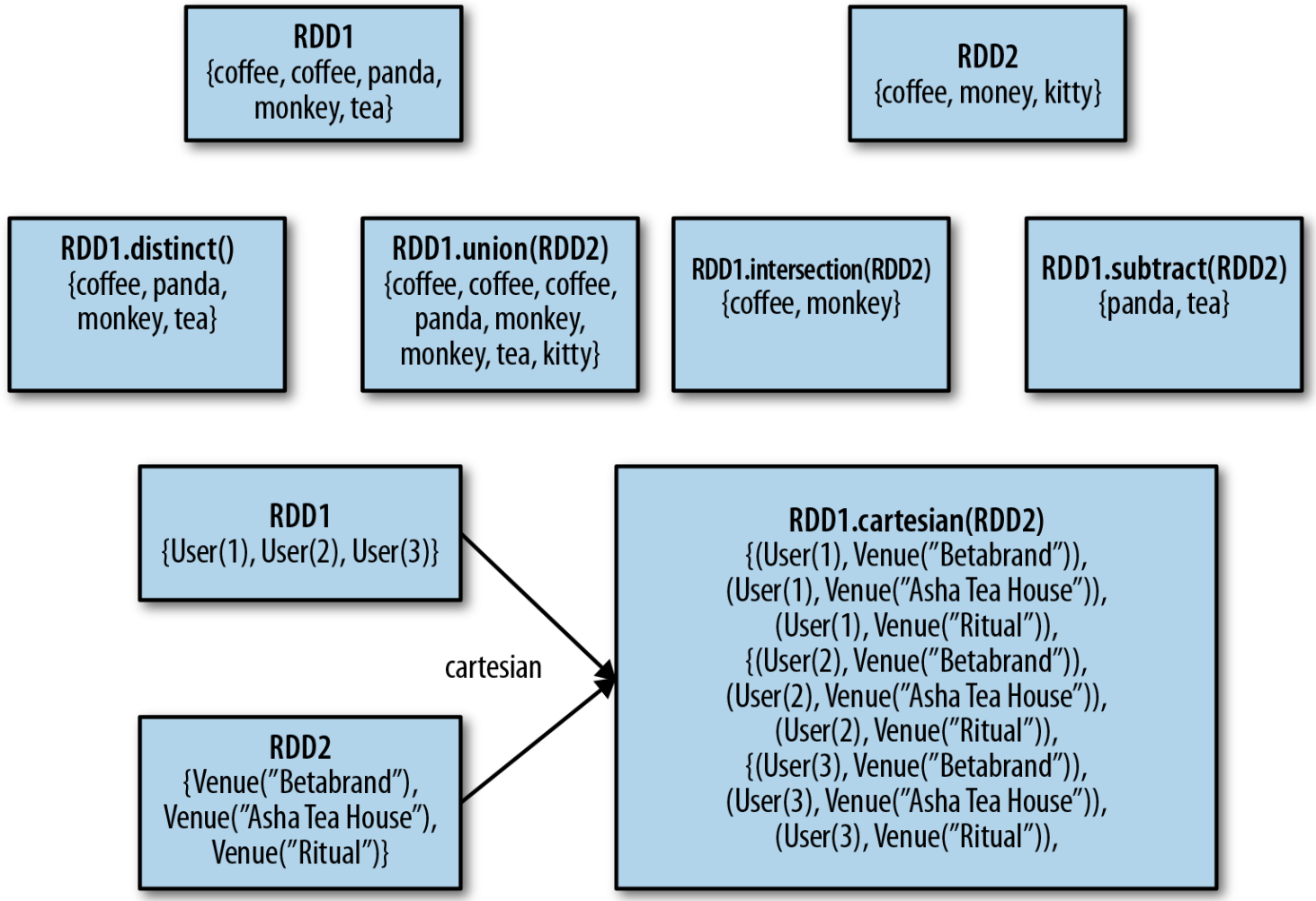


Map vs. FlatMap



Spark Core: Transformations

"List" like operations



Spark Core: Transformations

Resilient RDD containing {1, 2, 3, 3}

Operation	Purpose	Example	Results
collect()	Return all elements	rdd.collect()	{1, 2, 3, 3}
count()	Count all elements	rdd.count()	4
countByValue()	Count elements by value	rdd.countByValue()	{(1, 1), (2, 1), (3, 2)}
take(num)	Return a number of elements equal to num	rdd.take(2)	{1, 2}
top(num)	Return the top N values	rdd.top(2)	{3, 3}
reduce(func)	Reduce	rdd.reduce((x, y) => x + y)	9

Spark Core: Load data into RDDs

Parallelized Collections

- Existing collections from your driver program
- Variable, Lists, Sets

External Datasets

- S3
- HDFS
- Cassandra, MongoDB, and others
- MySQL, Postegres, and others

Spark Core: RDDs and Partitions

Partitions

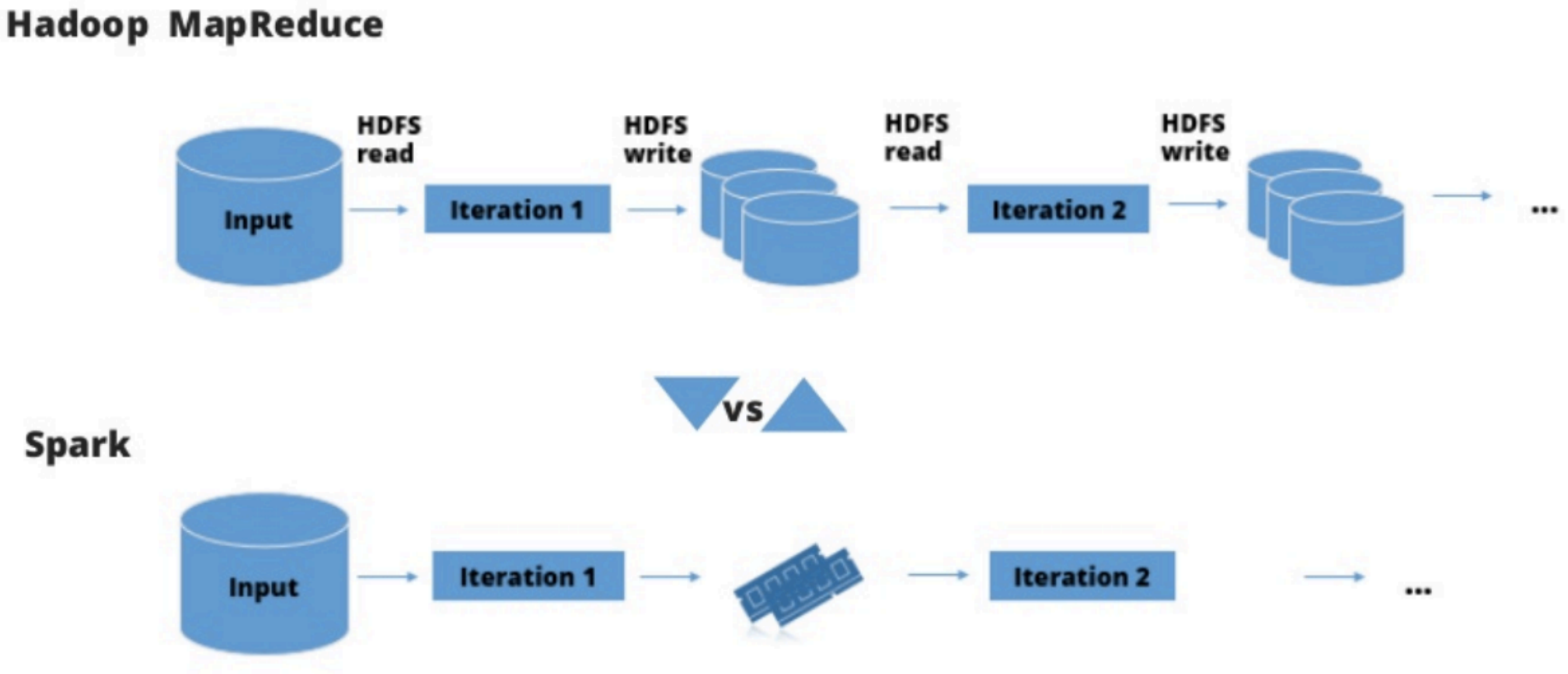
- RDDs partitioned through different nodes
- Number of partitions is an important value
- Typically 2-4 partitions per CPU
- Done automatically by default

Transformations

- By default applied on the RDD
- You can apply transformations on partitions
- User defined partitions functions
- Transformation (*map* vs. *mapPartition*)

Spark: The advantages of using memory

Data sharing in Hadoop is slow due to replication, serialization, and disk I/O



Spark core: Word Count

```java
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

```scala
val f = sc.textFile(inputPath)
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

Spark core: Average computation


```
private IntWritable one =
  new IntWritable(1)
private IntWritable output =
  new IntWritable()
proctected void map(
    LongWritable key,
    Text value,
    Context context) {
  String[] fields = value.split("\t")
  output.set(Integer.parseInt(fields[1]))
  context.write(one, output)
}

IntWritable one = new IntWritable(1)
DoubleWritable average = new DoubleWritable()

protected void reduce(
    IntWritable key,
    Iterable<IntWritable> values,
    Context context) {
  int sum = 0
  int count = 0
  for(IntWritable value : values) {
     sum += value.get()
     count++
    }
  average.set(sum / (double) count)
  context.Write(key, average)
}
```


```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [x.[1], 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

Spark SQL: Why Structured?

Many datasets have a fixed structure (JSON, CSV, DATABASES)

RDDs : Imperative API

MAP and REDUCE operations specify how but not what

Single RDD operations are not optimizable

```
exampleRDD.map(lambda x: x *x);
```

The order of the operations matters

```
RDD1.join(RDD2).filter(…);
```

Spark SQL: DataFrame and Dataset

Two different notions of structured collections:

- DataFrames
- Datasets

- Distributed table-like collections
- Well defined rows and columns

- Immutable and lazy evaluated plans
- Conceptually similar to RDDs
- But more complex and optimizable

Spark SQL: Column and Row

Column

- Simply type: Integer, String, custom type, …
- Complex type: array, map, …
- Can contain null values
- Transformations

Row

- Record of data
- Type Row
- Different sources: SQL, RDDs, other data sources or manually created

Spark SQL: Available Types

Data type	*Value type in Scala*	*API to access or create a data type*
ByteType	Byte	ByteType
ShortType	Short	ShortType
IntegerType	Type	IntegerType
LongType	Long	LongType
FloatType	Float	FloatType
DoubleType	Double	DoubleType
DecimalType	java.math.BigDecimal	DecimalType
StringType	String	StringType
BinaryType	Array [Byte]	BinaryType
BooleanType	Boolean	BooleanType
TimestampType	java.sql.Timestamp	TimestampType
DateType	java.sql.Date	DateType
ArrayType	scala.collection.Seq	ArrayType(elementType, [containsNull]) Note: The default value of containsNull is true.
MapType	scala.collection.Map	MapType (keyType, valueType, [valuecontainsNull]) Note: The default value of valuecontainsNull is true.
StructType	org.apache.spark.sql.Row	StructType (fields) Note: fields is a Seq of StructFields. Also, two fields with the same name are not allowed.
Struct Field	The value type in Scala of the data type of this field (For example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, [nullable]) Note: The default value of nullable is true.

Spark SQL: DataFrame vs. Dataset

Two more APIs

- ‘untyped’ DataFrames
- ‘typed’ Dataset

What does it mean ‘untyped’?

- There is a type
- It’s maintained by Spark
- check those types at runtime

DataFrame consists of a series of records

- Row type
- Columns

Schemas define the name and the type of each columns

Dataframe are partitioned on different machines

Datasets with types conform at compile time

Datasets are available on JVM based languages: Scala and Java

DataFrames are simply Datasets of Types Row

Spark SQL: DataFrame vs. Dataset

```
# in Python
myRange = spark.range(1000).toDF("number")
```

Transformation

```
divisBy2 = myRange.where("number % 2 = 0")
```

Action

```
divisBy2.count()
```

Spark SQL: Structured API Execution

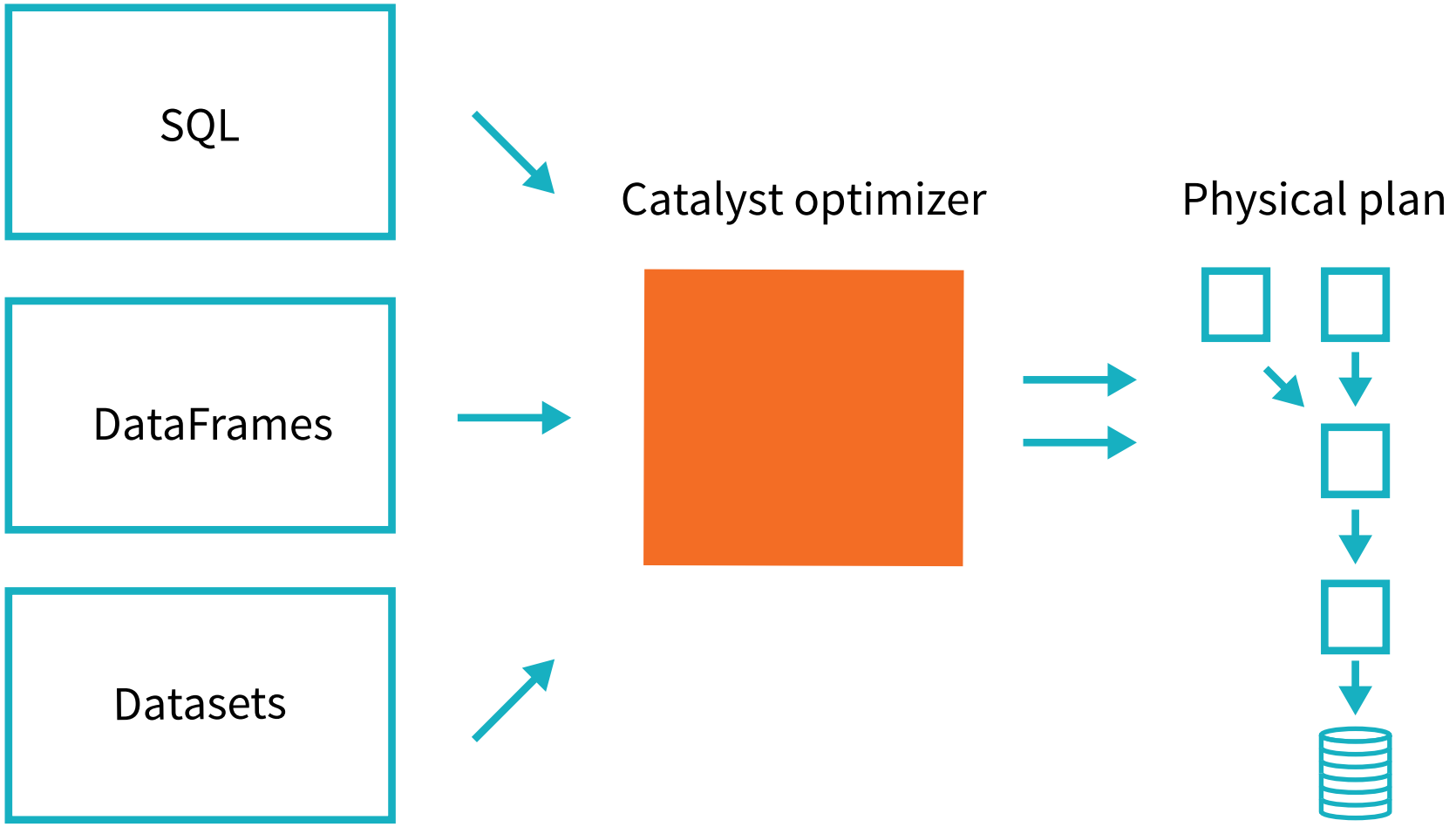
1. Write DataFrame/Dataset/SQL Code

2. If valid code, Spark converts this to a Logical Plan

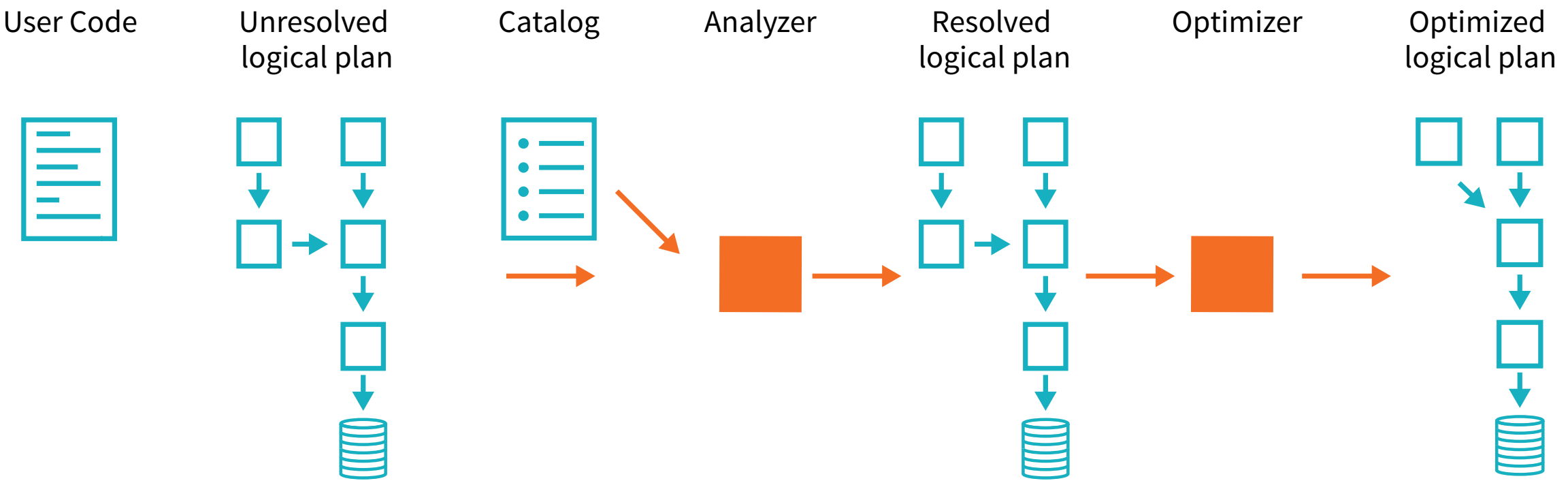
3. Spark transforms this Logical plan to a Physical Plan, checking for optimization along the way

4. Spark then executes this Physical Plan (RDD manipulations) on the cluster

Spark SQL: Structured API Execution

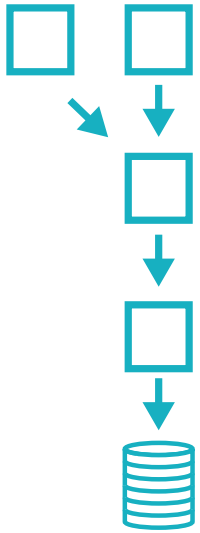


Spark SQL: Structured API Execution

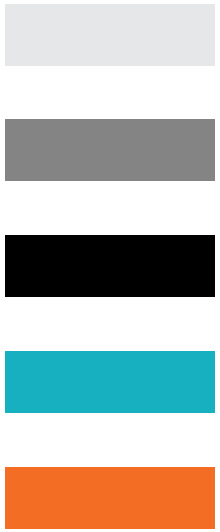


Spark SQL: Structured API Execution

Optimized logical plan



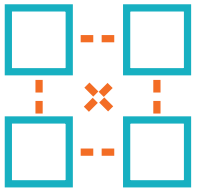
Physical plans



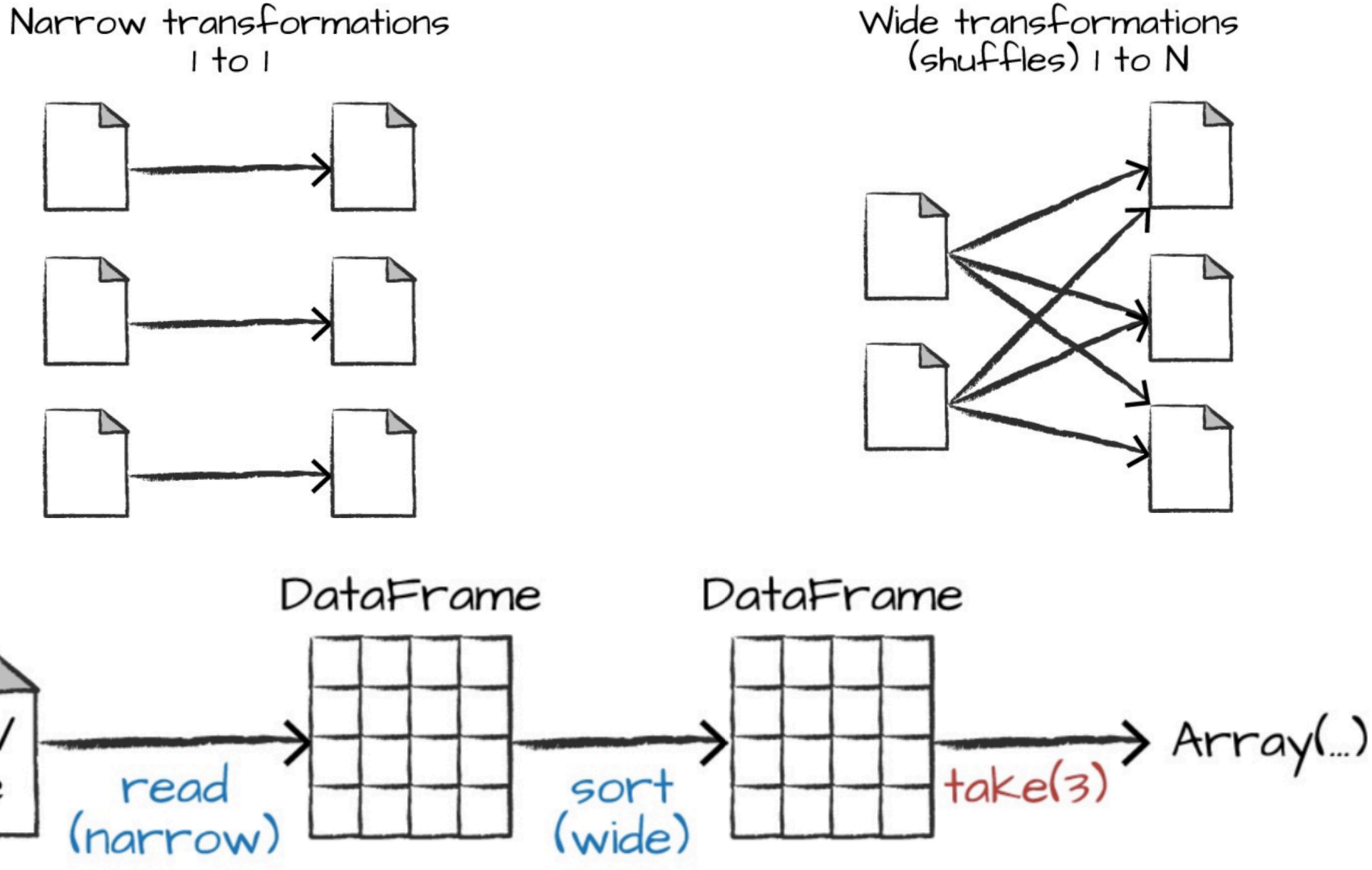
Cost model

Best physical plan

Executed on cluster



Spark SQL: Wide and Narrow Transformations



Basic structured operations: SCHEMA

```
# in Python
flightData2015 = spark\
  .read\
  .option("inferSchema", "true")\
  .option("header", "true")\
  .csv("/data/flight-data/csv/2015-summary.csv")
```

RETURNS

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

Basic structured operations: SCHEMA

```
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")
```

Columns

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

```
df.col("count")
```

Expressions

An expression is a set of transformations

```
from pyspark.sql.functions import col, expr
```

Columns as expressions

`expr("someCol - 5")` SAME as `col("someCol") - 5` SAME as `expr("someCol") - 5`

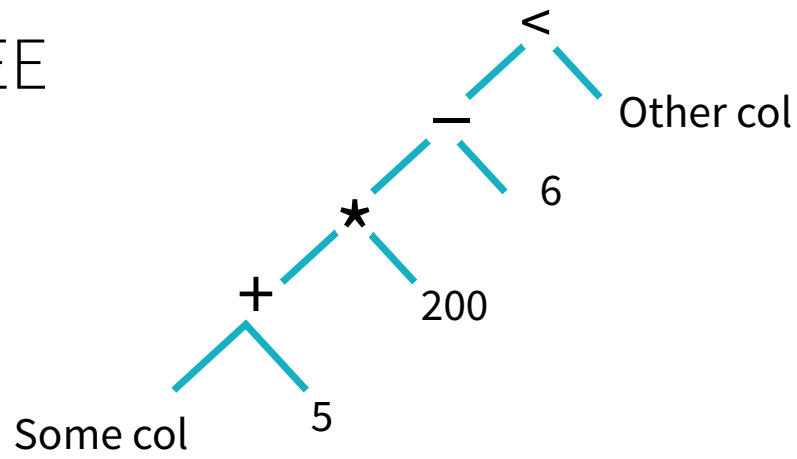
Not so easy at the beginning but REMEMBER:

1. Columns are just expressions
2. Columns and transformations of those column compile to the same logical plan as parsed expressions.

Expression example

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

LOGICAL TREE



CODE:

```
from pyspark.sql.functions import expr

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

Dataframe Operations

Retrieve columns

```
spark.read.format("json")
.load("/mnt/defg/flight-data/json/2015-summary.json")
.columns
```

Create rows

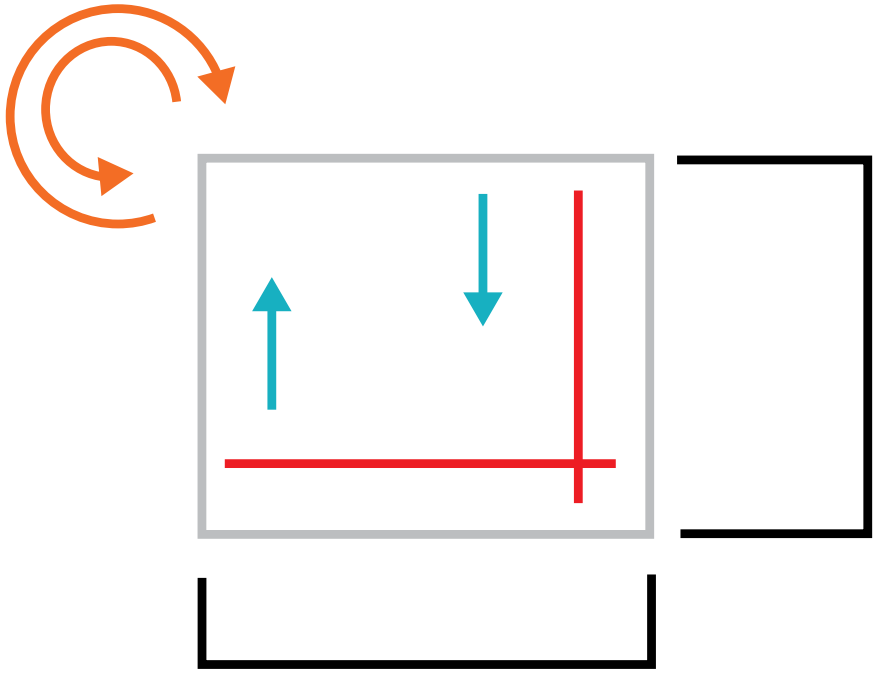
```
from pyspark.sql import Row
myRow = Row("Hello", None, 1, False)
```

Accessing rows

```
%scala
myRow(0) // type Any
myRow(0).asInstanceOf[String] // String
myRow.getString(0) // String
myRow.getInt(2) // String
```

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

DataFrame Transformations



- Remove columns or rows
- Row into a column or column into a row
- Add rows or columns
- Sort data by values in rows

Create DataFrames From File

```
df = spark.read.format("json")\
   .load("/mnt/defg/flight-data/json/2015-summary.json")

df.createOrReplaceTempView("dfTable")
```

Create Dataframe by hand

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

Select and SelectExpr (1)

```sql
-- in SQL
SELECT * FROM dataFrameTable
SELECT columnName FROM dataFrameTable
SELECT columnName * 10, otherColumn, someOtherCol as c
FROM dataFrameTable
```

```python
# in Python
sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")

dataFrameWay = flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .count()
```

Select and SelectExpr (1)

```
spark.sql("SELECT max(count) from flight_data_2015").take(1)
```

```
# in Python
from pyspark.sql.functions import max

flightData2015.select(max("count")).take(1)
```

Select and SelectExpr (2)

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

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

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

Select

```
# in Python
maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")
```

```
+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```

```
# in Python
from pyspark.sql.functions import desc

flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .sum("count")\
  .withColumnRenamed("sum(count)", "destination_total")\
  .sort(desc("destination_total"))\
  .limit(5)\
  .show()
```

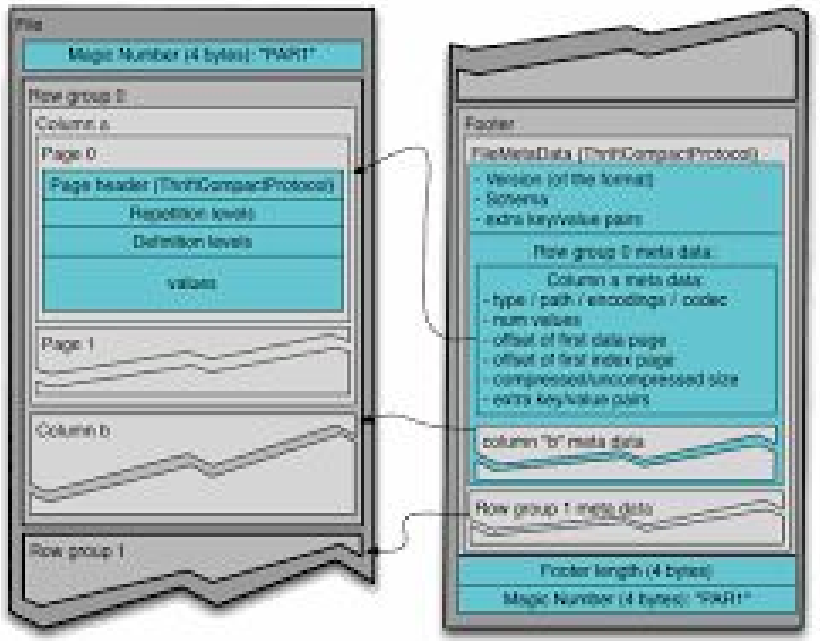
SparkSession

Provides methods and attribute for SQL

- read – create un DataFrame from a supported source
- sql(STRING query) – query data throught SQL
- catalog – contains information about database and table
- createDataFrame() – create a DataFrame from an rdd given a defined schema

Parquet

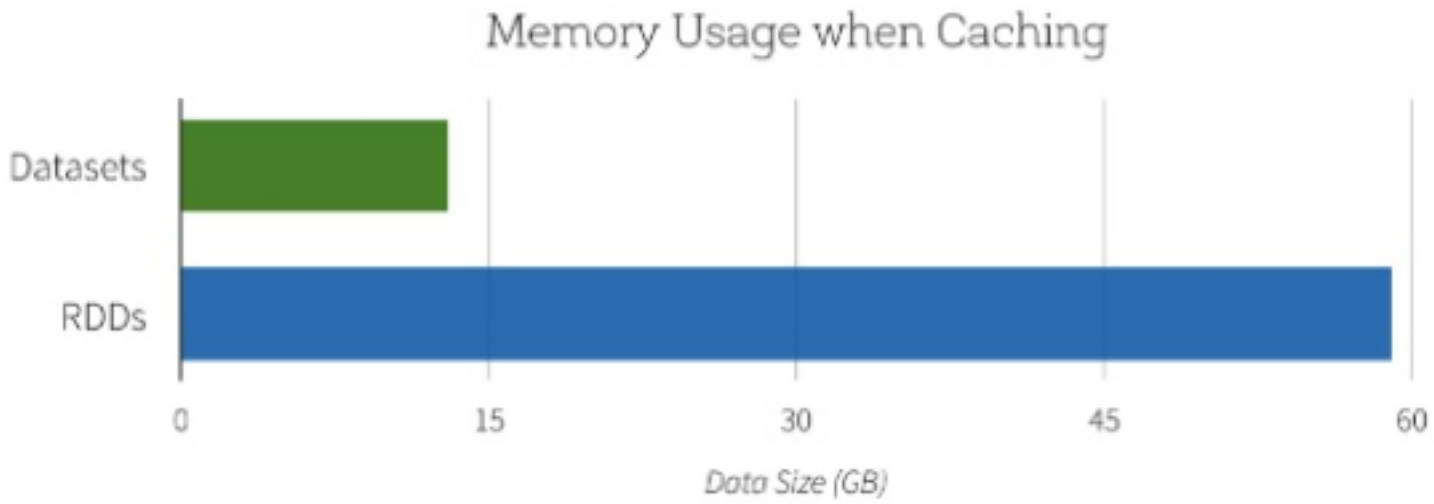
- Column storage
- Each .parquet file is a directory
- Supported by Spark
- Optimized for Big Data



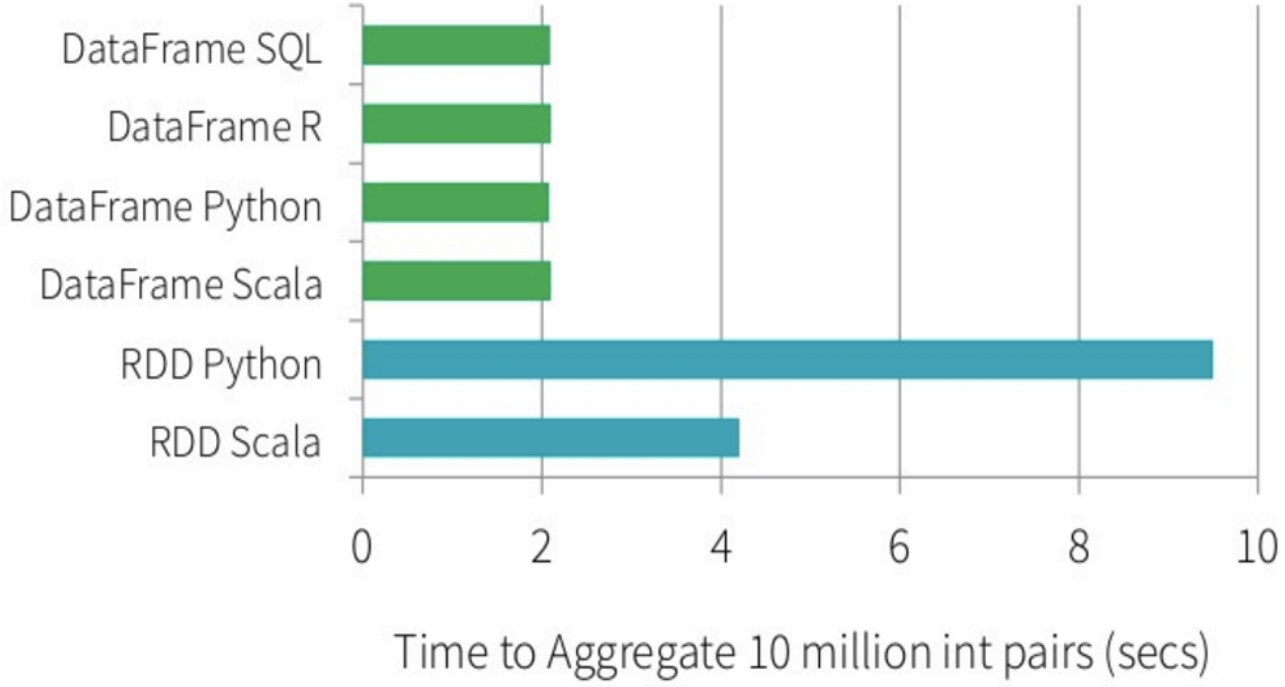
Structure

- A STRUCTURE limits what we can express
- But it can accommodate the vast majority of computations
- Limiting the space of what can be expressed enables optimizations!

Performance (1)



Performance (2)



ML vs Big Data

Machine Learning	Big Data Tools
Iterative computation	Single pass computation
Maintains state between stages	Stateless
CPU intensive	Data intensive
Vector/Matrix based or multiple rows/cols at a time	Single row/column at a time

ML in Hadoop

- In Hadoop each iteration of ML translates into a single Map/Reduce job
- Each of these jobs need to store data in HDFS, which leads to an important overhead
- Keeping state across jobs is not directly available in M/R
- Constant fight between quality of results vs performance

ML in Spark

- Spark is first general purpose big data processing engine built for ML from day one
- The initial design in Spark was driven by ML optimization
  - Caching - for running on data multiple times
  - Accumulator - to keep state across multiple iterations in memory
  - Good support for CPU intensive tasks with laziness
- One of the examples in Spark in the first version was about ML

ML Terminology

Training Set

Set of data used to teach the machine. In supervised learning both input vector and output will be available

Learning algorithm

Algorithm that consumes the training set to infer relation between input vectors that optimizes for known output labels

Model

Learnt function of input parameters

Algorithm Coverage

- Classification
  - Logistic regression
  - Naive Bayes
  - Streaming logistic regression
  - Linear SVMs
  - Decision trees
  - Random forests
  - Gradient-boosted trees
  - Multilayer perceptron
- Regression
  - Ordinary least squares
  - Ridge regression
  - Lasso
  - Isotonic regression
  - Decision trees
  - Random forests
  - Gradient-boosted trees
  - Streaming linear methods
  - **Generalized Linear Models**
- Frequent itemsets
  - FP-growth
  - PrefixSpan

Recommendation
- Alternating Least Squares

Feature extraction & selection
- Word2Vec
- Chi-Squared selection
- Hashing term frequency
- Inverse document frequency
- Normalizer
- Standard scaler
- Tokenizer
- One-Hot Encoder
- StringIndexer
- VectorIndexer
- VectorAssembler
- Binarizer
- Bucketizer
- ElementwiseProduct
- PolynomialExpansion
- **Quantile discretizer**
- **SQL transformer**

Model import/export

Pipelines

Clustering
- Gaussian mixture models
- K-Means
- Streaming K-Means
- Latent Dirichlet Allocation
- Power Iteration Clustering
- **Bisecting K-Means**

Statistics
- Pearson correlation
- Spearman correlation
- Online summarization
- Chi-squared test
- Kernel density estimation
- Kolmogorov–Smirnov test
- **Online hypothesis testing**
- **Survival analysis**

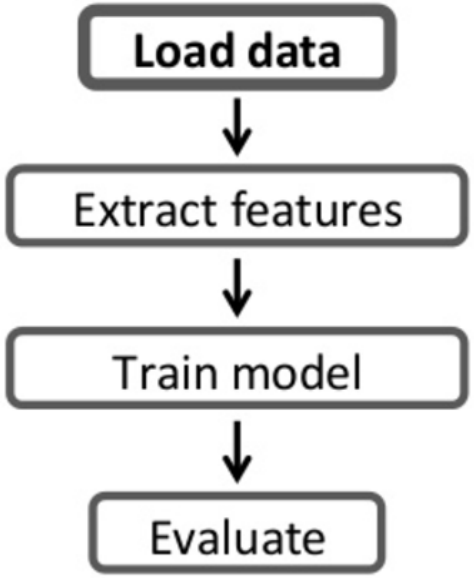
Linear algebra
- Local dense & sparse vectors & matrices
- **Normal equation for least squares**
- Distributed matrices
  - Block-partitioned matrix
  - Row matrix
  - Indexed row matrix
  - Coordinate matrix
- Matrix decompositions

databricks

List based on Spark 2.0

HUAWEI db Trento

Data Loading



Data sources for DataFrames

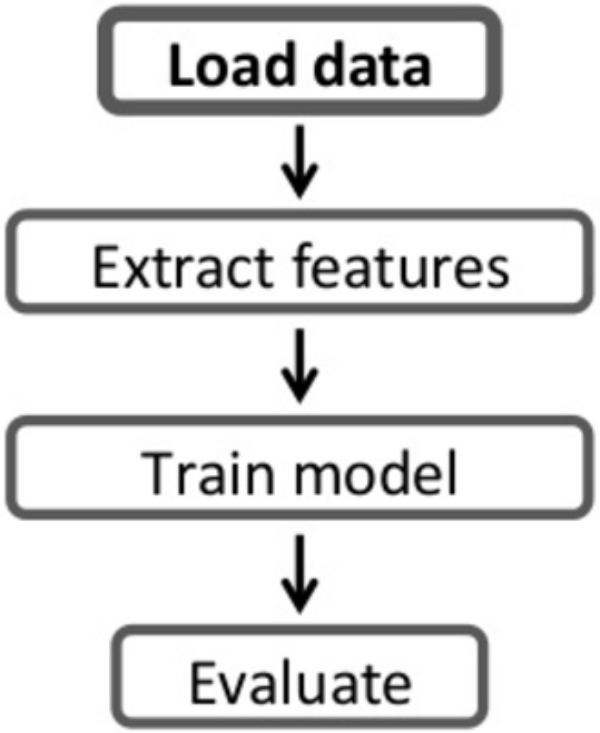
built-in

Parquet
JDBC
{ JSON }
PostgreSQL
HIVE
MySQL
HDFS
amazon S3
H2

external

AVRO
CSV
dBase
APACHE HBASE
elasticsearch.
cassandra
amazon Amazon Redshift
and more ...

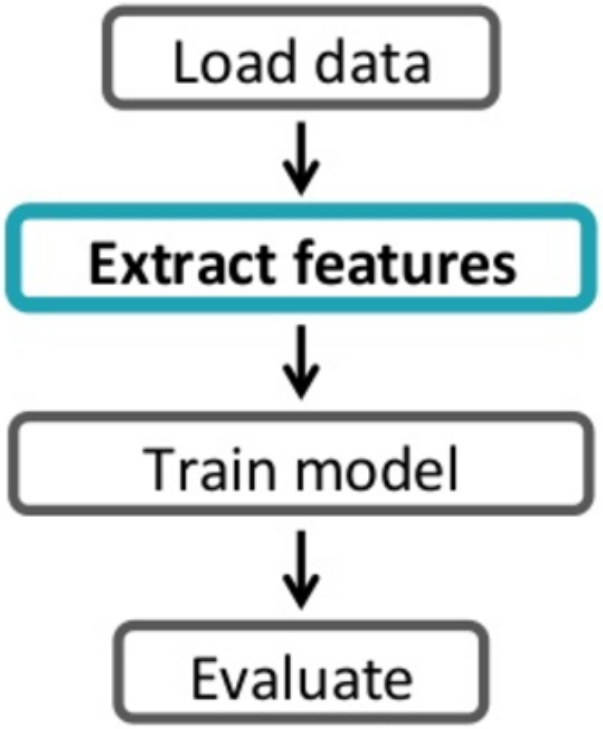
Data Loading



Data Schema:

- label: Int
- text: String

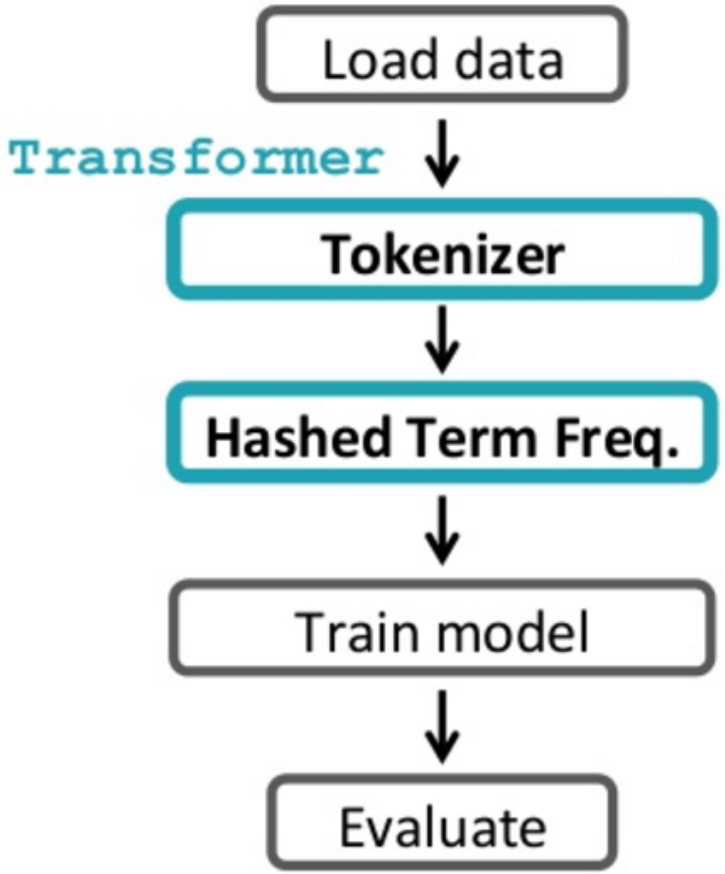
Feature Extraction



Data Schema:

- label: Int
- text: String

Feature Extraction

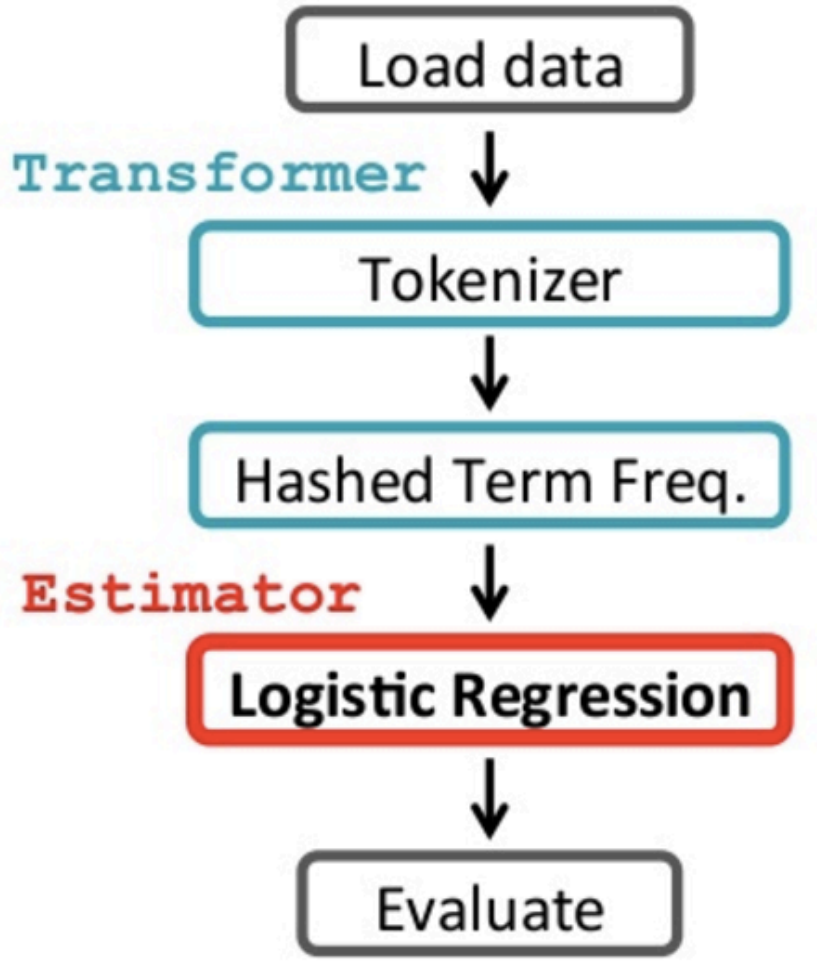


Data Schema:

- label: Int
- text: String

- words: Seq[String]
- features: Vector

Model Training



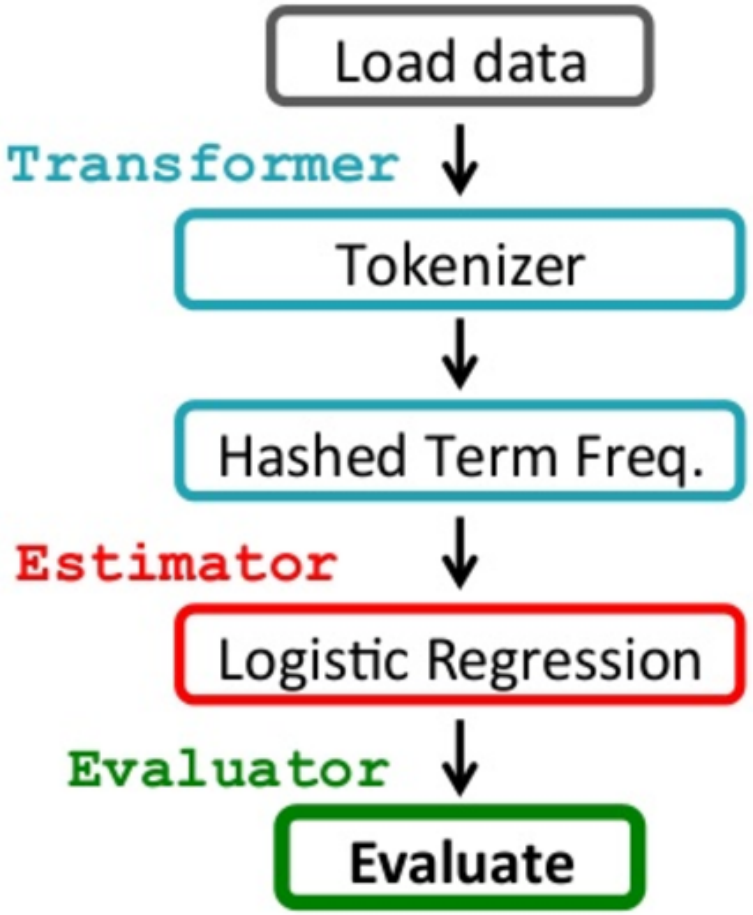
Data Schema:

- label: Int
- text: String

- words: Seq[String]
- features: Vector

- prediction: Int

Model Evaluation



Data Schema:

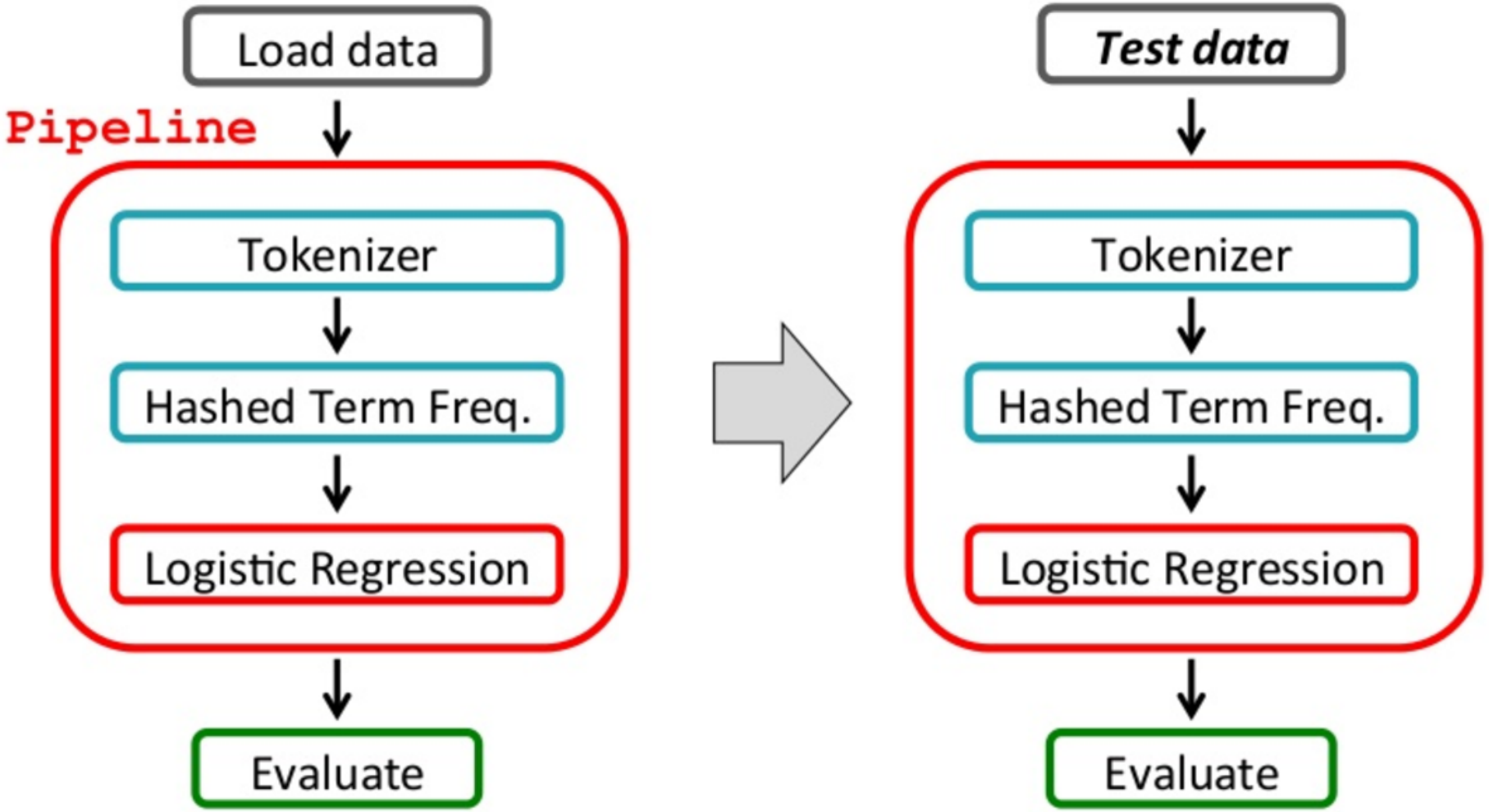
- label: Int
- text: String

- words: Seq[String]
- features: Vector

- prediction: Int

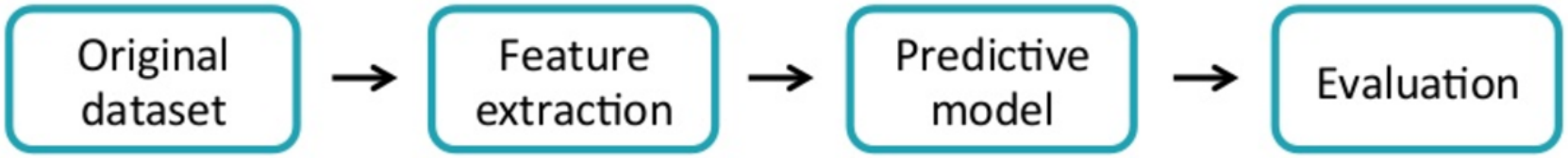
By default each step adds a column

ML Pipeline



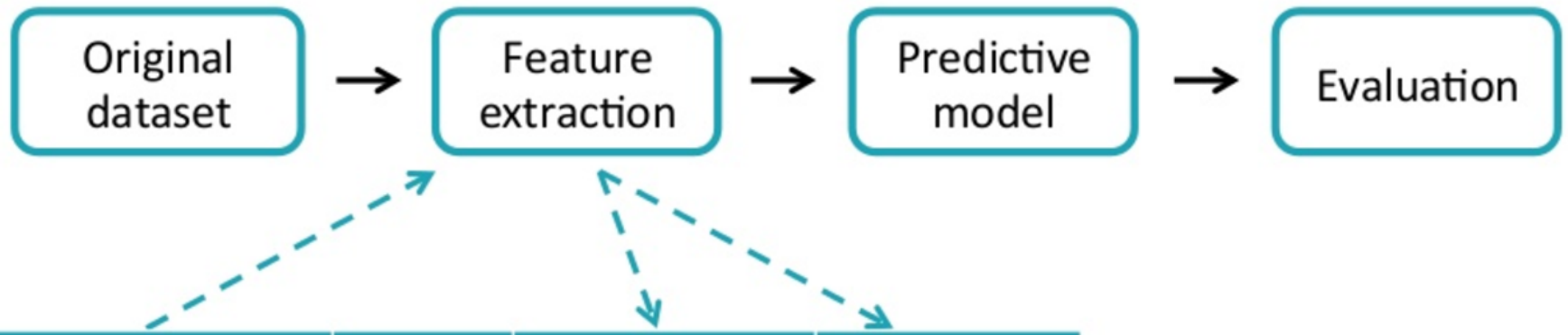
We use exactly the same pipeline over different data (train and test set)

Data Loading



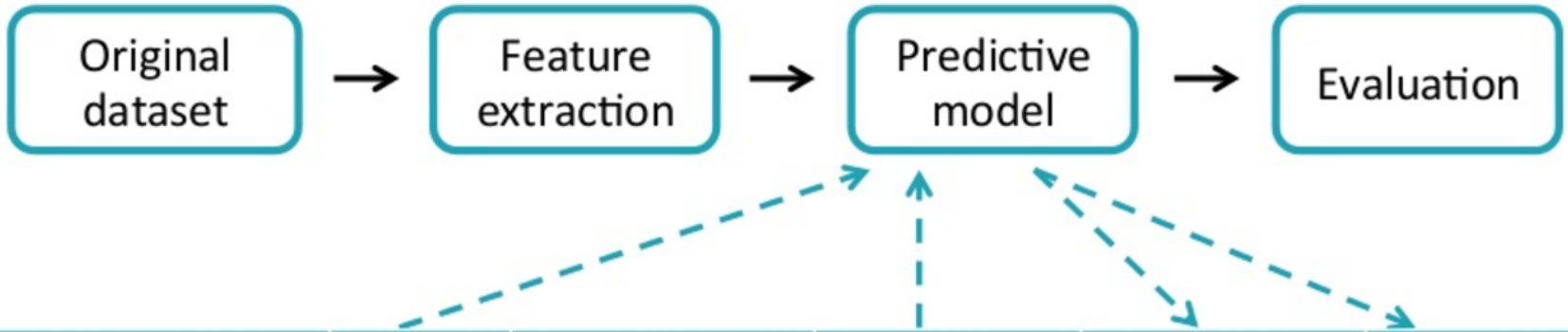
Text	Label
I bought the game...	4
Do NOT bother try...	1
this shirt is aweso...	5
never got it. Seller...	1
I ordered this to...	3

Feature Extraction



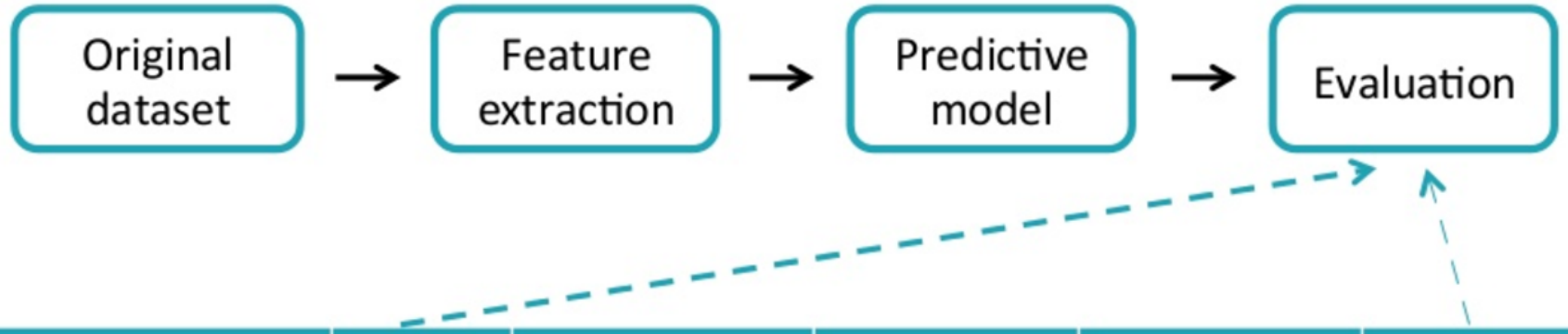
Text	Label	Words	Features
I bought the game...	4	“i", “bought”,...	[1, 0, 3, 9, ...]
Do NOT bother try...	1	“do”, “not”,...	[0, 0, 11, 0, ...]
this shirt is aweso...	5	“this”, “shirt”	[0, 2, 3, 1, ...]
never got it. Seller...	1	“never”, “got”	[1, 2, 0, 0, ...]
I ordered this to...	3	“i”, “ordered”	[1, 0, 0, 3, ...]

Model Fitting



Text	Label	Words	Features	Prediction	Probability
I bought the game...	4	“i", “bought”,...	[1, 0, 3, 9, ...]	4	0.8
Do NOT bother try...	1	“do”, “not”,...	[0, 0, 11, 0, ...]	2	0.6
this shirt is aweso...	5	“this”, “shirt”	[0, 2, 3, 1, ...]	5	0.9
never got it. Seller...	1	“never”, “got”	[1, 2, 0, 0, ...]	1	0.7
I ordered this to...	3	“i”, “ordered”	[1, 0, 0, 3, ...]	4	0.7

Model Evaluation



Text	Label	Words	Features	Prediction	Probability
I bought the game...	4	"i", "bought",...	[1, 0, 3, 9, ...]	4	0.8
Do NOT bother try...	1	"do", "not",...	[0, 0, 11, 0, ...]	2	0.6
this shirt is aweso...	5	"this", "shirt"	[0, 2, 3, 1, ...]	5	0.9
never got it. Seller...	1	"never", "got"	[1, 2, 0, 0, ...]	1	0.7
I ordered this to...	3	"i", "ordered"	[1, 0, 0, 3, ...]	4	0.7

Features

Extraction

Extracting features from “raw” data

Transformation

Scaling, converting, or modifying features

Selection

Selecting a subset from a larger set of features

Locality Sensitive Hashing (LSH)

Class of algorithms that combines aspects of feature transformation with other algorithms

Feature Extraction

- TF-IDF
- Word2Vec
- CountVectorizer

Feature Extraction: Tokenizer

```
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

sentenceData = spark.createDataFrame([
    (0.0, "Hi I heard about Spark"),
    (0.0, "I wish Java could use case classes"),
    (1.0, "Logistic regression models are neat")
], ["label", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
wordsData = tokenizer.transform(sentenceData)

hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=20)
featurizedData = hashingTF.transform(wordsData)
# alternatively, CountVectorizer can also be used to get term frequency vectors

idf = IDF(inputCol="rawFeatures", outputCol="features")
idfModel = idf.fit(featurizedData)
rescaledData = idfModel.transform(featurizedData)

rescaledData.select("label", "features").show()
```

Feature Extraction: Word2Vec

```
from pyspark.ml.feature import Word2Vec

# Input data: Each row is a bag of words from a sentence or document.
documentDF = spark.createDataFrame([
    ("Hi I heard about Spark".split(" "), ),
    ("I wish Java could use case classes".split(" "), ),
    ("Logistic regression models are neat".split(" "), )
], ["text"])

# Learn a mapping from words to Vectors.
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="text", outputCol="result")
model = word2Vec.fit(documentDF)

result = model.transform(documentDF)
for row in result.collect():
    text, vector = row
    print("Text: [%s] => \nVector: %s\n" % (", ".join(text), str(vector)))
```

Feature Transformation

- Tokenizer
- StopWordsRemover
- n-Gram
- PCA
- String Indexer
- OneHotEncoder
- And many others…

Feature Transformation: StopWordsRemover

```
from pyspark.ml.feature import StopWordsRemover

sentenceData = spark.createDataFrame([
    (0, ["I", "saw", "the", "red", "balloon"]),
    (1, ["Mary", "had", "a", "little", "lamb"])
], ["id", "raw"])

remover = StopWordsRemover(inputCol="raw", outputCol="filtered")
remover.transform(sentenceData).show(truncate=False)
```

Feature Transformation: NGram

```
from pyspark.ml.feature import NGram

wordDataFrame = spark.createDataFrame([
    (0, ["Hi", "I", "heard", "about", "Spark"]),
    (1, ["I", "wish", "Java", "could", "use", "case", "classes"]),
    (2, ["Logistic", "regression", "models", "are", "neat"])
], ["id", "words"])

ngram = NGram(n=2, inputCol="words", outputCol="ngrams")

ngramDataFrame = ngram.transform(wordDataFrame)
ngramDataFrame.select("ngrams").show(truncate=False)
```

Feature Transformation: OneHotEncoder

```
from pyspark.ml.feature import OneHotEncoderEstimator

df = spark.createDataFrame([
    (0.0, 1.0),
    (1.0, 0.0),
    (2.0, 1.0),
    (0.0, 2.0),
    (0.0, 1.0),
    (2.0, 0.0)
], ["categoryIndex1", "categoryIndex2"])

encoder = OneHotEncoderEstimator(inputCols=["categoryIndex1", "categoryIndex2"],
                                 outputCols=["categoryVec1", "categoryVec2"])
model = encoder.fit(df)
encoded = model.transform(df)
encoded.show()
```

Feature Selectors

- VectorSlicer
- Rformula
- ChiSqSelector

Feature Selectors

```
from pyspark.ml.feature import VectorSlicer
from pyspark.ml.linalg import Vectors
from pyspark.sql.types import Row

df = spark.createDataFrame([
    Row(userFeatures=Vectors.sparse(3, {0: -2.0, 1: 2.3})),
    Row(userFeatures=Vectors.dense([-2.0, 2.3, 0.0]))])

slicer = VectorSlicer(inputCol="userFeatures", outputCol="features", indices=[1])

output = slicer.transform(df)

output.select("userFeatures", "features").show()
```

Locality Sensitive Hashing (LSH)

LSH Operations

- Feature Transformation
- Approximate Similarity Join
- Approximate Nearest Neighbor Search

LSH Algorithm

- Bucketed Random Projection for Euclidean Distance
- MinHash for Jaccard Distance

Pipeline

Transformers

Converts through a method transform() a DataFrame into another DataFrame, typically appending one or more columns

Estimators

Converts through a method fit() a DataFrame into a Model, which is a Transformer. By calling fit(), it trains a model of the specified algorithm with the specified DataFrame

Random Forest Classifier

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Load and parse the data file, converting it to a DataFrame.
data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

# Index labels, adding metadata to the label column.
# Fit on whole dataset to include all labels in index.
labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)
```

Random Forest Classifier

```
# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer =\
    VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)

# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = data.randomSplit([0.7, 0.3])
```

Random Forest Classifier

```
# Train a RandomForest model.
rf = RandomForestClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures", numTrees=10)

# Convert indexed labels back to original labels.
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",
                               labels=labelIndexer.labels)
```

Random Forest Classifier

```
# Chain indexers and forest in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, rf, labelConverter])

# Train model.  This also runs the indexers.
model = pipeline.fit(trainingData)

# Make predictions.
predictions = model.transform(testData)
```

Linear Regression

```
from pyspark.ml.regression import LinearRegression

# Load training data
training = spark.read.format("libsvm")\
    .load("data/mllib/sample_linear_regression_data.txt")

lr = LinearRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Fit the model
lrModel = lr.fit(training)

# Print the coefficients and intercept for linear regression
print("Coefficients: %s" % str(lrModel.coefficients))
print("Intercept: %s" % str(lrModel.intercept))
```

K-Means Clustering

```
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator

# Loads data.
dataset = spark.read.format("libsvm").load("data/mllib/sample_kmeans_data.txt")

# Trains a k-means model.
kmeans = KMeans().setK(2).setSeed(1)
model = kmeans.fit(dataset)

# Make predictions
predictions = model.transform(dataset)
```

Apache Spark

- Spark is a PROCESSING framework, not a STORAGE system
- Spark is faster than Hadoop
- Spark is one of the most used distributed data processing systems in both industry and research

Spark Core

- RDD: Resilient Distributed Dataset, it is a collection of records spread over one or many partitions
- Resilient: i.e., fault-tolerant, able to recompute missing or damaged partitions due to node failures
- Distributed: with data residing on multiple nodes in a cluster
- Dataset: is a collection of primitive values (strings, integers, …) or values of values (tuples, arrays, or other objects)

Spark Core

- Operations: transformations, and actions
- Transformation: operations that return another RDD
  (map, flatMap, filter)
- Actions: operations that trigger computation and return values
  (count, collect)
- Lazy computation: the data inside RDD is not available or transformed until an action is executed that triggers the execution

Spark SQL

- DataFrame: evolution of RDD for tabular data, easier to access a field and to save as output

- DataFrames are distributed through multiple nodes in the same way an RDD is

Spark ML

- Two main operations: fit and transform
- At the beginning we have a ML algorithm (RandomForestClassifier, Kmeans, …)

- With fit we use the training dataset to obtain a ML model

- With transform we apply the model on the test dataset

Thank you!

Questions?

You can find more at

`github.com/forons/BigDataExamples`

Or write me an email

`daniele.Foroni@Huawei.com`

HUAWEI db Trento