Mining Massive Datasets

Lecture 2

Artur Andrzejak

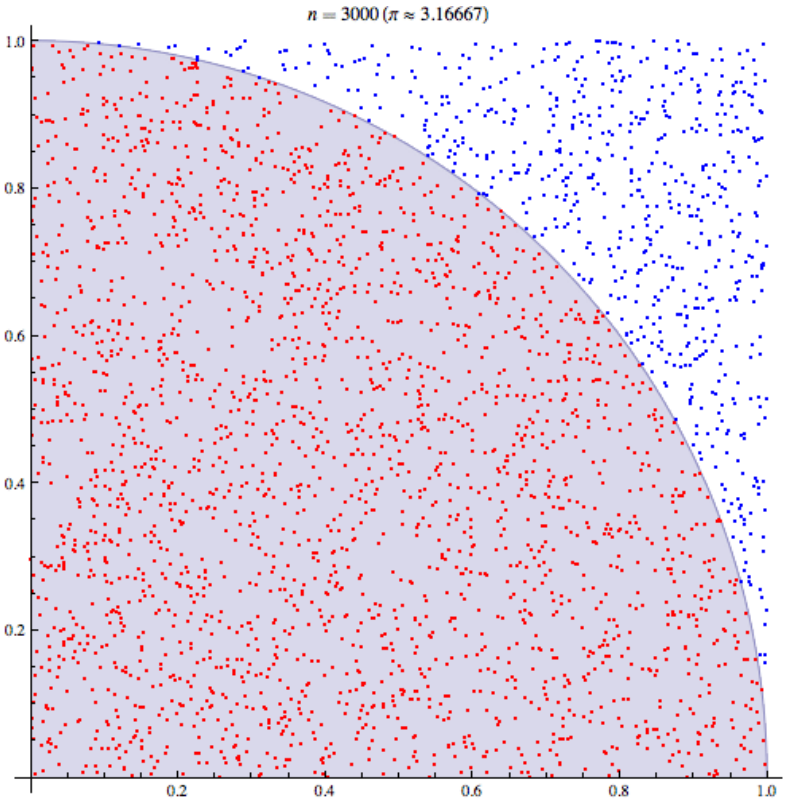
RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

http://pvs.ifi.uni-heidelberg.de

Spark RDD Programming: Example

Estimating π

- We use a Monte Carlo method to estimate the value of Pi (π)
- Idea: Count the share of random points (x, y) whose distance to (0,0) is 1 or less
- Naturally parallelizable



Source: Wikipedia

Estimating Pi/4

```
N = 1000000
def inCircle(p):
    x, y = random(), random()
    return 1  if x*x + y*y < 1 else 0

myplus = lambda a, b: a + b

rawDataRDD = sc.parallelize(range(0,N), partitions)
inCircleRDD = rawDataRDD.map(inCircle)
count = inCircleRDD.reduce(myplus)

print("Pi is roughly %f" % (4.0 * count / N))
```

Generates two pseudo-random numbers in [0.0, 1.0)

True iff distance of (x,y) to origin is < 1

Generates an iterable ("lazy list") from 0 to N-1

A Stand-Alone Program

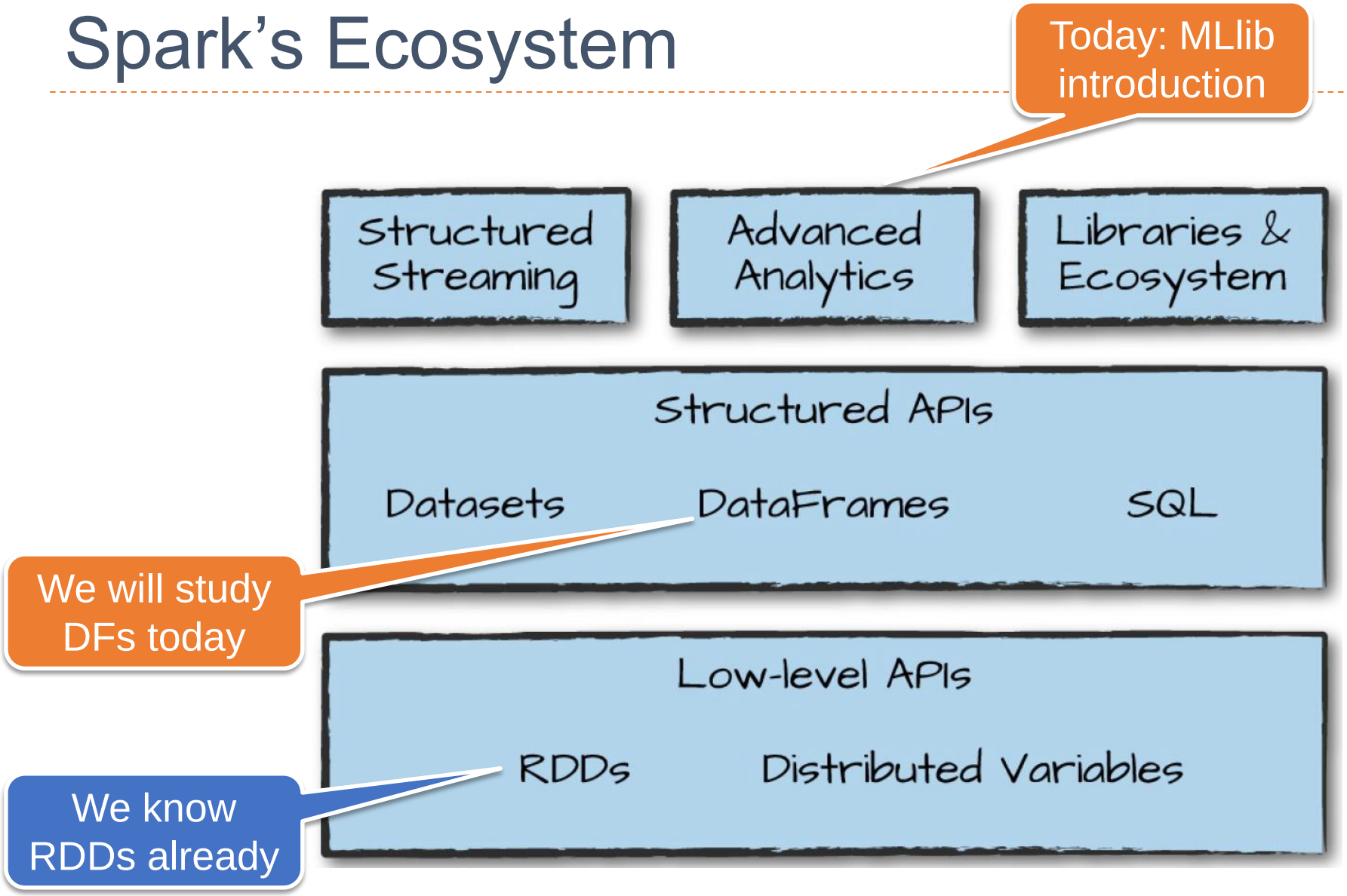
```
import sys
from random import random
from operator import add
from pyspark import SparkContext

if __name__ == "__main__":
"""Usage: pi [partitions]"""“

sc = SparkContext (appName="PythonPi")
partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2
N = 100000 * partitions

… [code as above, without N = 1000000] …

sc.stop()
```

Execution Example

- Open terminal window
- cd spark/examples/src/main/python
- spark-submit pi.py 1
  - => Pi is roughly 3.139168
- spark-submit pi.py 2
  - => Pi is roughly 3.138352
- spark-submit pi.py 3
  - => Pi is roughly 3.142220
- spark-submit pi.py 4
  - => Pi is roughly 3.142624

Structured Spark APIs: DataFrames

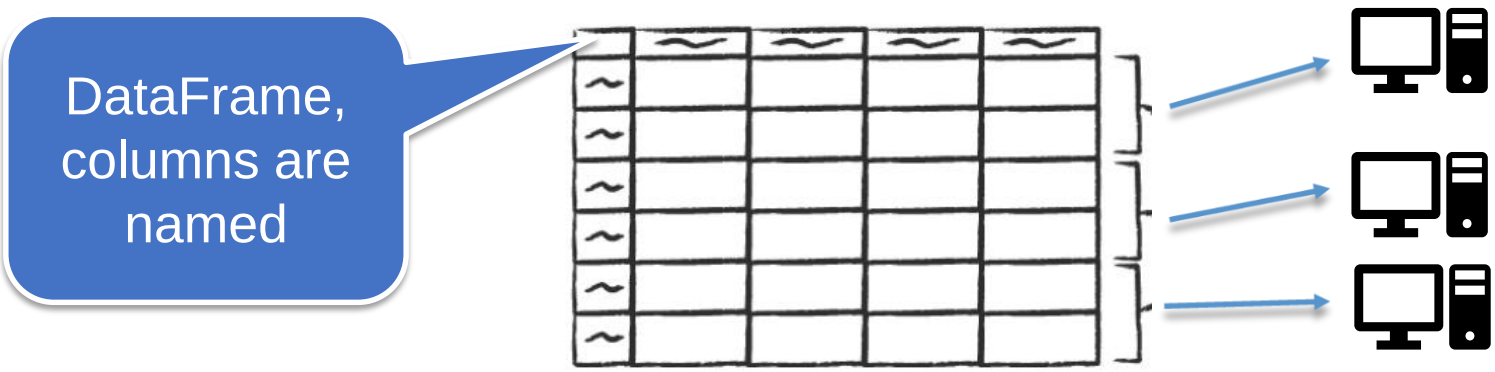
Spark's Ecosystem



From *Spark: The Definitive Guide*, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

What are DataFrames (DF)?

- **A table of data with rows and (named) columns**
  - Like a spreadsheet with named columns
  - Or dataframes in R, Python/Pandas or tables in DBs
- Can span thousands of computers



From *Spark: The Definitive Guide*, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

DataFrame Details

- DataFrame is the most commonly used API in Spark
- Schema
  - Each DF has an associated schema: essentially, **a definition of column names and their types**
  - Can be set programmatically, or inferred from data
- Partitions
  - Spark breaks up DFs into chunks called partitions
  - A P. is a **collection of rows on one physical machine**
  - Note: number of partitions should be ~ number of executors (worker CPUs/cores): if you have one partition, Spark will have a parallelism of only one, even if you have thousands of executors!

SparkSession vs. SparkContext

- **(!)** In the previous slides, we used SparkContext to initialize Spark and get access to APIs
- Nowadays, it is preferred to use **SparkSession** (via a builder method)
  - This instantiates the Spark contexts more robustly
- Python:

```
# Creating a SparkSession in Python
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local")\
      .appName("Word Count")\
      .config("spark.some.config.option", "value")\
      .getOrCreate()
```

Example – Using DataFrames

- Assume that your SparkSession-object is called „spark“
- We use a small data set with US flights
  - See https://github.com/databricks/Spark-The-Definitive-Guide/tree/master/data/flight-data
- File *2015-summary.csv* (csv = comma-separated values):
  - `DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count`
  - `United States,Romania,15`
  - `United States,Croatia,1`
  - `United States,Ireland,344`
  - …
- For more explanations, see Sec. 2 of „Spark: The Definitive Guide,..”, 2018 (online, see book list)

Read Data with Schema Inference

- We want to read this data into a DataFrame, and let Spark guess the schema of our DataFrame
  - To get the schema information, Spark reads in a little bit of the data and then attempts to parse the types in those rows according to the types available in Spark
  - You also have the option of strictly specifying a schema when you read in data (recommended, see later)
- Python:

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

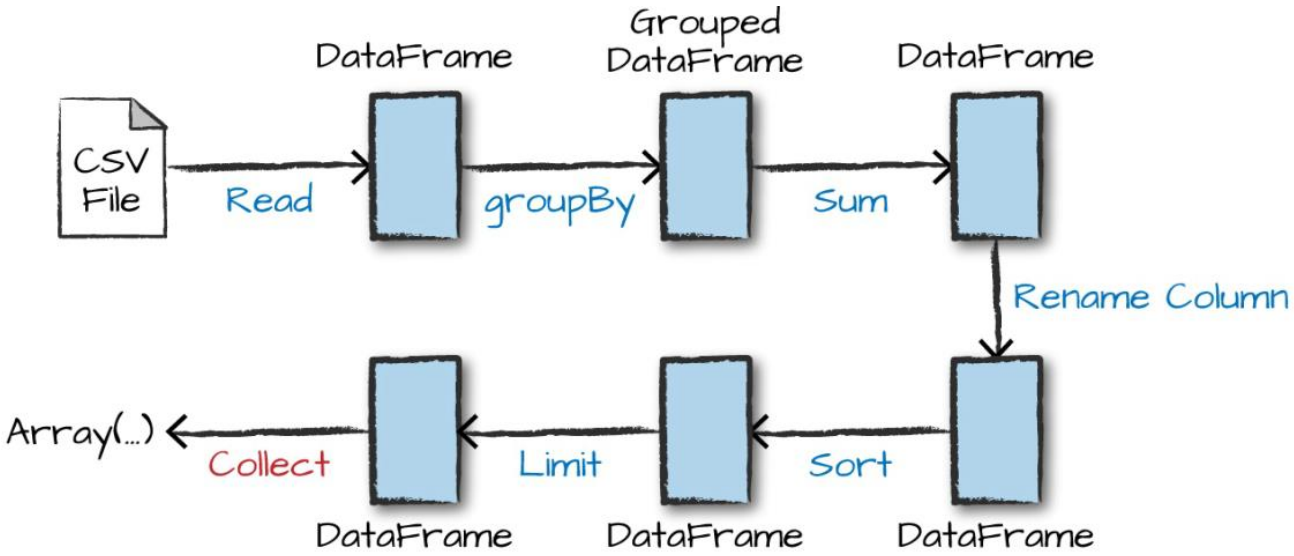
Querying Data

- Let us find the top five destination countries
- Python:

```
from pyspark.sql.functions import desc
flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .sum("count")\
  .withColumnRenamed("sum(count)“,"dest_total")\
  .sort(desc("dest_total"))\
  .limit(5)\
  .show()
```

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

Execution is Optimized (See Additional Slides)



- The specified chain of operations (transformations + action) is optimized before execution
- Operation „.explain()“ shows the physical plan

```
TakeOrderedAndProject(limit=5, orderBy=[destination_total#16194L DESC], outpu...
+- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[sum(count#7325L)])
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#7323, 5)
      +- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[partial_sum...
         +- InMemoryTableScan [DEST_COUNTRY_NAME#7323, count#7325L]
               +- InMemoryRelation [DEST_COUNTRY_NAME#7323, ORIGIN_COUNTRY_NA...
                     +- *Scan csv [DEST_COUNTRY_NAME#7578,ORIGIN_COUNTRY_NAME...
```

From *Spark: The Definitive Guide*, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

Creating DataFrames

A DataFrame can be created in multiple ways:

- **Loading** from text, csv, json, xml, parquet files
  - We saw an example with schema inference
- By **converting** from „normal“ **RDDs**
- **Importing** from DBMS (Hive, Cassandra, ..)
- **Creating** a DataFrame **on the fly** by taking a set of rows and converting them to a DataFrame

Creating DataFrames from Python Data

```
from pyspark.sql import Row
from pyspark.sql.types import StructField, StructType, StringType, LongType
# Specify schema manually
myManualSchema = StructType([
  StructField("some", StringType(), True),
  StructField("col", StringType(), True),
  StructField("names", LongType(), False)
])
# Create a row and a DataFrame with it
myRow = Row("Hello", None, 1)
myDf = spark.createDataFrame(
                          [myRow], myManualSchema)
myDf.show()
```

Creating DataFrames from RDDs

```
// Read file with rows: <name, age>
filePath = ".../people.txt“
parts = sc.textFile(filePath).map(
                              lambda line: line.split(","))
// Each row should become a tuple (name, age)
peopleRDD = parts.map(lambda p: (p[0], p[1].strip()))

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("X").getOrCreate()
schema = StructType([
            StructField ("name", StringType(), True),
            StructField ("age",  StringType(), True)
         ])
dfPeople = spark.createDataFrame(peopleRDD, schema)
print (dfPeople.take(5))
```

Reading DFs from Other Sources

```
filePath = ".../people.txt"
// Read from CSV (comma separated values) files
df_csv = spark.read.csv(filePath, schema = schema)
print (df_csv.take(5))


// Read from json – schema is inferred!
df_json = spark.read.json(".../people.json")
df_json.show()
  # +----+-------+
  # | age|   name|
  # +----+-------+
  # |null|Michael|
  # |  30|   Andy|
  # |  19| Justin|
  # +----+-------+
```

PySpark Operations in Cheat Sheets

- See "PySpark_SQL_Cheat_Sheet_Python.pdf“ in heibox, MMD-Materials\ Cheat_Sheets

Add, Update & Remove Columns

Adding Columns

```
>>> df = df.withColumn('city',df.address.city) \
           .withColumn('postalCode',df.address.postalCode) \
           .withColumn('state',df.address.state) \
           .withColumn('streetAddress',df.address.streetAddress) \
           .withColumn('telePhoneNumber',
                       explode(df.phoneNumber.number)) \
           .withColumn('telePhoneType',
                       explode(df.phoneNumber.type))
```

Updating Columns

```
>>> df = df.withColumnRenamed('telePhoneNumber', 'phoneNumber')
```

Removing Columns

```
>>> df = df.drop("address", "phoneNumber")
>>> df = df.drop(df.address).drop(df.phoneNumber)
```

GroupBy

```
>>> df.groupBy("age")\
      .count() \
      .show()
```

Group by age, count the members in the groups

Filter

```
>>> df.filter(df["age"]>24).show()
```

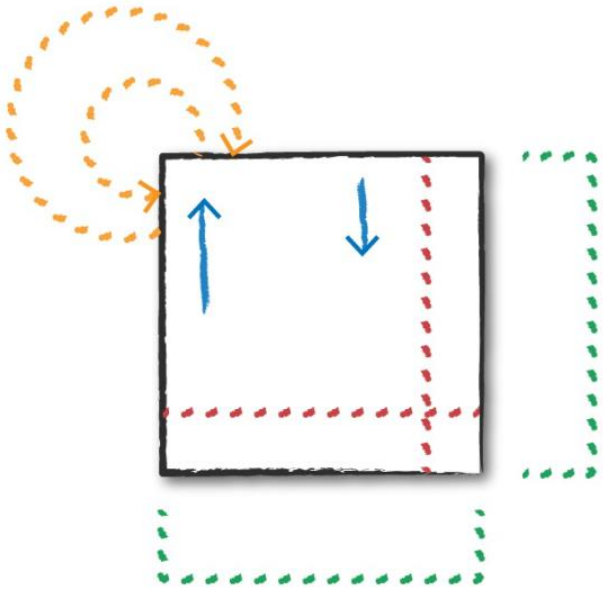
Filter entries of age, only keep those records of which the values are >24

Sort

```
>>> peopledf.sort(peopledf.age.desc()).collect()
>>> df.sort("age", ascending=False).collect()
>>> df.orderBy(["age","city"],ascending=[0,1])\
      .collect()
```

Typical DataFrame Tasks ...

- Add rows or columns
- Remove rows or columns
- Transform a row into a column (or vice versa)
- Change the order of rows based on the values in cols



- We can translate all of these tasks into simple transformations
- The most common scenario is: take one column, change it row by row, and then return the results

Examples of Complex DF Operations

Select only the "name" column

```
dfPeople.select("name").show()
# |    name|
# |Michael| …
```

Group by age and count per group

```
dfPeople.groupBy("age").count().show()
# |  age|count|
# |   19|    1|
# |null|    1|
# |   30|    1|
```

Print schema

```
dfPeople.printSchema()
# root
# |-- age: string (nullable = true)
# |-- name: string (nullable = true)
```

Standard DFs Functions

There are many ready-to-use functions, similar to those in convensional SQL

Type	Sample Available Functions
String functions	`startswith`, `substr`, `concat`, `lower`, `upper`, `regexp_extract`, `regexp_replace`
Math functions	`abs`, `ceil`, `floor`, `log`, `round`, `sqrt`
Statistical functions	`avg`, `max`, `min`, `mean`, `stddev`
Date functions	`date_add`, `datediff`, `from_utc_timestamp`
Hashing functions	`md5`, `sha1`, `sha2`
Algorithmic functions	`soundex`, `levenshtein`
Windowing functions	`over`, `rank`, `dense_rank`, `lead`, `lag`, `ntile`

From: Jeffrey Aven: Sams teach yourself Apache Spark in 24 hours, 2017

User Defined Functions (UDFs)

```
from pyspark.sql.functions import *
from pyspark.sql.types import *
df = sqlContext.read.parquet('.../stations.parquet')

lat2dir = udf(lambda x: 'N' if x>0 else 'S',      StringType())
lon2dir = udf(lambda x: 'E' if x>0 else 'W',      StringType())
df.select(df.lat,  lat2dir(df.lat).alias('latdir'),
          df.lon, lon2dir(df.lon).alias('londir')).show()
```

```
+----------+------+-----------+------+
|       lat|latdir|        lon|londir|
+----------+------+-----------+------+
| 37.329732|     N|-121.901782|     W|
| 37.330698|     N|-121.888979|     W|
| 37.333988|     N|-121.894902|     W|
|-37.331415|     S|   121.8932|     E|
| 37.336721|     N| 121.894074|     E|
+----------+------+-----------+------+
```

From: Jeffrey Aven: Sams teach yourself Apache Spark in 24 hours, 2017

Spark SQL: Writing Queries in SQL

- In Spark 2.0 support for SQL and Hive was added
  - See Sec. 10 of „Spark: The Definitive Guide...”
- => You can register any DataFrame as a table or view (a temporary table) and query it using pure SQL
  - There is no performance difference between writing SQL queries or writing DataFrame code
  - Spark implements a subset of ANSI SQL:2003
- Anecdote: Spark became so successful at replacing Hive, that even Facebook (Hive creator) uses it:
  - *“The Spark-based pipeline produced significant performance improvements (4.5–6x CPU, 3–4x resource reservation, and ~5x latency) compared with the old Hive-based pipeline, and it has been running in production for several months.” (Facebook blog post, link)*

Spark SQL: Example with flightData2015

- Make DataFrame into a table or view

```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

- A new „global“ table "flight_data_2015“ is added to session data => you don‘t need to capture the output
- As before, let us find the top five destination countries

```
maxSql = spark.sql("""
  SELECT DEST_COUNTRY_NAME, sum(count) as dest_total
  FROM flight_data_2015
  GROUP BY DEST_COUNTRY_NAME
  ORDER BY dest_total DESC
  LIMIT 5""")
maxSql.show()
```

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

Spark SQL: Example with Dataset „people“

Standard DF API:

```
# Select people older than 25
dfPeople.filter(dfPeople['age']>25).show()
```

Same result with Spark SQL:

```
# Register the DataFrame as a SQL temporary view
dfPeople.createOrReplaceTempView("people")

df = spark.sql("SELECT * FROM people where age > 25")
df.show()
```

Quiz on Dataframes

- Open the link Pingo <todo>
- Answer the questions (2 minutes)

- Mark true statements related to Spark Resilient Distributed Datasets (RDDs) or Spark DataFrames.
  1. RDDs allow specifying data types, DataFrames not
  2. Querying or processing DataFrames via Spark SQL is as powerful and fast as via API (chains of method calls)
  3. You can emulate RDD-pair transformations like reduceByKey via DataFrames and its API
  4. RDDs are higher-level data structures and internally map to DataFrames

Apache Spark: Machine Learning (ML) with Spark

Advanced Data Analytics …

- .. Are techniques for *deriving insights* and *making predictions* or *recommendations* based on data

Common tasks include:

- Supervised learning: classification and regression
  - Goal: predict label/real value for data point based features
- Recommendation engines
  - Goal: suggest products to users based on behavior
- Unsupervised learning: clustering, anomaly detection, topic modeling
  - Goal: discover structure in the data
- Graph analytics: finding patterns in networks

MLlib: a Core Package of Spark

- **MLlib** is a package of Spark with APIs/routines for:
  - Gathering and cleaning data
  - Feature engineering and feature selection
  - Training and tuning large-scale un/supervised ML models
  - Using such ML models in production
- **MLlib** consists of two packages
  - **pyspark.ml** (or org.apache.spark.ml)
    - Preferred higher level API, currently the “main” API (Spark 2.x)
    - Uses DataFrames and provides ML pipelines
  - **pyspark.mllib** (or org.apache.spark.mllib)
    - Lower-level package using RDDs
    - Now in maintenance mode, no new features

Workflow for Obtaining ML Models in Spark

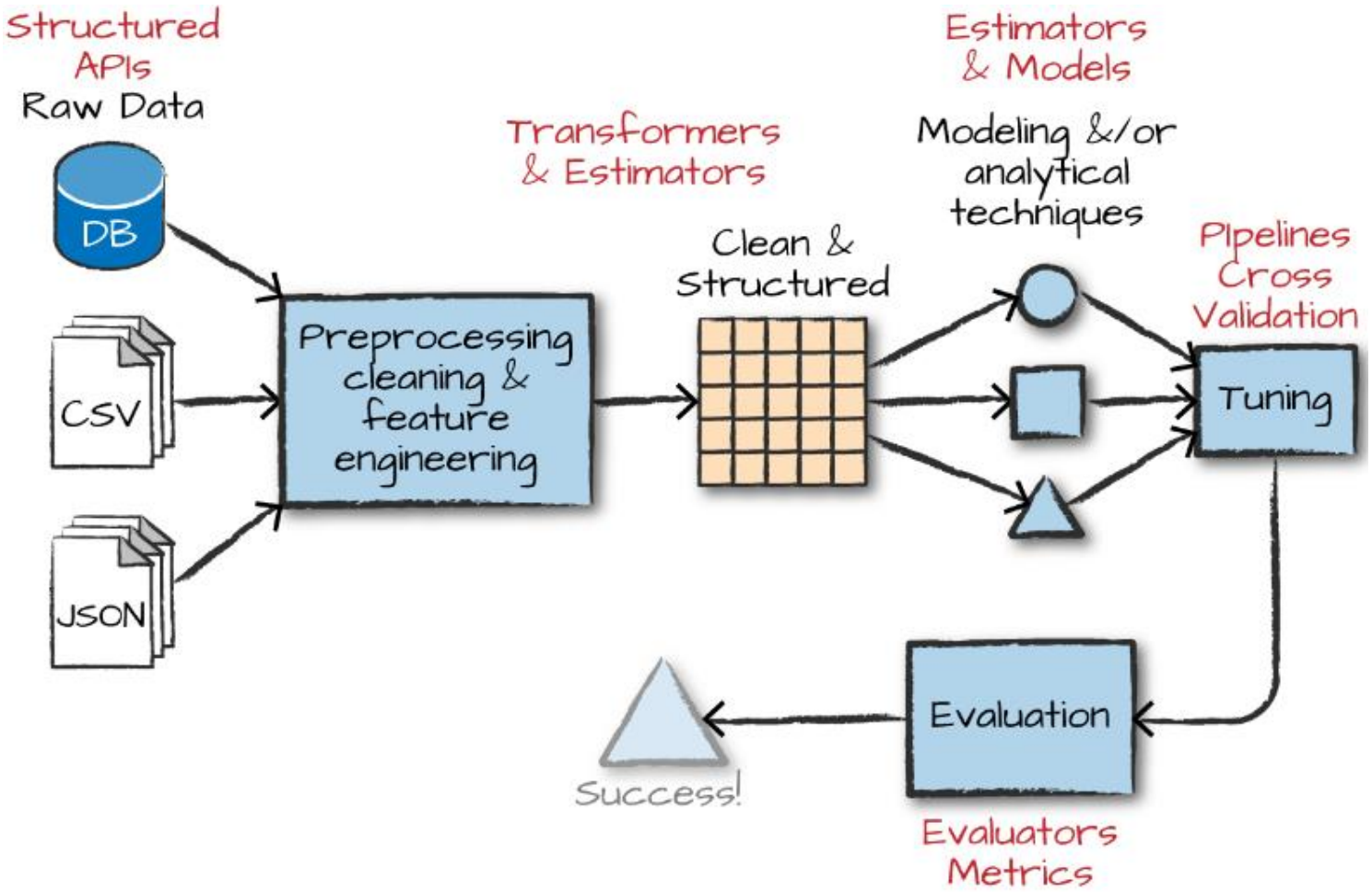


Fig. 24-2. from *Spark: The Definitive Guide*, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

ML Pipelines in Spark

- Spark's **ML Pipelines** API (link) allow to set up a sequence of stages for
  - Data cleaning, feature extraction, model training, model validation and tuning, model testing, ..
- They make it easier to combine multiple algorithms into a single pipeline (workflow)
  - The pipeline concept is mostly inspired by the scikit-learn project (for Python)
- They use DataFrames (DF), i.e. essentially DB-like tables with columns of various types
  - See previous lecture/slides

Essential Elements of the ML Pipelines API

- **Transformer**: an algorithm which can transform one DataFrame into another DataFrame
  - E.g., an ML model transforms a DataFrame with features into a DataFrame with predictions

- **Estimator**: an algorithm which can be fit on a DataFrame to produce a Transformer
  - E.g., a learning algorithm trains on a DataFrame and produces a *model*

Essential Elements of the ML Pipelines API

- **Pipeline**: an object which chains multiple Transformers and Estimators together to a workflow

- **Parameter**: a common API for specifying parameters of Transformers and Estimators

Transformers: Details

- An abstraction that includes *feature transformers* and *learned models*
- Transformers convert data in some way
  - E.g. normalize a column; predict a label for each feature
- They add more columns or change the DF values
- Call the method **transform()** to activate

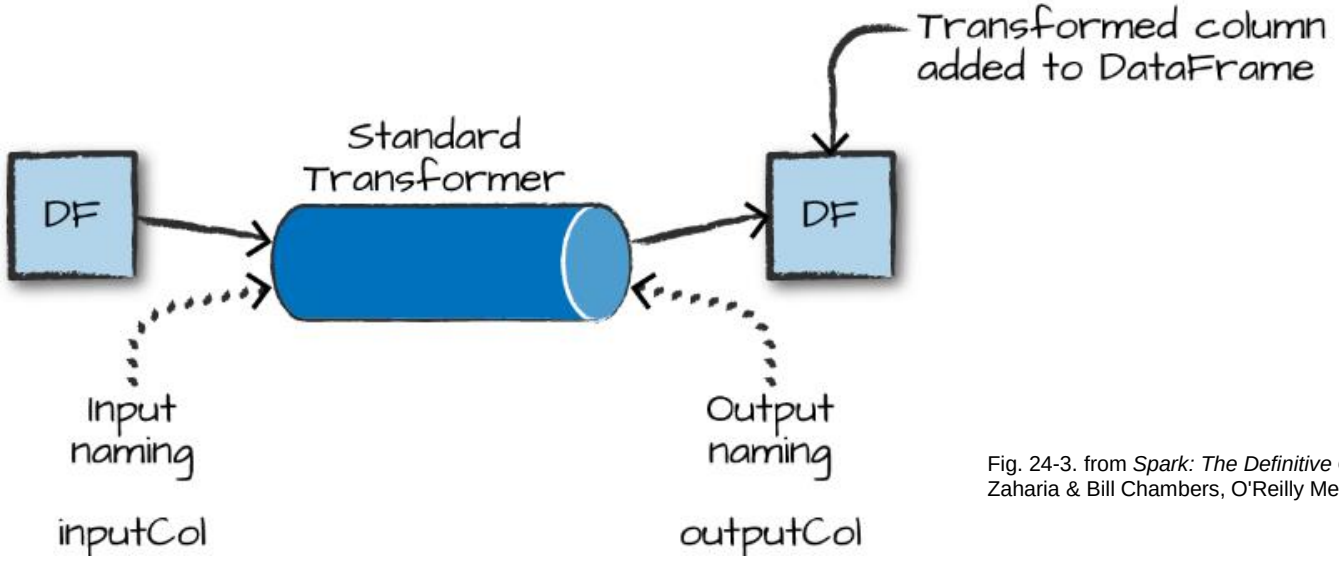


Fig. 24-3. from *Spark: The Definitive Guide*, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

Estimators: Details

- An Estimator abstracts the concept of a learning algorithm or any algorithm that fits or trains on data
- Technically, an E. implements a method **fit()**, which accepts a DataFrame and produces a *model* (Transformer)
  - E.g., a learning algorithm such as LogisticRegression is an Estimator, and calling **fit()** trains a LogisticRegressionModel, which is a Model and hence a Transformer

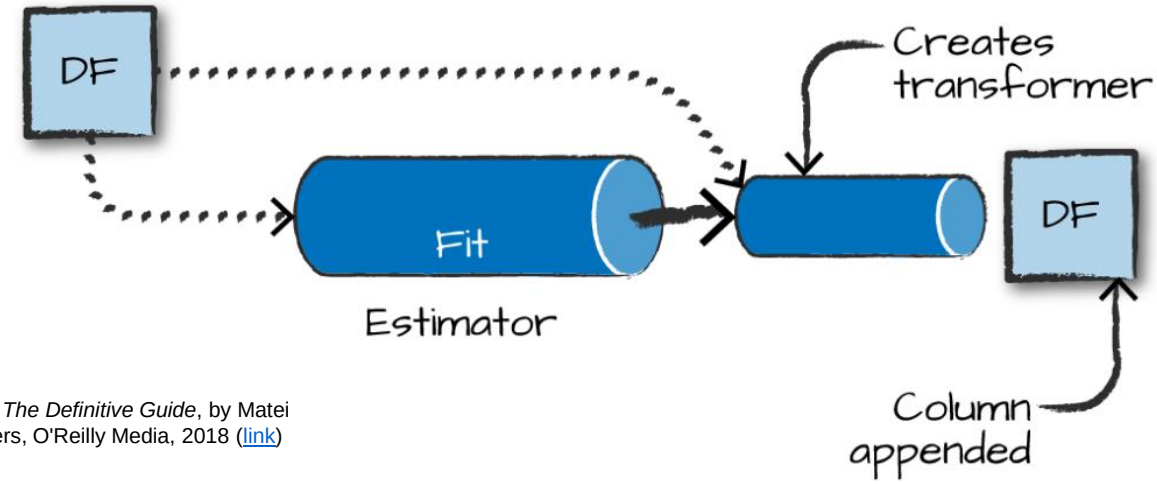


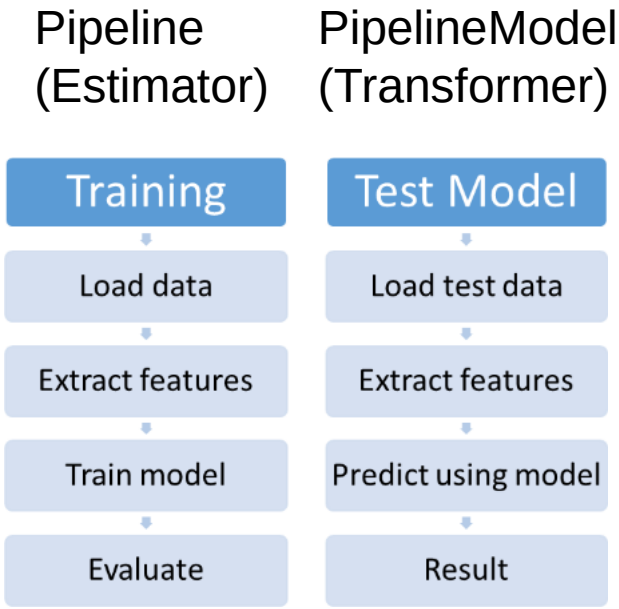
Fig. 25-2. from *Spark: The Definitive Guide*, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

Pipelines as Sequences

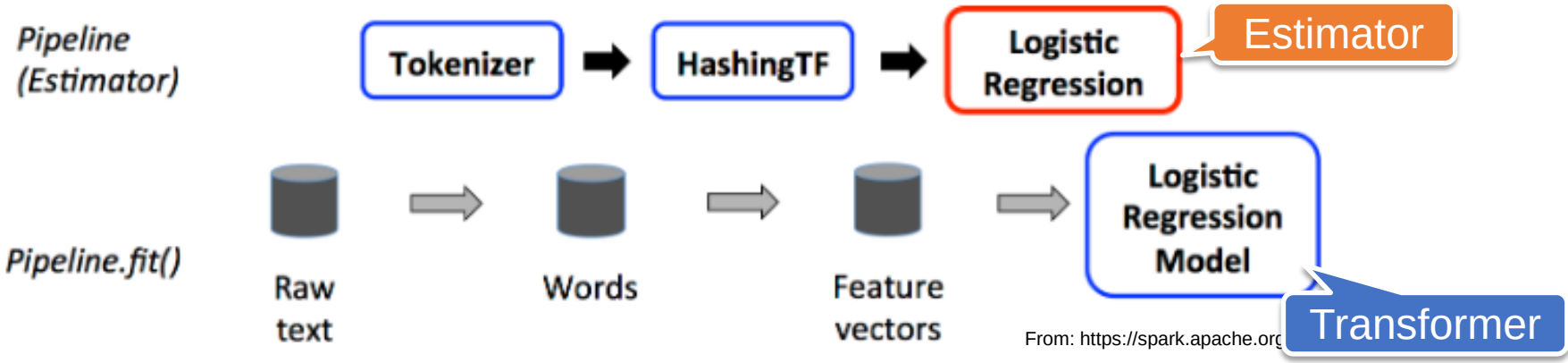
- In ML it is common to run a sequence of algorithms to process and learn from data
- E.g., a simple text document processing workflow might include several stages:
  - Split each document's text into words
  - Convert each word into a numerical feature vector
  - Learn a prediction model using the feature vectors and labels
- MLlib represens this workflow as a **Pipeline**
  - It consists of a sequence of **PipelineStages** (Transformers and Estimators) to be run in a specific order

Pipelines vs. PipelineModels

- In an ML study, we typically have these subtasks:
  - Training: build model(s) according to data
  - Test/prediction: apply models to new data
- The ML Pipelines API supports both subtasks
- A complete "untrained" Pipeline ***pipe*** is an Estimator
- Training:
  - We call ***pipe***.fit()
  - The result is an object ***model*** of type **PipelineModel**, which is a Transformer
- Test/prediction:
  - Set test data as input to ***model***
  - Call ***model***.transform() to perform predictions on test data

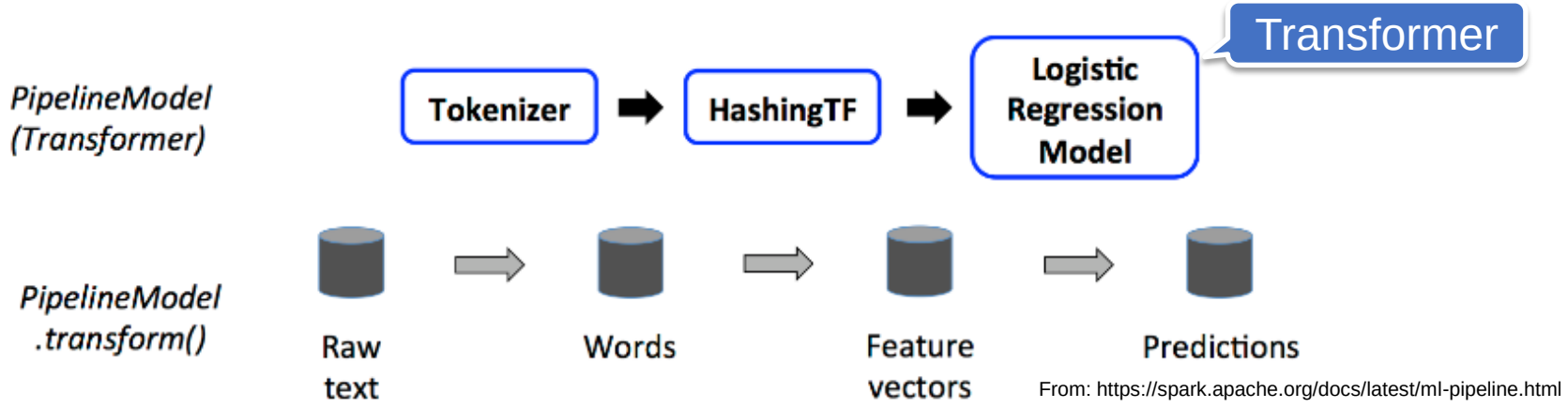


Example Pipeline Usage: Training Time



- The Pipeline.fit() method is called on the original DataFrame with raw text documents and labels
  - Recall: the *complete* pipeline is of type Estimator
- Pipeline calls the transform() methods of Tokenizer and HashingTF, then fit() method of Log. Regression
- After a Pipeline's fit() method runs, it produces a **PipelineModel**, which is a Transformer (=> phase 2)

Example Pipeline Usage: Prediction Time



- The PipelineModel has the same number of stages as the original Pipeline, but all Estimators become Transformers!
- When the PipelineModel's transform() method is called on a test/production dataset, the data are passed through the fitted (= trained) pipeline in order
- In particular, the LogisticRegressionModel performs classification (predictions) due to model trained in 1. phase

Vectors

- We need some lower-level data types, esp. **Vector**
- To pass a set of features to a model, we need to do it as a vector that consists of Doubles
- Vectors can be either sparse (where most of the elements are zero) or dense ("normal")
  - Create sparse: specify an array of all the values
  - Create dense: specify the total size and the indices and values of the non-zero elements

```
from pyspark.ml.linalg import Vectors
denseVec = Vectors.dense(1.0, 2.0, 3.0)
size = 3
idx = [1, 2] # indices of non-zero elements in vector
values = [2.0, 3.0]
sparseVec = Vectors.sparse(size, idx, values)
```

Example: Pipeline (Training) (link)

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# Prepare training documents from a list of (id, text, label)
tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of three stages:
tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(),
outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)
```

Example: Pipeline (Prediction)

```
# Prepare test docs (unlabeled (id, text) tuples)
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "spark hadoop spark"),
    (7, "apache hadoop")
], ["id", "text"])

# Make predictions on test docs, print cols of interest
prediction = model.transform(test)
selected = prediction.select("id", "text",
                             "probability", "prediction")
for row in selected.collect():
    rid, text, prob, prediction = row
    print("(%d, %s) --> prob=%s, prediction=%f" %
          (rid, text, str(prob), prediction))
```

Implemented Functionality in MLlib /1

MLlib provides a large variety of scalable functions:

- Feature Engineering
  - 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)
- Classification
  - Logistic regression, decision tree, random forest,
  - Gradient-boosted trees, multilayer perceptron, linear SVM
- Regression
  - (Generalized) linear regression, decision tree, random forest, .., survival regression, isotonic regression

Implemented Functionality in MLlib /2

- Clustering
  - K-means, Latent Dirichlet allocation (LDA), Bisecting k-means, Gaussian Mixture Model (GMM)
- Collaborative filtering (recommender systems)
- Frequent Pattern Mining
  - FP-Growth, PrefixSpan
- Model selection and hyperparameter tuning
  - Model selection using Pipelines (CrossValidator and TrainValidationSplit)

Overview Feature Engineering Functions

- Feature Extractors
  - TF-IDF
  - Word2Vec
  - CountVectorizer
  - FeatureHasher
- Feature Transformers
  - Tokenizer
  - StopWordsRemover
  - n-gram
  - Binarizer
  - PCA
  - PolynomialExpansion
  - Discrete Cosine Transform (DCT)
  - StringIndexer
  - IndexToString
  - OneHotEncoderEstimator
  - VectorIndexer
  - Interaction
  - Normalizer
  - StandardScaler
  - MinMaxScaler
  - MaxAbsScaler
  - Bucketizer
  - ElementwiseProduct
  - SQLTransformer
  - VectorAssembler
  - VectorSizeHint
  - QuantileDiscretizer
  - Imputer
- Feature Selectors
  - VectorSlicer
  - RFormula
  - ChiSqSelector
- Locality Sensitive Hashing
  - LSH Operations
  - LSH Algorithms

Examples Feature Engineering Functions

- Extractor: Word2Vec
  - Takes sequences of words and trains a Word2VecModel
  - The model maps each word to a unique fixed-size vector
- Feature Transformer: QuantileDiscretizer
  - Takes a column with continuous features and outputs a column with binned categorical features
  - The number of bins is set by the numBuckets parameter
- Feature Selector: ChiSqSelector
  - Uses the *Chi-Squared test of independence* to decide which features to choose
  - It supports 5 selection methods: numTopFeatures, percentile, fpr, fdr, few; e.g.
    - fpr chooses features whose *p-values* are below a threshold

More on Feature Engineering with Spark

Spark: The Definitive Guide, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

Chapter 25. Preprocessing and Feature Engineering

- "Any data scientist worth her salt knows that one of the biggest challenges (and time sinks) in advanced analytics is preprocessing.
- It's not that it's particularly complicated programming, but rather that it requires deep knowledge of the data you are working with and an understanding of what your model needs in order to successfully leverage this data."

NLDSL

- A Visual Studio Code extension which gives you a Domain Specific Language (DSL) for Spark & Pandas
- DSL is expanded into final code during editing
- Try it out ☺
  - https://pvs.ifi.uni-heidelberg.de/software/nldsl
  - Download for Windows, Linux, or Mac

```
## target code = pandas
import pandas as pd

#! name 'data', type 'csv', path 'covid_19_data.csv'
# Dataframe 'data' will have the following columns and their respective names with the no header file:
# SNo, ObservationDate, Province/State, Country/Region, Last Update, Confirmed, Deaths, Recovered

# Load the csv file into the dataframe 'data'
## data = load from 'covid_19_data.csv' as csv_with_header
data = pd.read_csv('covid_19_data.csv')

# Group data by 'Country/Region' and sum the column 'Country/Region' and sort ascending by Confirmed
## on data | group by 'Country/Region' | apply sum on 'Confirmed' as 'Total' | sort by 'Total' ascending | show
print(data.groupby(['Country/Region']).agg({'Confirmed' : 'sum'}).rename(columns={'Confirmed' : 'Total'}).sort_values(['To
```

Recommended Videos on Spark

- Introduction tutorials on Spark
  - Parallel programming with Spark Presented by Matei Zaharia UC Berkeley AmpLab 2013
  - https://www.youtube.com/watch?v=e-56inQL5hQ&t=30s
  - Parallel Programming with Spark (Part 1 & 2) by Matei Zaharia (2012)
  - https://www.youtube.com/watch?v=7k4yDKBYOcw
- Coursera
  - Big Data Analysis with Scala and Spark
  - https://www.coursera.org/learn/scala-spark-big-data
  - Enroll -> Audit, then for free!

Reading Materials on Spark

- Free Materials:
  - Spark Programming Guide
    - https://spark.apache.org/docs/latest/rdd-programming-guide.html
  - Apache Spark Tutorial: ML with PySpark
    - https://goo.gl/u4RjeB
  - Cheat Sheet PySpark-RDD Basics, https://goo.gl/UF5zVr
  - Jacek Laskowski, Mastering Apache Spark 2, GitBook.com, https://goo.gl/yFYRYm
- Books
  - Matthew Rathbone: 10+ Great Books for Apache Spark
    - https://blog.matthewrathbone.com/2017/01/13/spark-books.html

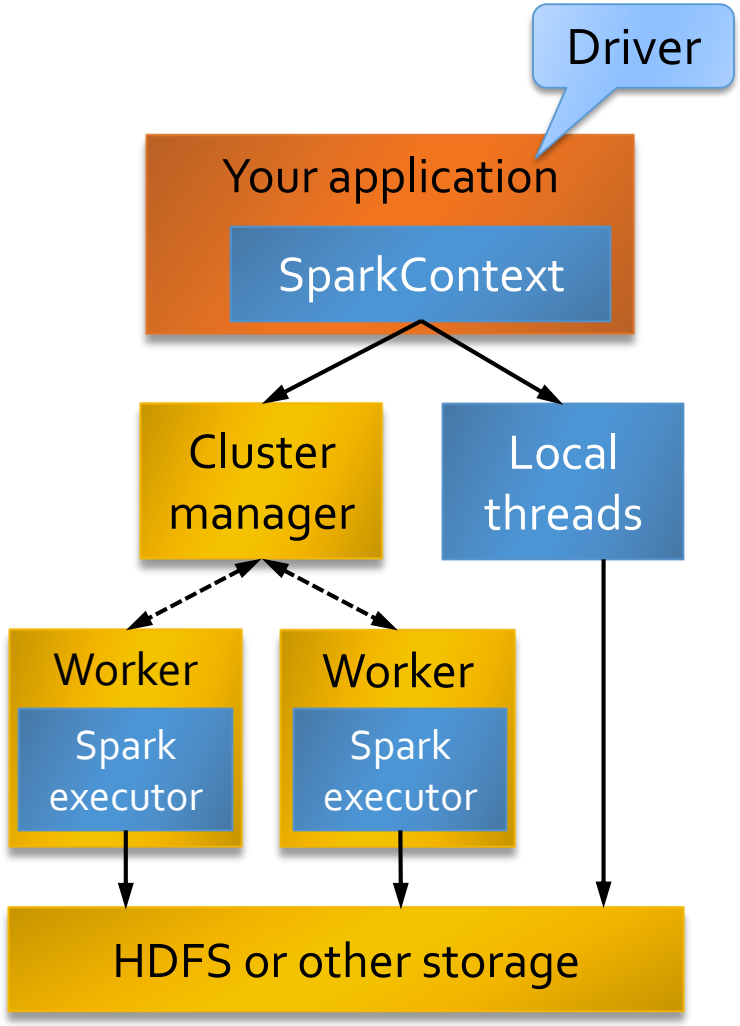
The End

Additional Slides

Spark: Execution Details

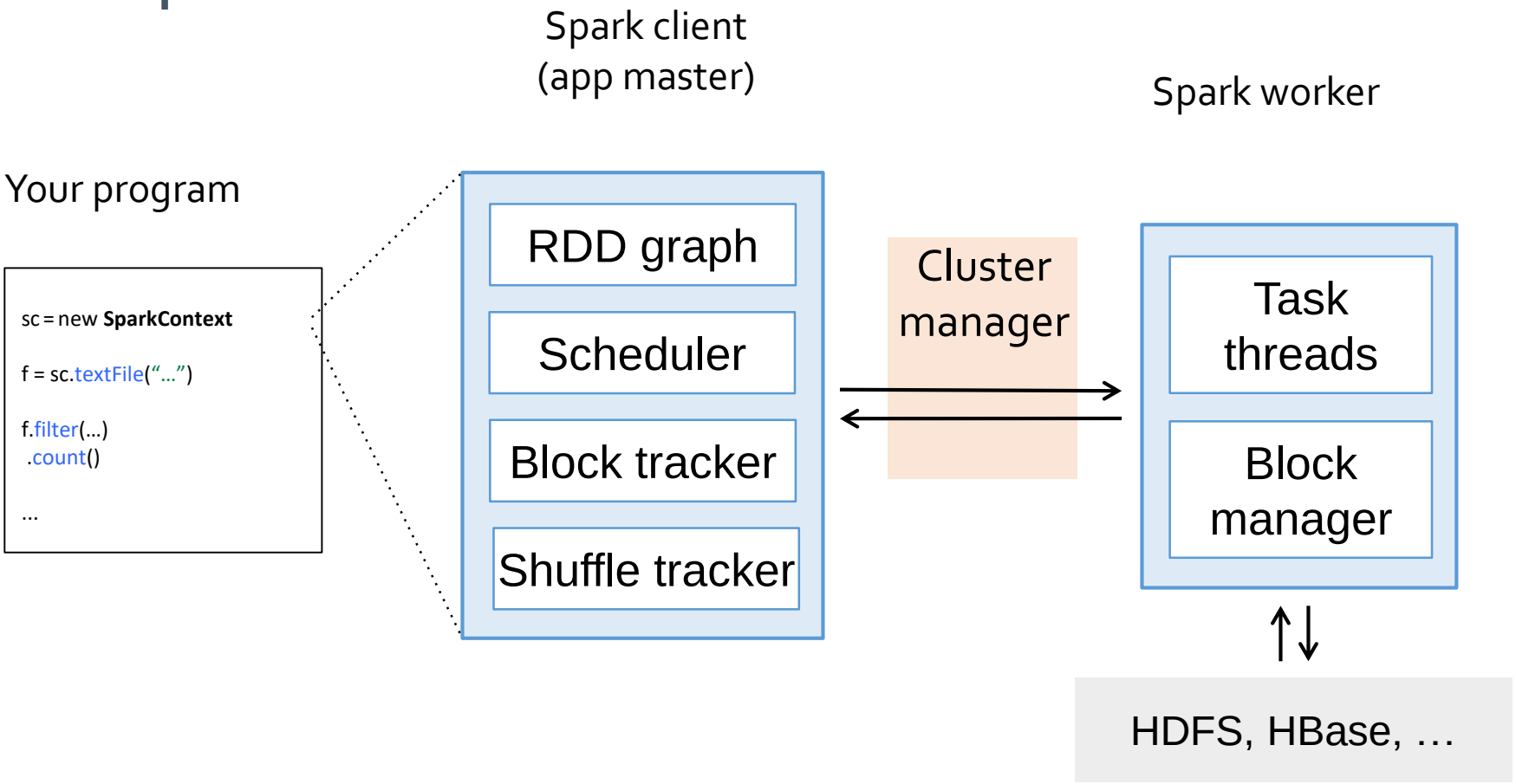
Software Components

- Spark runs as a library in your program
  - Runs tasks locally / on cluster*
    - Standalone, YARN, Mesos
    - See Cluster Mode Overview*
- Accesses storage systems via Hadoop API
  - Can use HBase, HDFS, S3, …



*=http://spark.apache.org/docs/latest/cluster-overview.html

Components

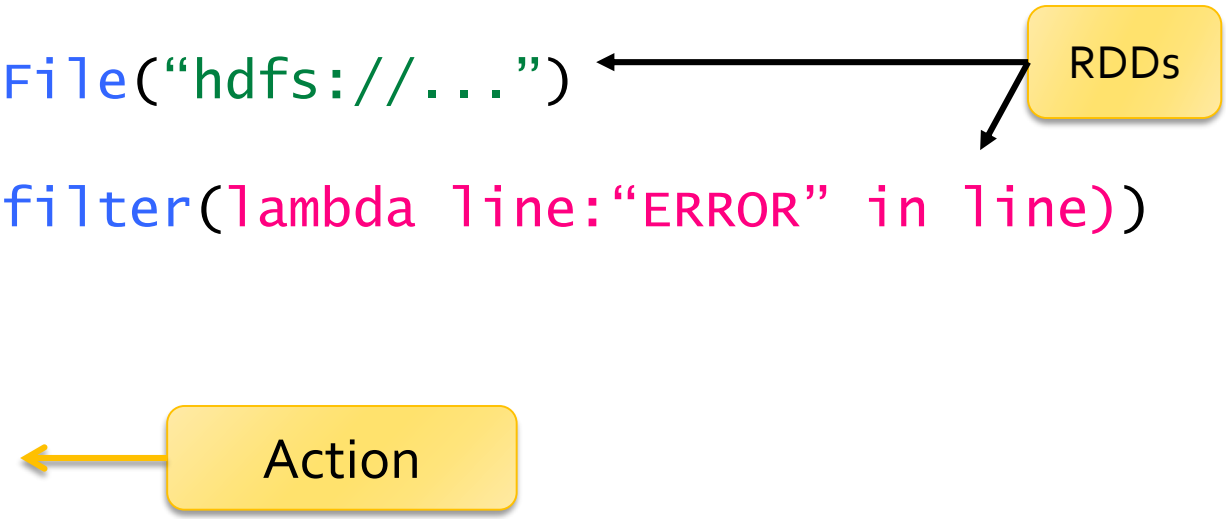


For more info see video "Introduction to AmpLab Spark Internals" (https://www.youtube.com/watch?v=49Hr5xZyTEA) and read slides http://files.meetup.com/3138542/dev-meetup-dec-2012.pptx

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Example Job

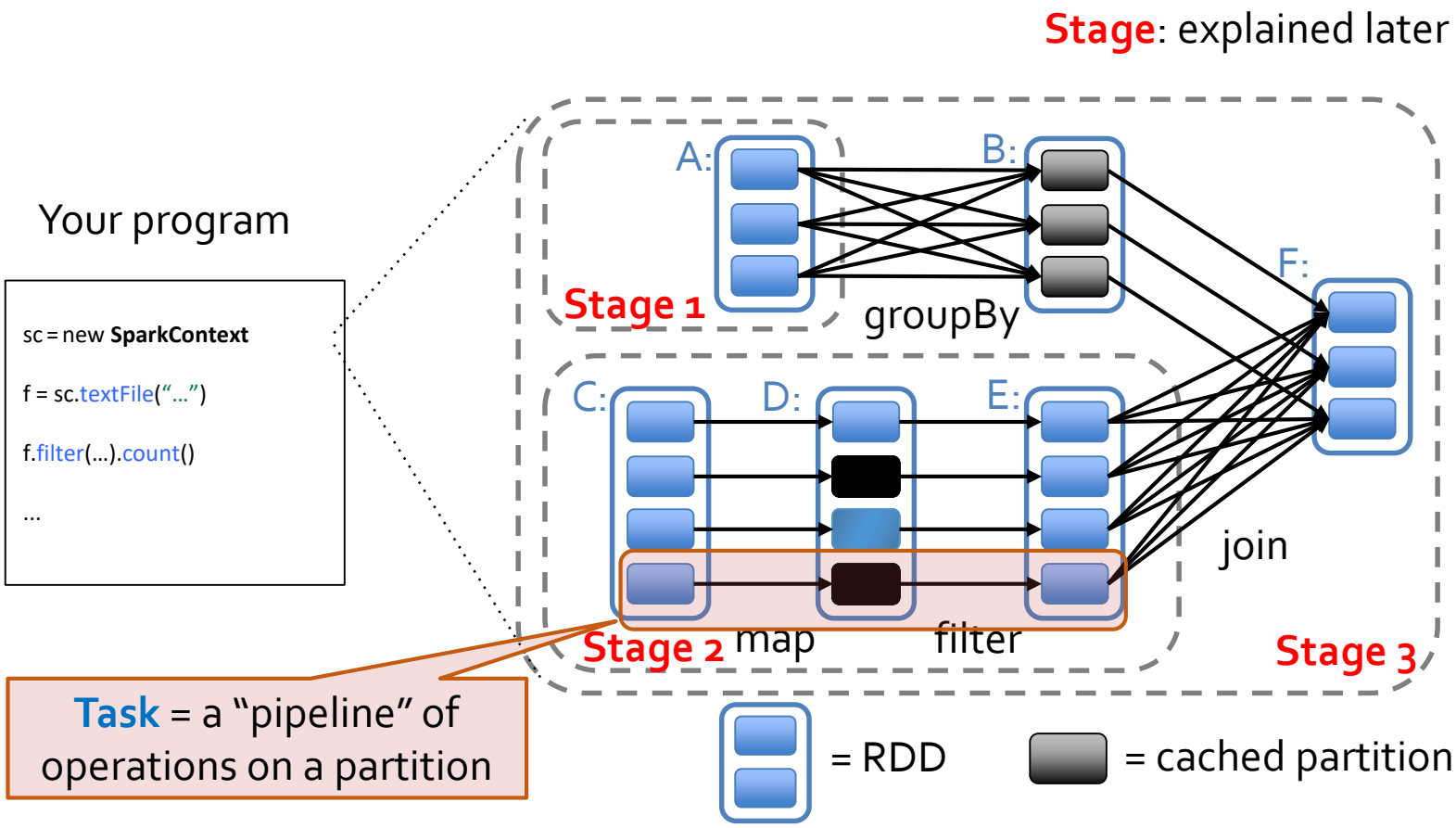
```
sc = SparkContext (appName="PythonExample")

file = sc.textFile("hdfs://...")

errors = file.filter(lambda line:"ERROR" in line))

errors.cache()

errors.count()
```



From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Operator DAG

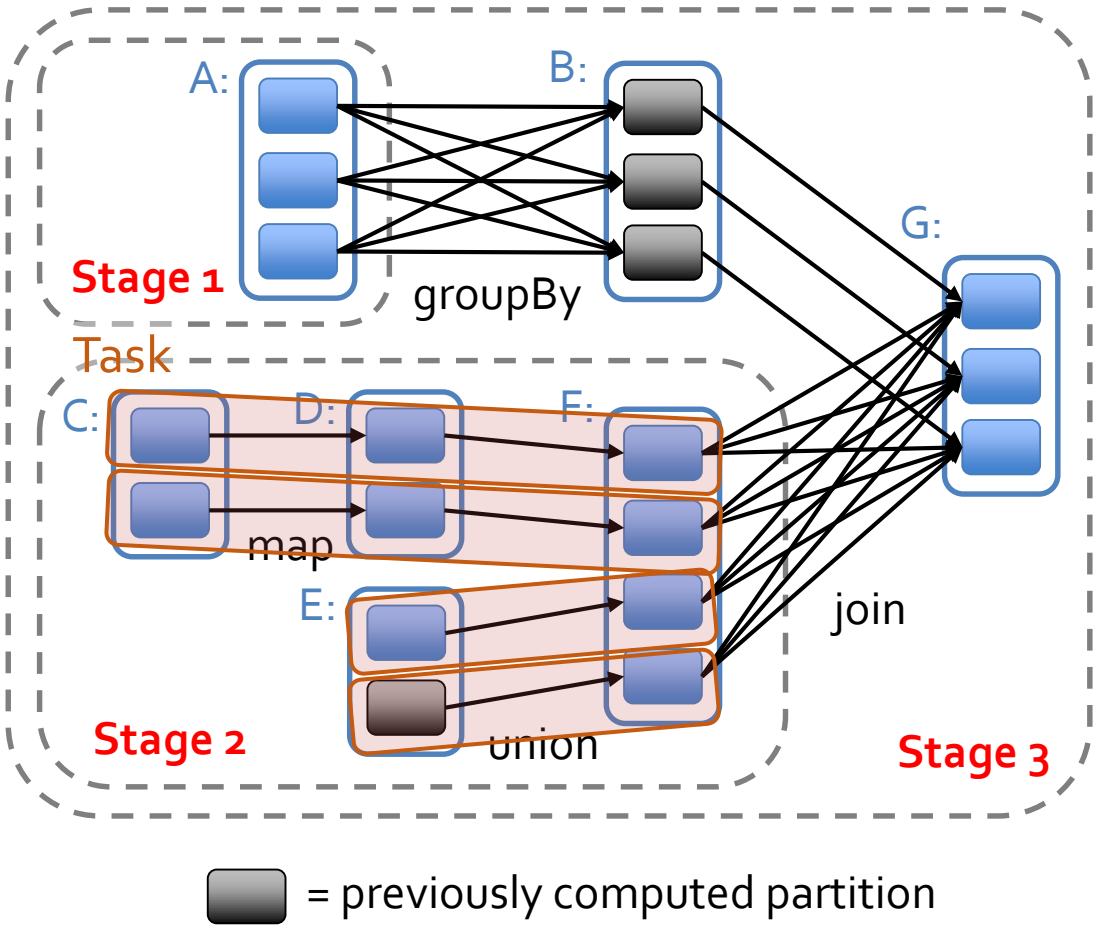
The **operator DAG** (Directed Acyclic Graph) captures RDD dependencies



From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

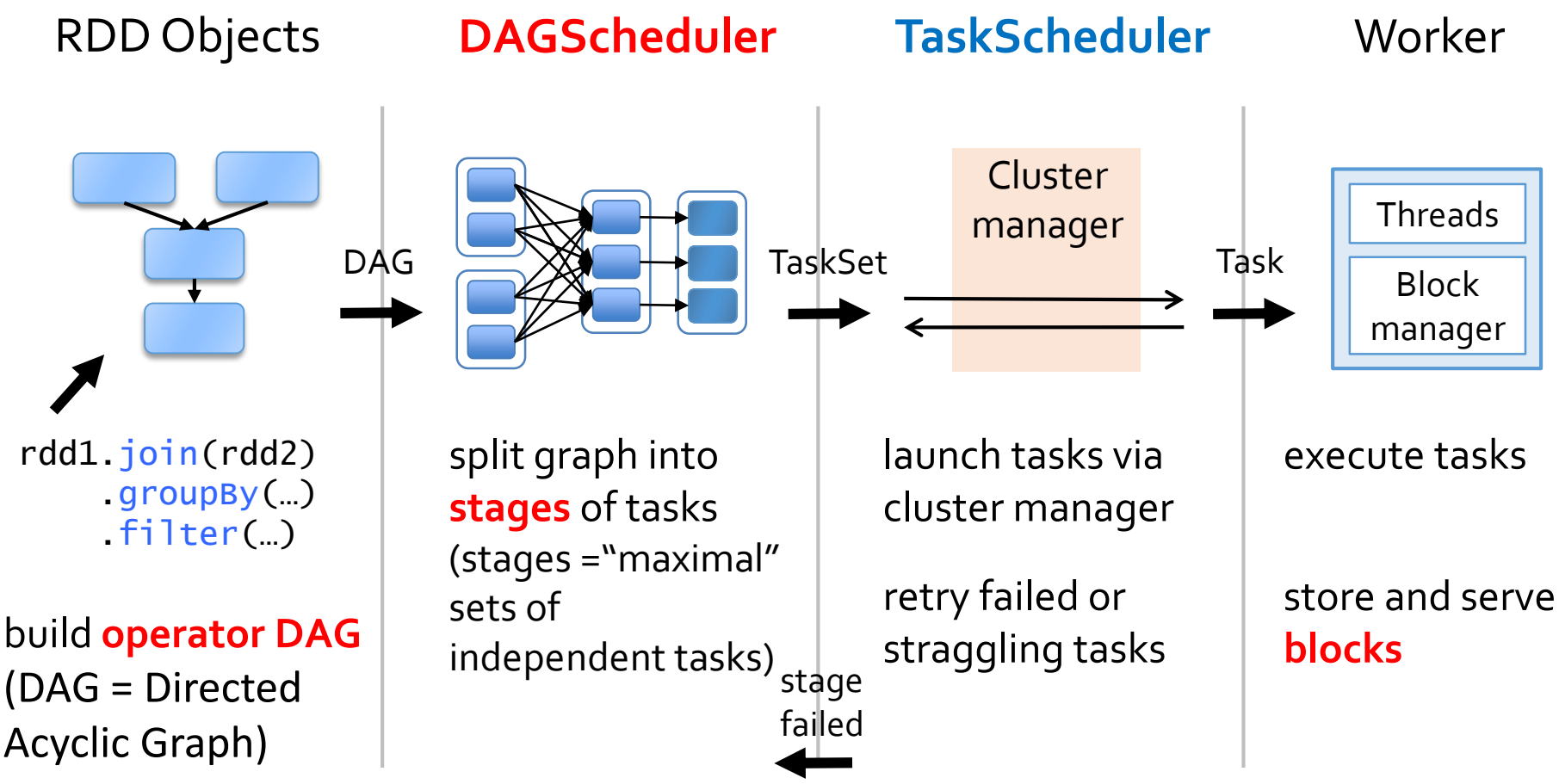
Stages

- A set of independent tasks, as large as possible
- Stage boundaries are only at input RDDs or “shuffle” operations (like groupBy*, join, …)



From: *Introduction to Spark Internals,* **Matei Zaharia**, 2012

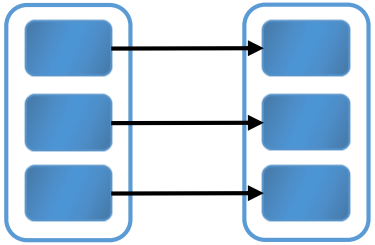
Scheduling Process



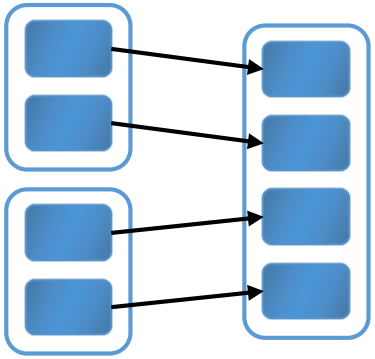
From: *Introduction to Spark Internals,* **Matei Zaharia**, Meetup Dec 2012

Dependency Types in DAG

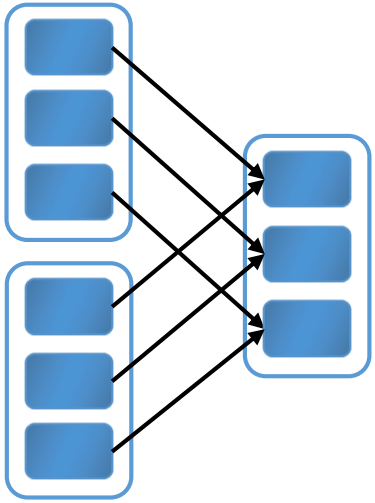
"Narrow" dependencies:



map, filter

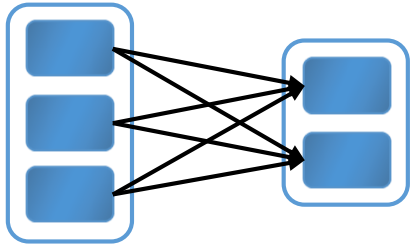


union

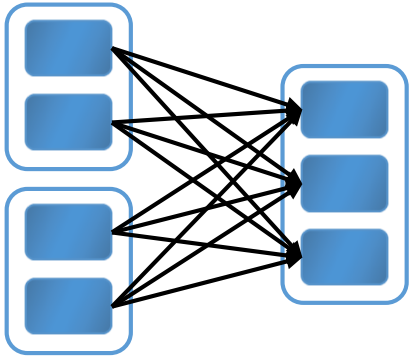


join with inputs co-partitioned

"Wide" (shuffle) deps:



groupByKey

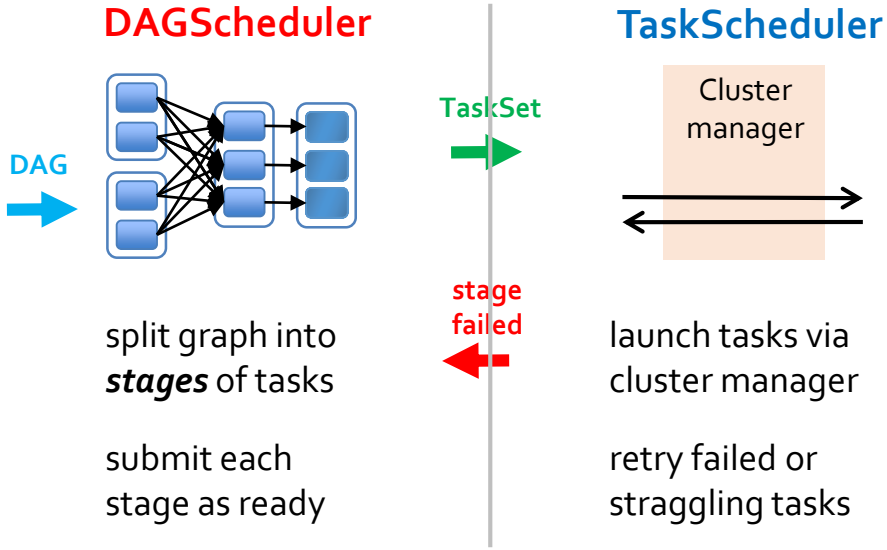


join with inputs not co-partitioned

From: *Introduction to Spark Internals,* **Matei Zaharia**, Meetup Dec 2012

DAG Scheduler vs. Task Scheduler

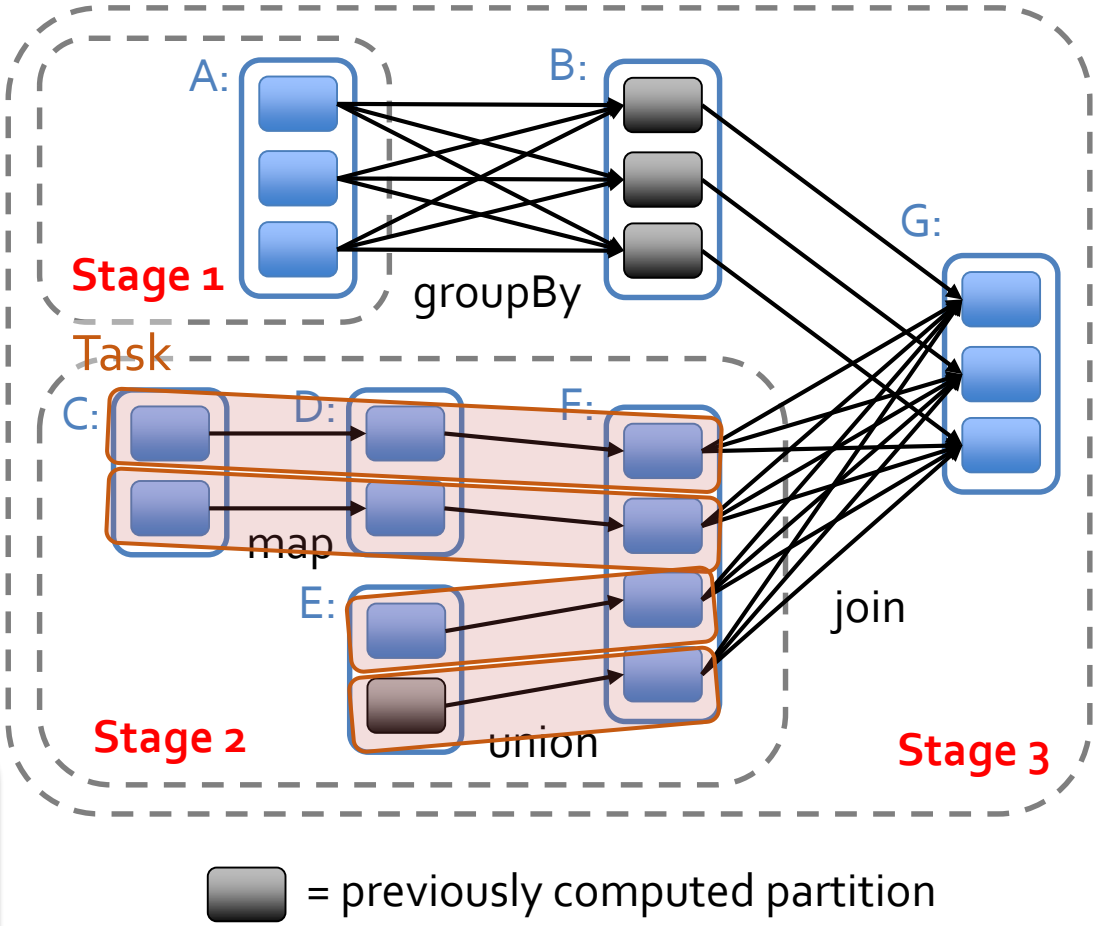
- DAG Scheduler – "higher level"
  - Builds stages of task objects (by code + preferred location)
  - Submits them to TaskScheduler as ready
  - Resubmits failed stages if outputs are lost
- **TaskScheduler**
  - "Lower level" – similar to Hadoop master
  - Given a set of tasks, runs it and reports results
  - Exploits data locality
  - Local / cluster implementation



From: *Introduction to Spark Internals,* **Matei Zaharia**, Meetup Dec 2012

Scheduler Optimizations

- Pipelines narrow ops. within a stage
- Picks join algorithms based on partitioning (minimize shuffles)
- Reuses previously cached data

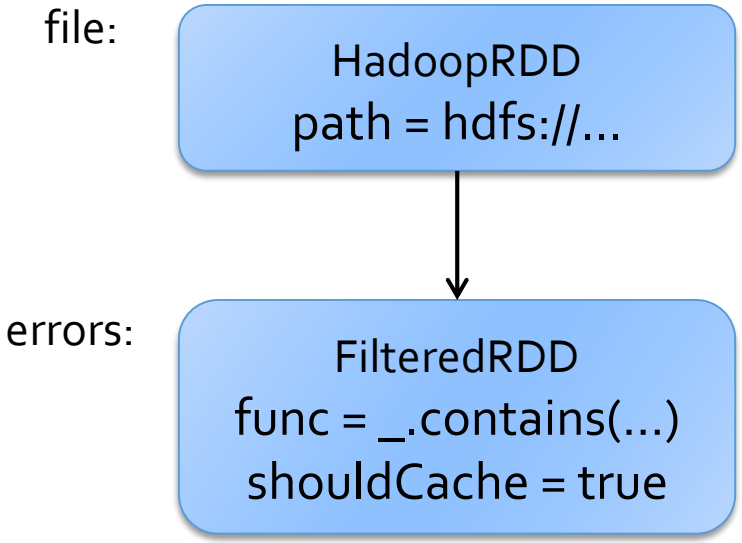


In MapReduce, each M-R phase is "individual" => Less optimization!

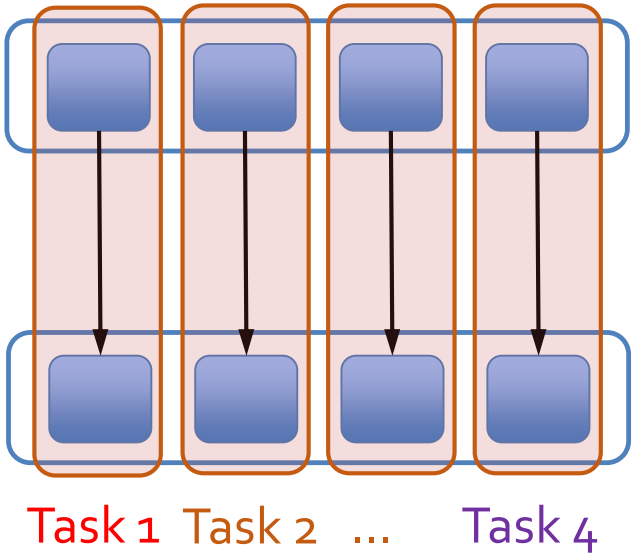
From: *Introduction to Spark Internals,* **Matei Zaharia**, 2012

RDD Graph

Dataset-level view:



Partition-level view:



- **Partition**: a subset of RDD, usually corresponding to a block of HDFS (or other file system)
- **Task**: a "pipeline" of operations on a single partition

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

RDD Interface

- Set of partitions (“splits”)
- List of dependencies on parent RDDs
- Function to compute a partition given parents
- Optional preferred locations
- Optional partitioning info (Partitioner)

Captures all current Spark operations!

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Example: HadoopRDD

- partitions = one per HDFS block
- dependencies = none
- compute*(partition)* = read corresponding block

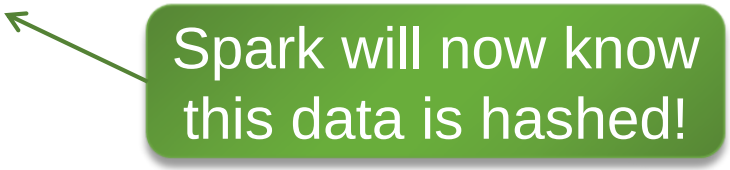
- preferredLocations*(part)* = HDFS block location
- partitioner = none

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Example: JoinedRDD

- partitions = one per reduce task
- dependencies = “shuffle” on each parent
- compute*(partition)* = read and join shuffled data

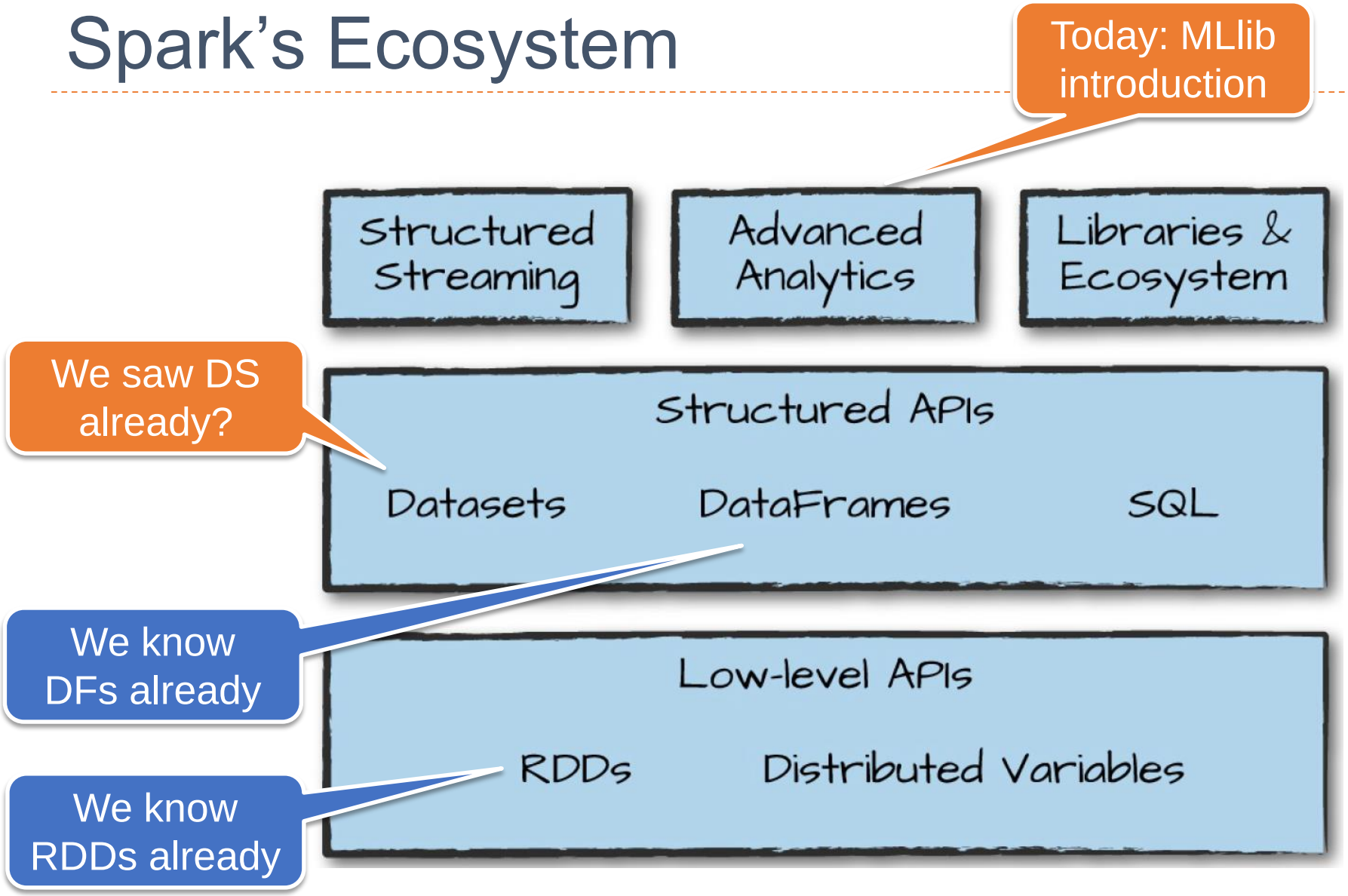
- preferredLocations*(part)* = none
- partitioner = HashPartitioner(numTasks)



From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Structured APIs in Spark: DataFrames vs. Datasets

Spark's Ecosystem



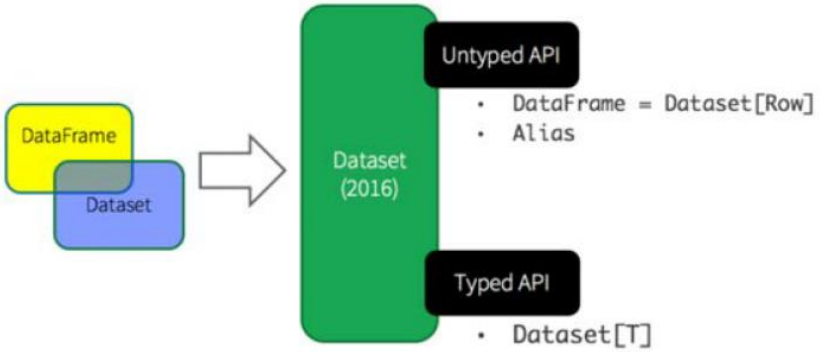
From *Spark: The Definitive Guide*, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

Structured APIs in Spark

- Structured APIs allow manipulating all sorts of data, from unstructured (e.g. log files) to highly structured
- There are three core types of them:
  - DataFrames (seen before)
  - SQL tables and views (seen before)
  - Datasets
- **Datasets** are type-safe Structured APIs
  - Think of them as essentially "typed" DataFrames
  - Spark checks whether data types are valid during code compilation, before execution starts
  - **(!)** Datasets are *only available to Java Virtual Machine* (JVM)*–based languages*, i.e. Scala and Java

Datasets vs. DataFrames

- In reality, DataFrames are special cases of Datasets
  - Spark 2.0 unified DFs and datasets

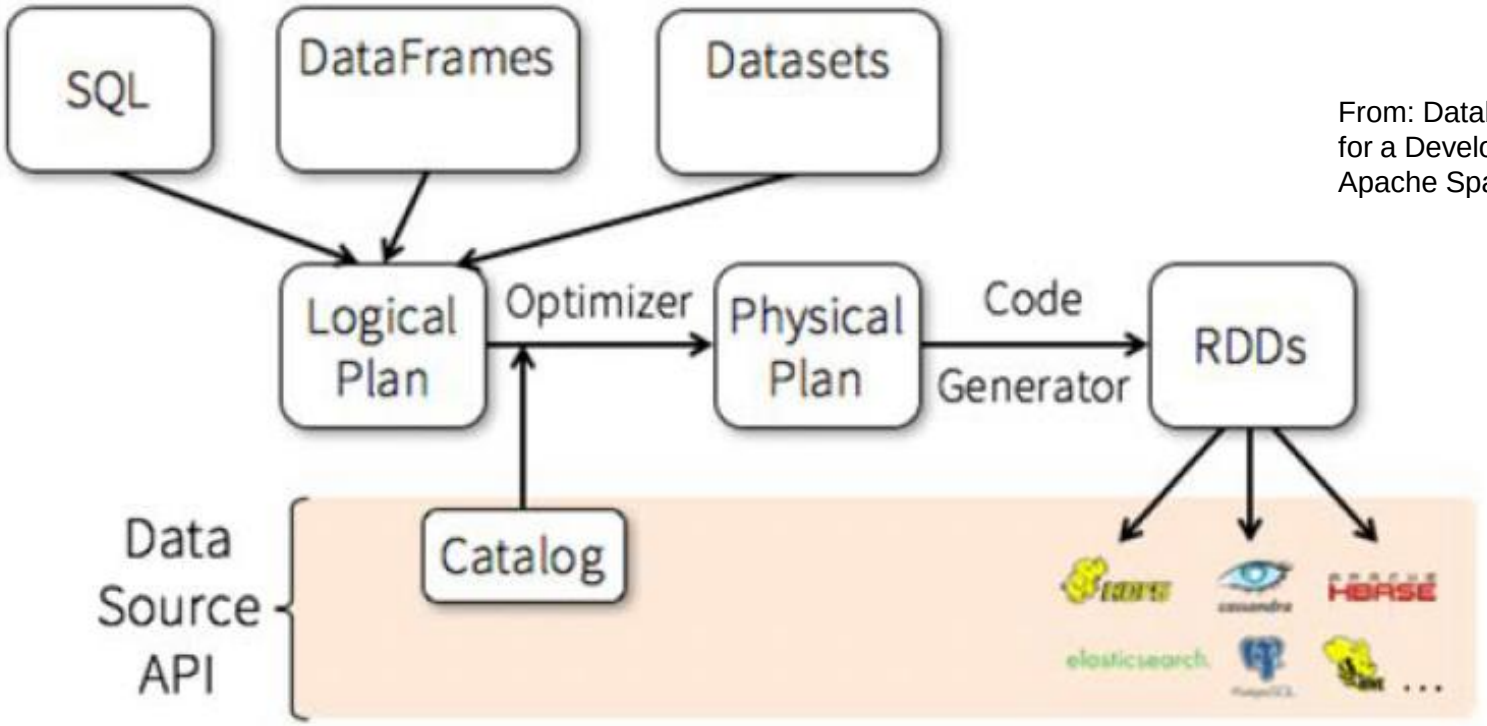


- Major differences:
  - For DFs, Spark only checks whether DF types line up to those specified in the schema at runtime
  - For Datasets, this happens earlier, at compile time

	Spark SQL	DataFrame	Dataset
Syntax errors	Runtime	Compile Time	Compile Time
Analysis errors	Runtime	Runtime	Compile Time

DataFrame APIs Allow Faster Execution

- APIs for DFs/Datasets provide high-level operators like sum, count, avg, min, max etc. (seen for DFs)
- Spark translates these into highly-efficient code – execution can be faster than for RDDs!



From: Databricks, 7 Steps for a Developer to Learn Apache Spark, 2017

APIs as a Domain Specific Language (DSL)

- You can regard DF/Datasets APIs as a special „programming language“: a **Domain Specific Language**
- Example (in Scala):

```
// dataset with field names fname, lname, age, weight
// access using object notation
val seniorDS = peopleDS.filter(p => p.age > 55)
// dataframe with columns fname, lname, age, weight
// access using col name notation
val seniorDF = peopleDF.where(peopleDF(“age”) > 55)
// Spark SQL code (equivalent)
val seniorDF = spark.sql("SELECT age from person where
                          age > 35")
```

From: Databricks, 7 Steps for a Developer to Learn Apache Spark, 2017

More Resources on Structured APIs

- Jeffrey Aven: Sams teach yourself Apache Spark in 24 hours, 2017, http://katalog.ub.uni-heidelberg.de/cgi-bin/titel.cgi?katkey=68102164 (free from univ. domain!)
- Databricks, 7 Steps for a Developer to Learn Apache Spark, 2017, https://goo.gl/chn8GE
- Spark Documentation: Spark SQL, DataFrames and Datasets Guide, https://goo.gl/HuHNEq
- Ankit Gupta: Complete Guide on DataFrame Operations in PySpark, 2016, https://goo.gl/mrNL4i
- Michael Armbrust, Wenchen Fan, Reynold Xin and Matei Zaharia: Introducing Apache Spark Datasets, 2016, https://goo.gl/VLu9dn