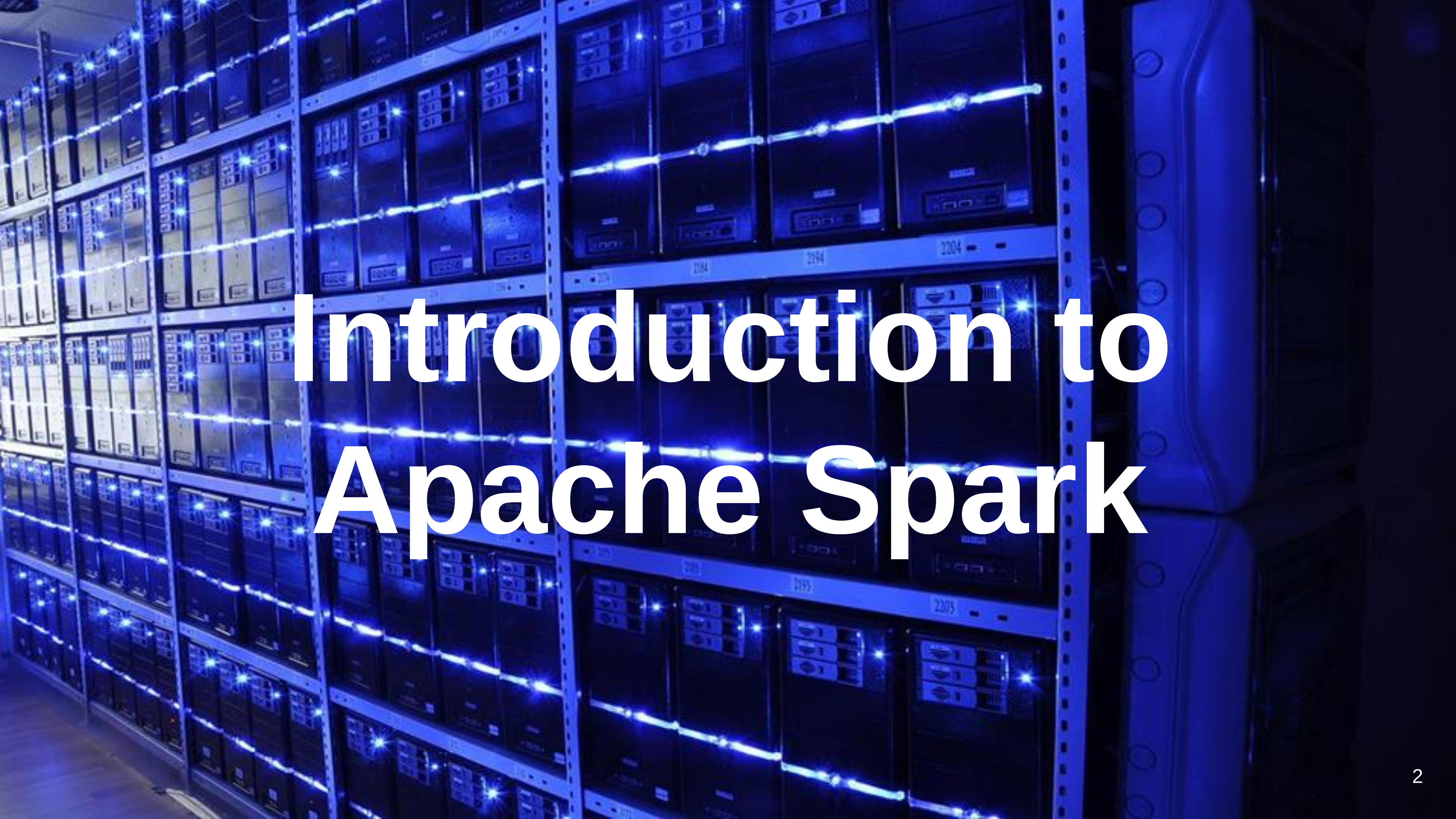
Data-Intensive Computing

2.0 VU / 3.0 ECTS, 2025S

Lecture 5 - Large-Scale Machine Learning Analytics with Apache Spark + Assignment 2

Dr. Alessandro Tundo
alessandro.tundo@tuwien.ac.at

Credits for the original version of slides to P. Knees and D. Winkler



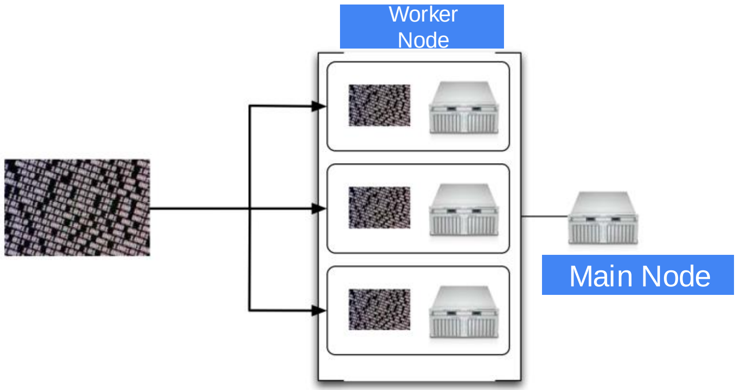
Introduction to Apache Spark

Apache Hadoop: Main Motivations

- Abstracts much of the complexities in distributed data processing applications

- As a developer:
  - you only needs to specify what needs to be done
  - not worry about system-level challenges such as:
    - coordination
    - message passing
    - race conditions
    - data starvation
    - data partitioning
    - code distribution etc.

- Focus on application development and business logic



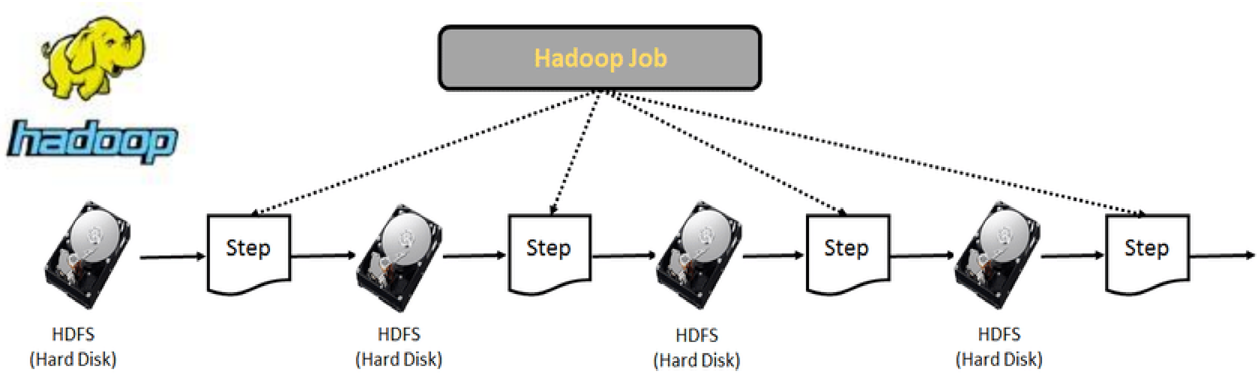
...Hower there's no free lunch! Limitations?!

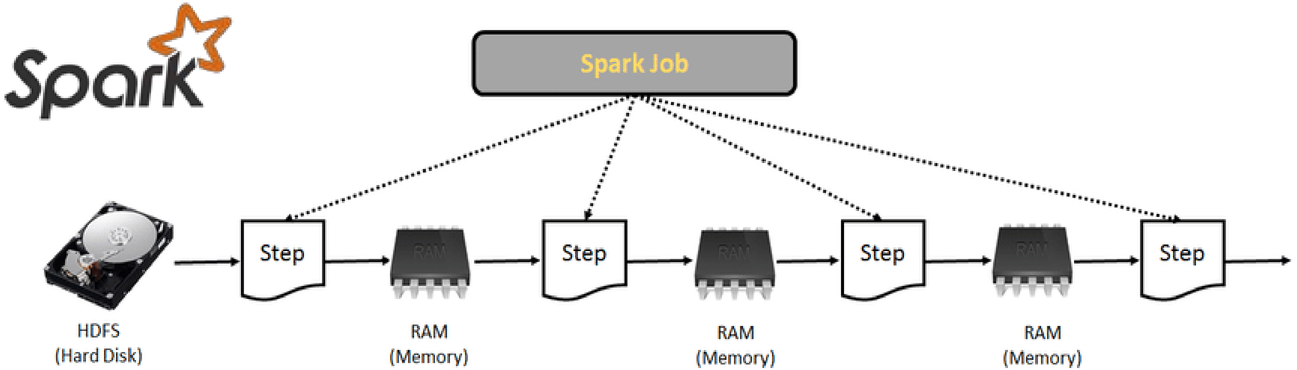
- Good for one-pass computation, not for multi-pass algorithms
- Not intended for interactive use (Hive, etc. batch only)
- APIs: tedious for analytic applications

- Forces data analysis into map and reduce steps:
  - analytic tasks often require complex chains of MR jobs

- **Data from disk must be re-loaded for each MR job:**
  - very inefficient for iterative algorithms (e.g., machine learning)
  - may spend 90% of the time doing I/O

Apache Hadoop Job *VS.* Apache Spark Job

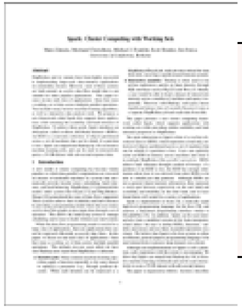




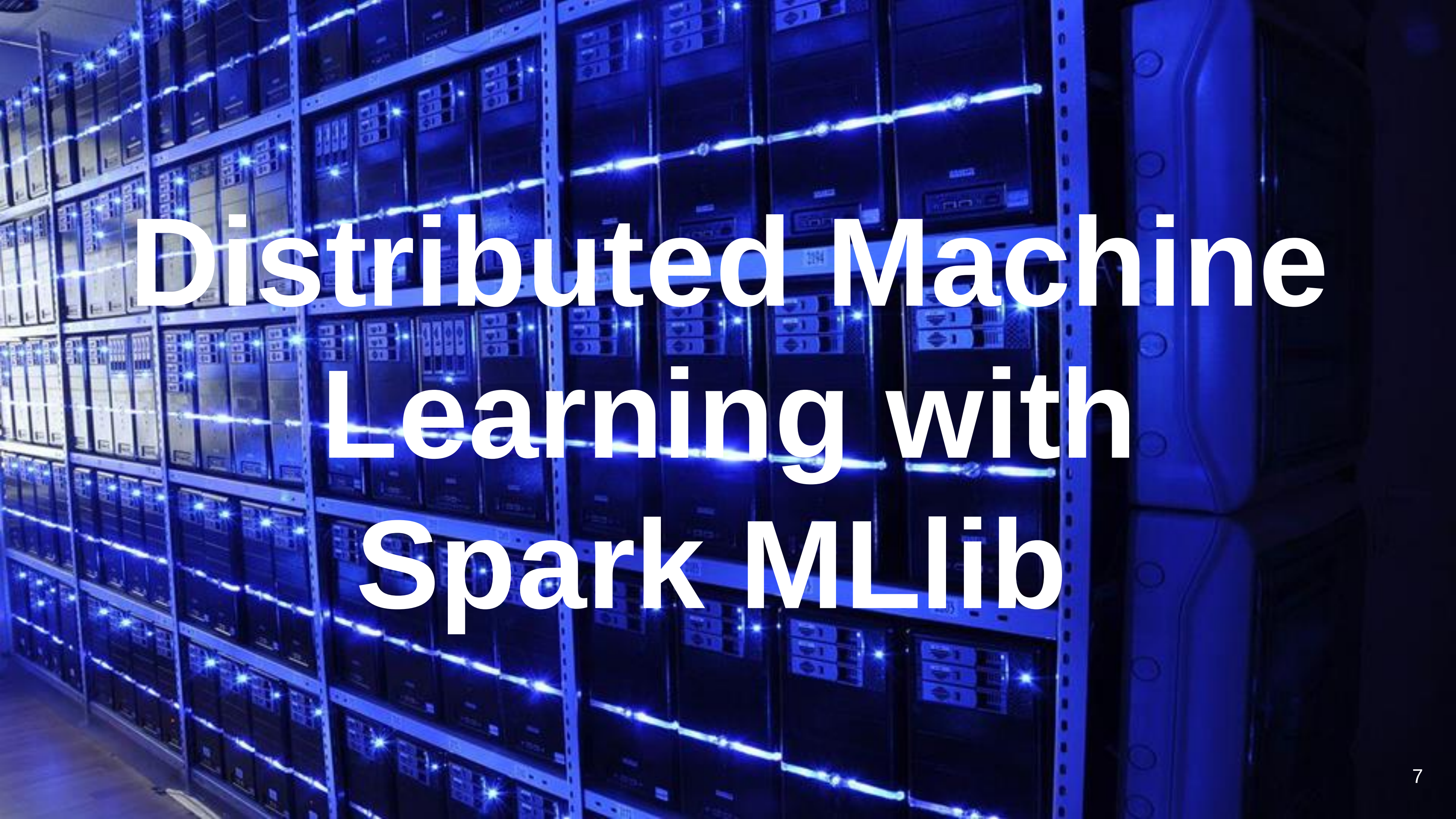
Bajaber F., El Shawi R., Batarfi O., Altalhi A., Baranwi A., Sakr S.: „Big Data 2.0 Processing Systems: Taxonomy and Open Challanges“, Journal of Grid Computing 14(3), 2016.

Apache Spark: The Origins

- General-purpose data-centric cluster computing system
- Promises **large speedups** compared to MR in iterative applications
- History:
  - 2009 Developed at UC Berkeley
  - 2010 open sourced under BSD license
  - 2013 donated to Apache Software Foundation 2014 top-level Apache project
- High-level APIs: Java, Scala, Python and R
- Interactive shells: Scala (`spark-shell`) and Python (`pyspark`)



M. Zaharia et al.
"Spark: Cluster Computing with Working Sets,"
Proc. of the 2nd USENIX conference on Hot topics in cloud computing, June 2010



Distributed Machine Learning with Spark MLlib

Spark in Industry

- Currently most active open source community in big data
- Used for many internet-scale machine learning problems
- 200+ developers, 50+ companies contributing

Example business use cases:

- Microsoft Bing – Spark Streaming to merge tens of TBs of query events and click events
  - per hour, Office365 analytics [1]
- Facebook: entity ranking with 60 TB+ (migrated from Hive). [2]
- Uber: Spark Streaming to process TBs of event data
- Netflix: streaming and machine learning
- Financial industry (e.g., Credit Fraud Prevention)

See here an updated list of projects and companies: https://spark.apache.org/powered-by.html

[1] Spark Streaming at Bing Scale; Spark Summit talk; https://www.youtube.com/watch?v=LrjKnGPXz14
[2] Apacke Spark @ Scale: https://code.facebook.com/posts/1671373793181703/apache-spark-scale-a-60-tb-production-use-case/

Spark in Scientific Computing

Roots in research, but more widely adopted in industry

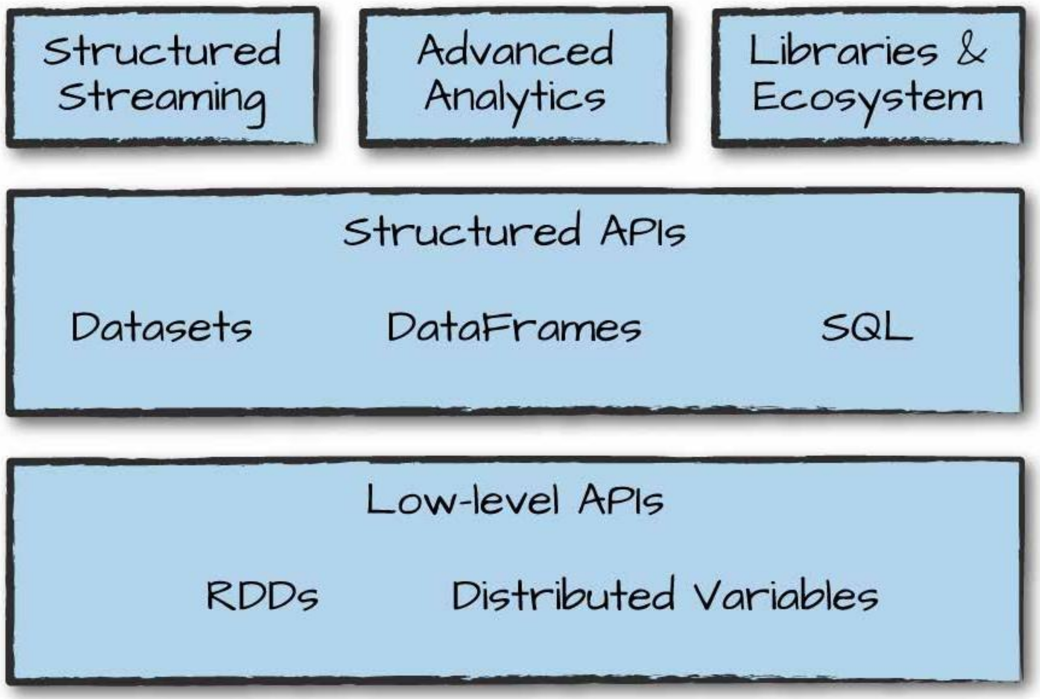
Example research use cases:

- **Kira**[1]: distributed astronomy image processing toolkit using Apache Spark
- **Hail**[2]: Scalable Genomic data analysis with Spark
- **SciSpark**[3] (NASA): e.g. for weather event detection

Scientific data analysis: NASA, CERN, Broad Institute of MIT, Harvard [4]

[1] Zhao et al. (2015): Scientific Computing Meets Big Data Technology: An Astronomy Use Case, https://arxiv.org/abs/1507.03325
[2] Hail: https://github.com/hail-is/hail
[3] SciSpark: https://pdfs.semanticscholar.org/999e/4cc75b0d9bcfba019a0538cc318eb6a4aec2.pdf
[3] Spark Guide: https://learning.oreilly.com/library/view/spark-the-definitive/9781491912201/ch01.html

Spark's toolkit

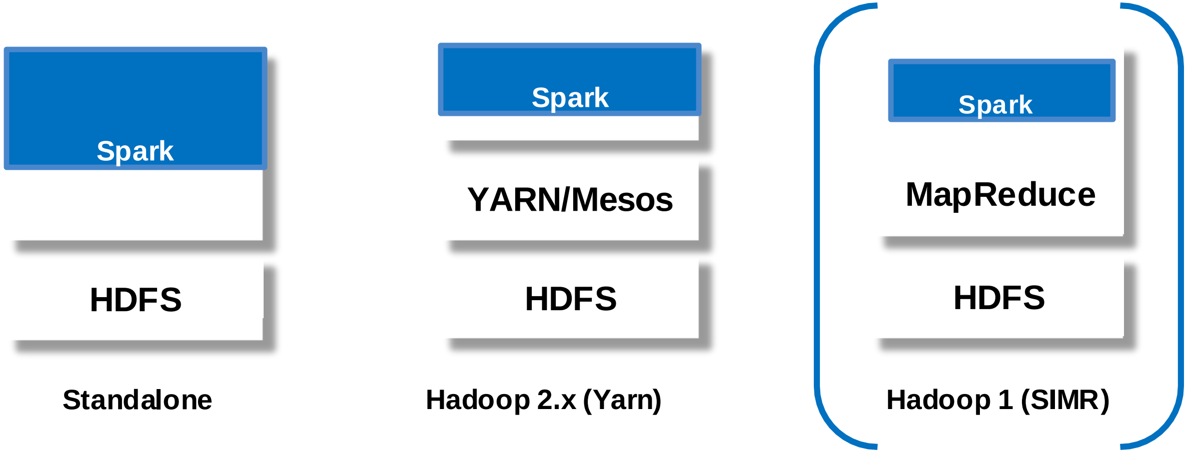


Spark Guide: https://learning.oreilly.com/library/view/spark-the-definitive/9781491912201/ch01.html

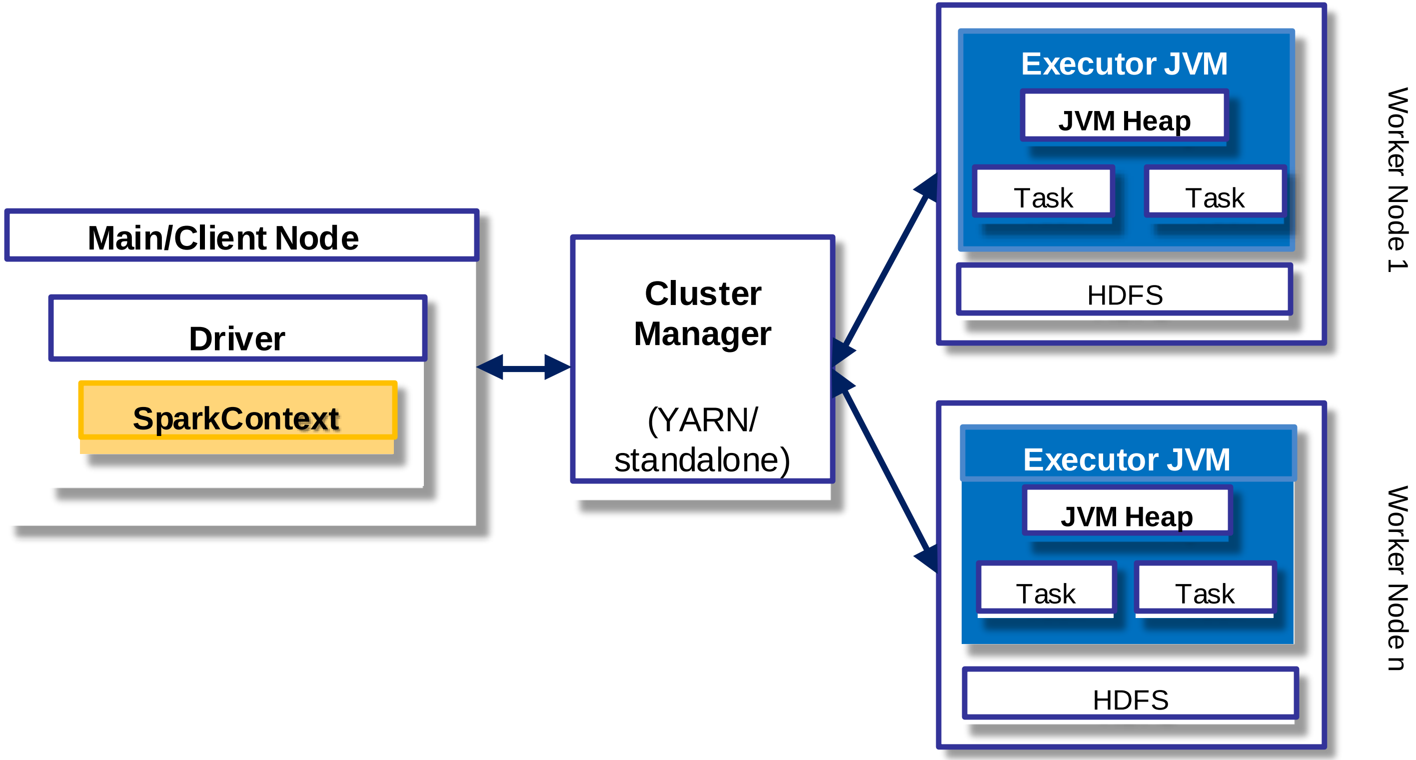
Where and how can you run Spark?

- Integrates into Hadoop ecosystem (HDFS, YARN)
- Can access data from **HDFS, HBase, Hive** (+ Cassandra, S3, Tachyon, ..)

Hadoop integration:



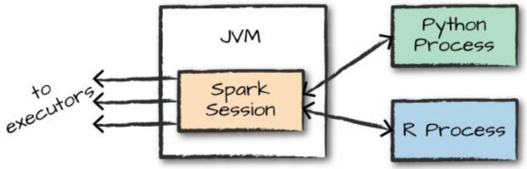
Spark Architecture (simplified)



Spark Language Choices

Java/Scala:

- Typically significantly faster than Python (native implementation, static typing)
- Scala is more verbose than Python, but less verbose than Java
- Scala supports REPL (Read-Evaluate-Print Loop)



Python:

- More compact, easier to use (?)
- Excellent for interactive development (Jupyter,..) and as a "glue" language
- Wide range of data science libraries (e.g., NumPy for numerical work, SciPy for scientific computing, Pandas for data munging, matplotlib for visualization,..)
- PySpark API often lagging behind the native Scala implementation a bit
- Python wrapper calls underlying Spark code written in Scala
  → potential for bugs, tedious debugging,..

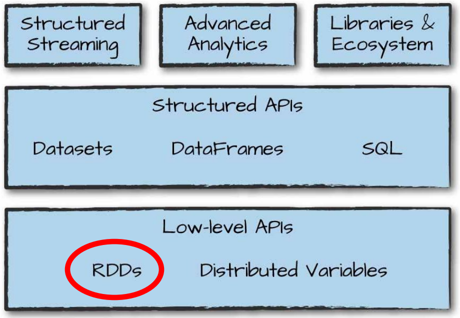
R:

- SparkR: R package that provides a light-weight frontend to use Spark from R
- Widely used language for data analysis
- Significantly less developed than PySpark API

Figure source: https://learning.oreilly.com/library/view/spark-the-definitive/9781491912201/ch02.html

What is an RDD?

- RDD = **“Resilient Distributed Dataset”**
- Fundamental data structure
- Conceptually, a distributed collection of elements (think: distributed array or list)
- Spread out across multiple nodes in the cluster
- An RDD represents *partitioned* data
- Within your program (the Driver), an `RDD` object is a *handle* to that distributed data



Working with RDDs

Creating RRDs:

1. Parallelized collections: `parallelize` method in the driver
2. External datasets: local files, HDFS, Cassandra, HBase..
3. From existing RDDs through transformations

(+ Spark Streaming)

Operations on RDDs:

- *Transformations* (wide and narrow)
- *Actions*: do not produce a new RDD, typically used to obtain final result

Creating RDDs: Parallelize

Take an existing in-memory collection and pass it to SparkContext's parallelize method...

Scala

```
val wordsRDD = sc.parallelize(List(“fish”, “cats”,
“dogs”))
```

Java

```
JavaRDD<String> wordsRDD =
          sc.parallelize(Arrays.asList(“fish”, “cats”, “dogs”));
```

Python

```
wordsRDD = sc.parallelize(["fish", "cats", "dogs"])
```

Not generally used outside of prototyping and testing, since it requires entire dataset in memory on one machine (→ driver = bottleneck)

Source: adapted from Sameer Farooqui: Advanced Apache Spark Training (Spark Summit 2015)

Creating RDDs from external source

Python

```
# Turn a local collection into an RDD with 3
partitions integer_rdd = sc.parallelize(range(10), 3)

integer_rdd.collect()
Out: [0,1,2,3,4,5,6,7,8,9]

integer_Rrdd.glom().collect()
Out: [[0,1,2],[3,4,5],[6,7,8,9]]

# Load text file from local FS, HDFS, or
S3 text_rdd = sc.textFile("file.txt")
text_rdd = sc.textFile("directory/*.txt")
text_rdd =
sc.textFile("hdfs://namenode:9000/path/file")

text_rdd.take(1) #output the first line

# Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Other methods to read data from C*, S3, Hbase etc.

Source: adapted from Matei Zaharia: "An Introduction to the Berkeley Data Analytics Stack – Part 1"

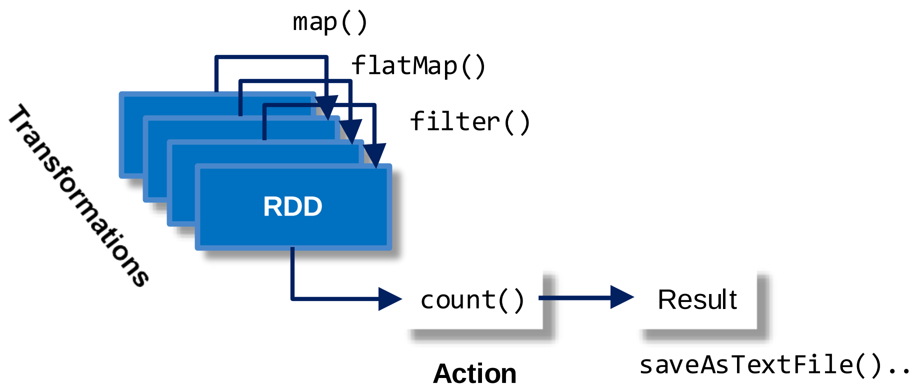
Parallelization in Spark

- Based on partitioned RDDs
  - each RDD split into multiple partitions
  - provide a **very restricted *distributed shared memory*** that only allows a limited set of transformations
  - sequence of transformations rather than fine-grained updates to a shared state

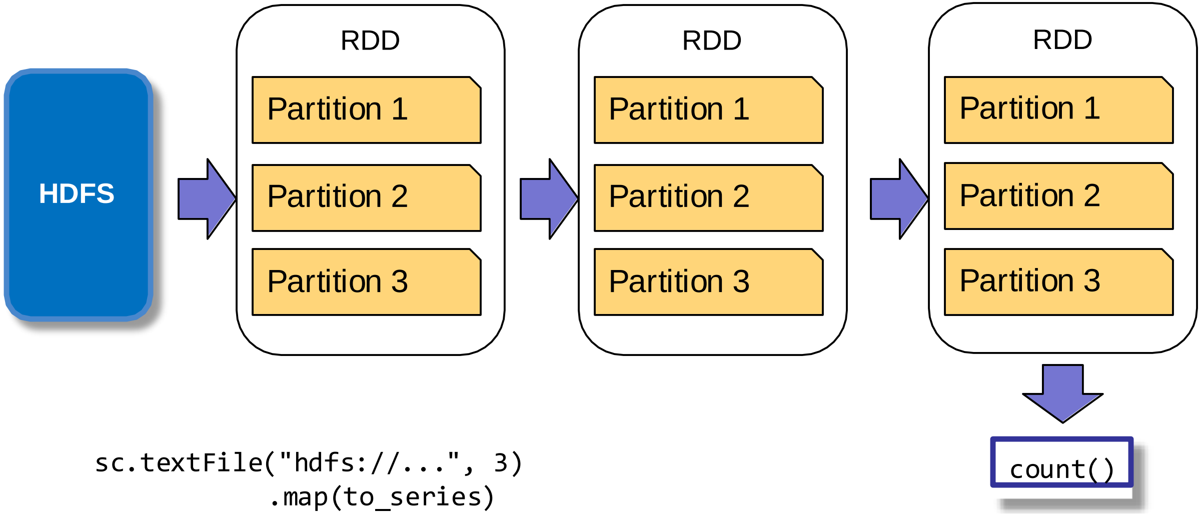
- **Data-centric:** Spark exposes RDDs through language-integrated API, no arbitrary direct access to shared memory
- **Locality brings processing to the data**
- **DAG and pipelining minimizes coordination overhead** and data exchange over the network (only necessary for “wide transformations”)

Transformations and Actions

- Transformations are **functions that produce new RDDs** from an existing RDD
- **Sequence of transformation steps** creates a lineage graph
- Used to create a logical execution plan (= Directed Acyclic Graph)
- Programmer can control persistence of RDDs (memory, disk,..)



RDD Example

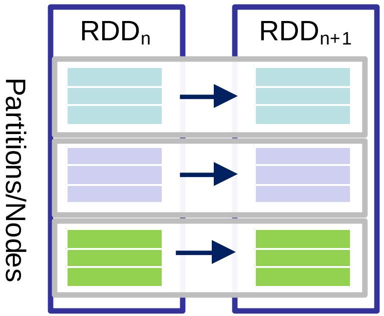



```
sc.textFile("hdfs://...", 3)
            .map(to_series)
            .filter(has_outlier)
            .count()
```

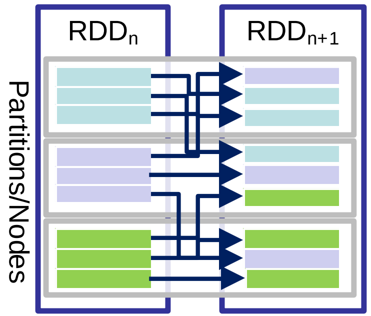
Types of RDD Transformations

RDDs created by a transformation can be

- smaller (e.g., `filter`, `count`, `dinstinct`, `sample`)
- larger (e.g., `flatMap`, `union`, ..)
- same size (e.g., `map`)

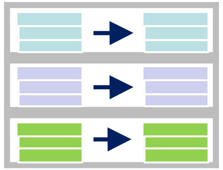


Narrow
(e.g., `map`)



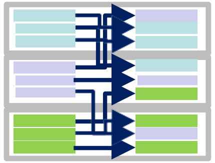
Wide
(e.g., `groupByKey`)

Narrow Transformations



- `map(func)`
  - apply function to each element of an RDD
  - from partition to same-sized partition
- `flatMap(func)`
  - multiple output elements for each input element
  - e.g., split input string into words
- `mapPartition(func)`
  - Like map, but applied to each partition of the RDD
- `filter(func)`
  - keeps only elements where func is true
- `coalesce(numPartitions)`
  - FlatMap, filter.. lead to uneven partitions
  - Coalesce reduces the number of partitions
  - Tries to be narrow (may sometimes require network communication)
- ..

Wide Transformations



- `groupByKey()`
  - (K,V) pairs → (K, iterable of all V)
- `reduceByKey(func)`
  - (K,V) pairs → (K, result of reduction by func on all V)
- `intersection(otherRDD)`
  - Only the common elements of both RDD and other RDD
- `distinct()`
  - Distinct elements of the source dataset
- `join(otherRDD)`
  - combines two pair RDDs on the basis of the key
- `repartition(numPartitions)`
  - Shuffles data to increase or decrease number of partitions to numPartitions
- ..

RDD Actions

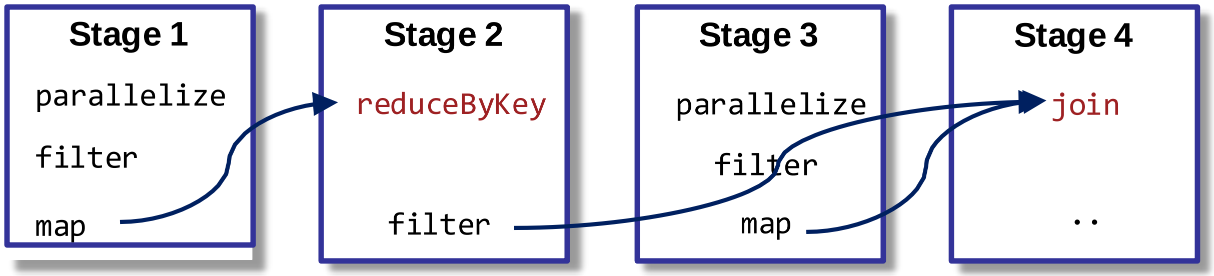
- Do not produce a new RDD, but materialize a value
- Returns final result of RDD computations to driver or external storage system
- Triggers execution using lineage graph

- **`Driver Actions:`**
  - `collect():` copy all elements to the driver
  - `take(n):` copy first n elements
  - `top(n):` copy top n elements
  - `count():` number of elements
  - `countByValue():` how many times each value occurs in the RDD

- **`Distributed Actions:`**
  - `reduce(func):`
    - aggregate elements with func;
    - takes two elements as input and produces output of the same type (e.g., addition)
  - `foreach(func)`
  - `saveAsTextFile()`
  - `aggregate, fold..`

Pipelining

- Narrow transformations are grouped into *stages*
- Wide transformations → stage boundaries
- Operator graph created from code
- DAG scheduler groups operations into stages
- Divided into tasks based on the partitions of the RDDs
- Task scheduler launches tasks through the cluster manager



SparkContext

- Main entry point to Spark functionality
- Created for you in Spark shells (typically as variable `sc`)
- Create your own in standalone programs
- Functionality:
  - Set Configuration
  - Access services (`TaskScheduler`, `BlockManager`, `SchedulerBackend`..)
  - Cancel jobs and stages
  - Cleanup (after action invocation)
  - Access persistent RDDs and unpersistent them
  - etc.

Creating a Spark Context

Scala

```
import spark.SparkContext
import spark.SparkContext._

val sc = new SparkContext(“masterUrl”, “name”, “sparkHome”,
Seq(“app.jar”))
```

Java

```
import
spark.api.java.JavaSparkContext;
JavaSparkContext sc = new JavaSparkContext(
    “masterUrl”, “name”, “sparkHome”, new String[] {“app.jar”}));
```

Python

```
from pyspark import SparkContext

sc = SparkContext(“masterUrl”, “name”, “sparkHome”, [“library.py”]))
```


Source: adapted from Matei Zaharia: “An Introduction to the Berkeley Data Analytics Stack – Part 1”

RDDs, DataFrames, Datasets: when should you use what?

- Use `DataFrame` (= `Dataset[Row]`) if..
  - .. you want rich semantics, high-level abstractions, and domain-specific APIs
  - .. your processing demands high-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access and use of lambda functions on semi-structured data => **Query friendly**
  - .. you are a Python or R user.
  - .. you need the highest possible speed.
  - .. for interactive analysis (e.g., Python Jupyter notebooks).
- Use `Dataset[T]` if..
  - .. you want type-safety at compile time and typed JVM objects
  - .. you need to optimize space efficiency.
  - .. you are a Scala or Java user.
- Use RDDs if..
  - .. you need low-level functionality and control (NOT SUGGESTED AND DEPRECATED!)

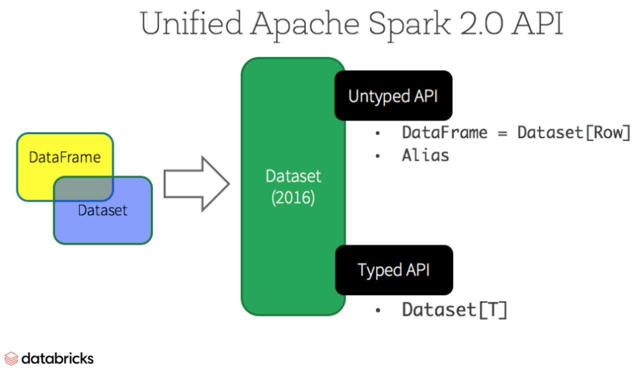


Figure: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Spark MLlib & Text Processing

ML Feature Extraction

- Common functionalities for machine learning built upon Datasets / DataFrames
- Feature extraction, transformation and selection, e.g.,
  - `Tokenizer / RegexTokenizer`
    - Splitting based on whitespaces or regular expression
  - `StopWordsRemover`
    - Removes words from defined language-specific dictionaries
  - `HashingTF / CountVectorizer`
    - Extraction of term frequencies
  - `IDF`
    - Applies inverse document frequency weighting to term frequency vectors
  - `ChiSqSelector`
    - Chi-square test for feature selection
  - `Normalizer`
    - Vector normalization to unit length

https://spark.apache.org/docs/latest/ml-features.html

ChiSqSelector Example

```
from pyspark.ml.feature import ChiSqSelector
from pyspark.ml.linalg import Vectors
from pyspark.sql import SparkSession

# Create Spark session
spark = SparkSession.builder.appName("ChiSqSelectorExample").getOrCreate()

# Create sample data
data = [
    (7, Vectors.dense([0.0, 0.0, 18.0, 1.0]), 1.0),
    (8, Vectors.dense([0.0, 1.0, 12.0, 0.0]), 0.0),
    (9, Vectors.dense([1.0, 0.0, 15.0, 0.1]), 0.0)
]

# Create DataFrame with specified column names
df = spark.createDataFrame(data, ["id", "features", "class"])

# Initialize and configure ChiSqSelector
selector = ChiSqSelector(
    numTopFeatures=1,
    featuresCol="features",
    labelCol="class",
    outputCol="selectedFeatures"
)

# Fit and transform the data
result = selector.fit(df).transform(df)

# Show results
result.show()
```

https://spark.apache.org/docs/latest/ml-features.html#chisqselector

Normalizer Example

```
from pyspark.ml.feature import Normalizer
from pyspark.ml.linalg import Vectors
from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder.appName("NormalizerExample").getOrCreate()

# Create DataFrame with sample data
data = [
    (0, Vectors.dense([1.0, 0.5, -1.0])),
    (1, Vectors.dense([2.0, 1.0, 1.0])),
    (2, Vectors.dense([4.0, 10.0, 2.0]))
]
df = spark.createDataFrame(data, ["id", "features"])

# Initialize Normalizer with L2 norm (p=2.0)
normalizer = Normalizer(
    inputCol="features",
    outputCol="normFeatures",
    p=2.0
)

# Apply transformation
l2_norm_data = normalizer.transform(df)

# Show results
l2_norm_data.show(truncate=False)
```

https://spark.apache.org/docs/latest/ml-features.html#normalizer

Spark MLlib & Classification

Spark ML: Classification

- Classification, experiment control, evaluation, parameter optimization
- `LinearSVC`, `Naïve Bayes`, etc.
    - Different types of classifiers (as well as regression algorithms)
- `OneVsRest`
    - Encapsulates binary classifiers for multiclass classification
- `Pipeline`
    - Sequence of `Transformers` and `Estimators`
- `ParamGridBuilder`
    - Parameter subspace definition to find optimal parameters
- `CrossValidator`, `TrainValidationSplit`
    - Testing strategies
- `BinaryClassificationEvaluator`, `MultilabelClassificationEvaluator`, `RegressionEvaluator`
    - Performance criteria

https://spark.apache.org/docs/latest/ml-features.html

LinearSVC Example

```python
from pyspark.ml.classification import LinearSVC

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

lsvc = LinearSVC() \
    .setMaxIter(10) \
    .setRegParam(0.1)

# Fit the model
lsvc_model = lsvc.fit(training)

# Print the coefficients and intercept
print(f"Coefficient: {lsvc_model.coefficients} Intercept: {lsvc_model.intercept}")
```

https://spark.apache.org/docs/latest/ml-classification-regression.html#linear-support-vector-machine

LinearSVC

- Implements the linear support vector classifier
- What is a (linear) Support Vector Machine?

- Supervised learning approach
- Used for (binary) classification and regression

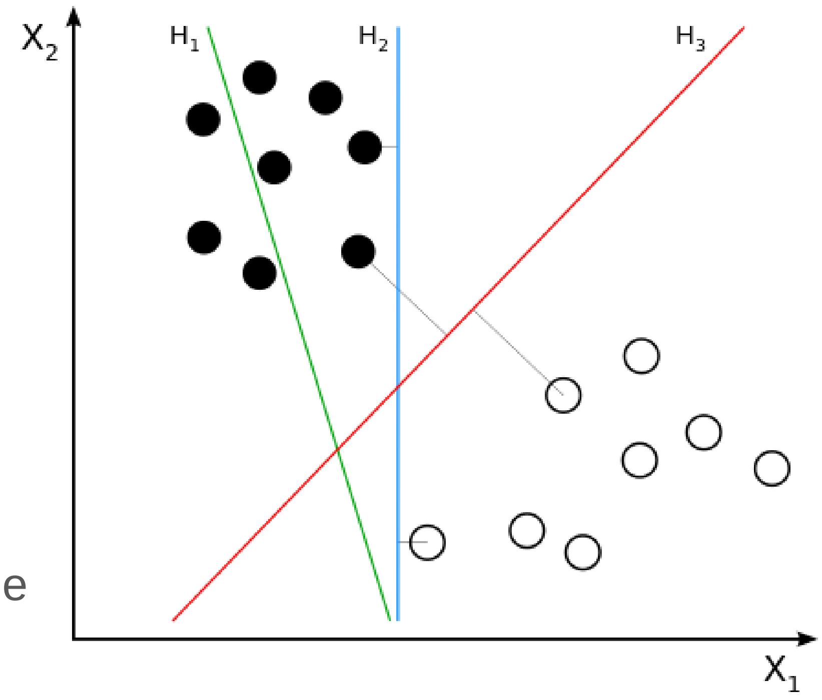
- Support-vector machine constructs a hyperplane or set of hyperplanes in an (implicit) high- or infinite-dimensional space
(→ *implicit* … cf. Kernel trick)
- Maps problem from feature space (where problem is typically not linearly separable) to this space

Hyperplane

- H_1 does not separate classes
- H_2, H_3 do

- Which one is better?

- H3 separates with larger margin (better generalization)
- Goal: find hyperplane separating classes with largest margin possible
  - (largest distance to the nearest training-data point of any class)



https://en.wikipedia.org/wiki/Support-

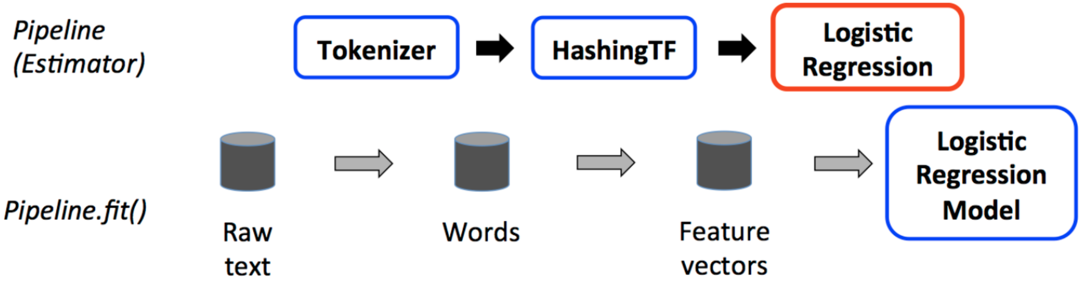
Multiclass SVM Example

```
from pyspark.ml.classification import LinearSVC, OneVsRest
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Load data
input_data = spark.read.format("libsvm").load("...")

# Split into training and test sets
train, test = input_data.randomSplit([0.8, 0.2])

# Initialize base classifier
classifier = LinearSVC() \
    .setMaxIter(10) \
    .setRegParam(0.1)

# Create OneVsRest wrapper
ovr = OneVsRest(classifier=classifier)

# Train model
ovr_model = ovr.fit(train)

# Make predictions
predictions = ovr_model.transform(test)

# Evaluate accuracy
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print(f"Test Accuracy = {accuracy:.4f}")
```

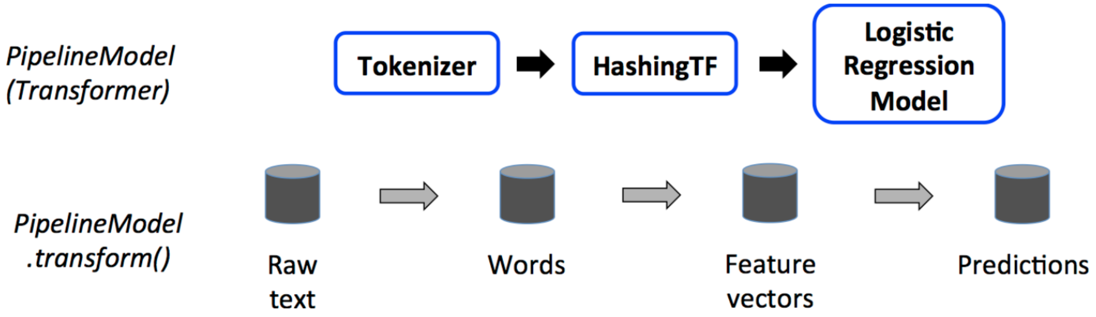
https://spark.apache.org/docs/latest/ml-classification-regression.html#one-vs-rest-classifier-aka-one-vs-all

Pipelines

- Sequence of stages
- Training (cf. `Estimator`)



- Testing (cf. `Transformer`)



Pipeline Example

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

# Configure ML pipeline components
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashing_tf = HashingTF(numFeatures=1000, inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)

# Build pipeline
pipeline = Pipeline(stages=[tokenizer, hashing_tf, lr])

# Fit pipeline to training data
model = pipeline.fit(training)

# Save fitted pipeline model
model.write().overwrite().save("/tmp/spark-logistic-regression-model")

# Apply model to test data
results = model.transform(test) \
    .select("id", "text", "probability", "prediction")

# Collect and display results
for row in results.collect():
    print(f"({row.id}, {row.text}) --> prob={row.probability}, prediction={row.prediction}")
```

https://spark.apache.org/docs/latest/ml-pipeline.html#example-pipeline

Model Selection & Hyperparameter Tuning Example

```
from pyspark.ml import Pipeline
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Create pipeline (assuming existing tokenizer, hashingTF, lr components)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Build parameter grid
param_grid = ParamGridBuilder() \
    .addGrid(hashingTF.numFeatures, [10, 100, 1000]) \
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .build()

# Configure cross-validator
cv = CrossValidator(
    estimator=pipeline,
    estimatorParamMaps=param_grid,
    evaluator=BinaryClassificationEvaluator(),
    numFolds=5,
    parallelism=2  # Process 2 parameter combinations in parallel
)

# Train cross-validated model
cv_model = cv.fit(training)
```

https://spark.apache.org/docs/latest/ml-tuning.html

Options to run Spark

- Jupyter notebooks on the cluster (**Recommended**)
  - Login with provided cluster account (with TU Wien VPN connection)
- Using interactive Spark shells on the cluster
  - Command *pyspark*
  - https://spark.apache.org/docs/latest/quick-start.html
- Submitting Spark jobs to Yarn in cluster mode
  - Steps as in Exercise_0 to access the cluster by ssh
  - Command spark-submit with python script as argument
  - https://spark.apache.org/docs/latest/running-on-yarn.html
- Locally
  - https://endjin.com/blog/2025/01/spark-devcontainers-local-spark

Using a simple Python script to approximate π

```python
import pyspark
import random

def inside(p):
 x, y = random.random(), random.random()
 return x*x + y*y < 1

sc = pyspark.SparkContext(appName="Pi")

num_samples = 10000


count = sc.parallelize(range(0, num_samples)).filter(inside).count()

pi = 4 * count / num_samples

print(pi)

sc.stop()
```

Assignment 2

Assignment 2: Instructions

Available here:

https://tuwel.tuwien.ac.at/pluginfile.php/4423610/mod_resource/content/1/Assignment_1_Instructions.pdf

Let's have a look!