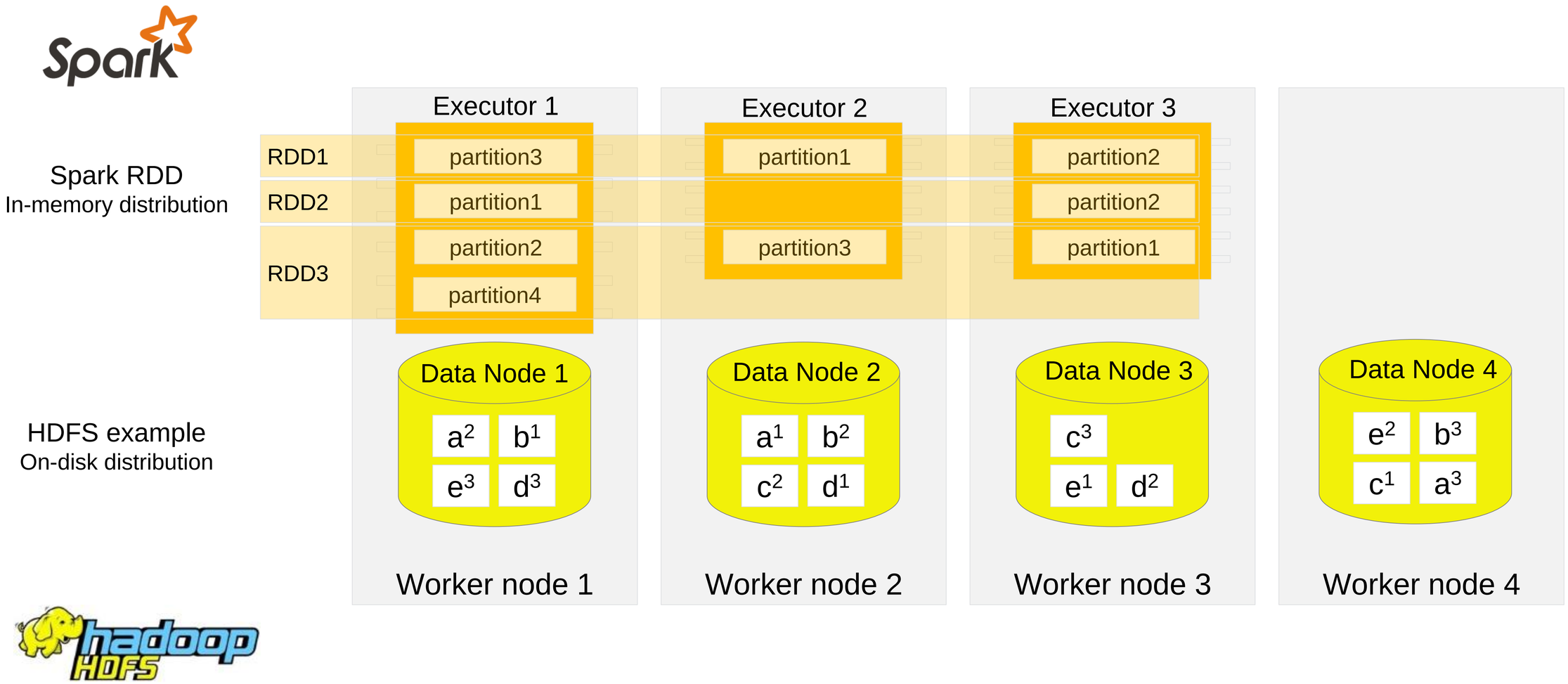
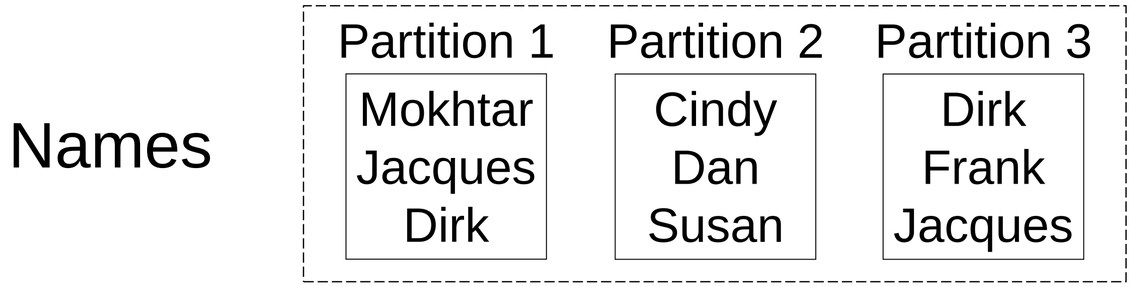
Spark as a distributed computation framework



Resilient Distributed Dataset (RDD): definition

- **An RDD is a distributed collection of Scala/Python/Java objects of the same type**: RDD of strings, integers, (key, value) pairs, Scala/Python/Java class.

- **An RDD is physically distributed across the cluster, but manipulated as one logical entity**: Spark will “distribute” any required processing to all partitions where the RDD exists and perform necessary redistributions and aggregations as well.

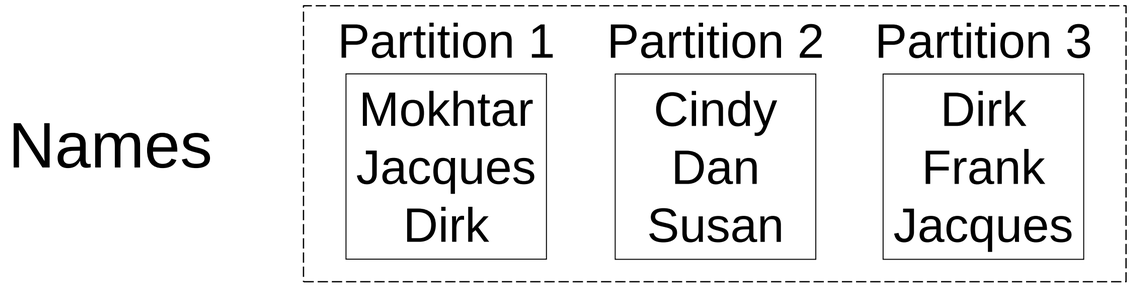


Resilient Distributed Dataset (RDD): basics

- **Suppose we want to know the number of names in the RDD “names”**
- **User simply requests:** names.count
- Spark will automatically distribute count processing to all partitions
- Local counts are subsequently aggregated

- **To lookup the first element in the RDD**: names.first

- **To display all elements of the RDD:** names.collect



Resilient Distributed Dataset (RDD): creation and manipulation

- **Several methods for creation:**
- Distributing a collection of objects from the driver program (using the *parallelize* method of the Spark context):

```
rddNumbers = sc.parallelize (range(1, 11))
rddLetters = sc.parallelize (["a", "b", "c", "d"])
```

- Loading an external dataset (file):

```
quotes = sc.textFile("hdfs:/sparkdata/sparkQuotes.txt")
```

- Transformation from another existing RDD:

```
rddNumbers2 = rddNumbers.map(lambda x: x+1)
```

- **Dataset from any Hadoop supported storage:**
  HDFS, Cassandra, Amazon S3, HBase, etc.

- **File types supported:** TextFiles, SequenceFiles, Parquet, JSON, etc.

Resilient Distributed Dataset (RDD): properties

- **Immutable**
- **Two types of operations:**
- Transformations:

```
rddNumbers = sc.parallelize (range(1, 11))
rddNumbers2 = rddNumbers.map (lambda x: x+1)
```

The lineage on how to obtain rddNumbers2 from rddNumbers is saved as a DAG (Directed Acyclic Graph). No data processing takes place: **lazy evaluation**.

- Actions:

```
rddNumbers2.collect()
```

Perform transformations and action, returns a value (or write to a file).

- **Fault tolerance:**

If data in memory is lost, it will be recreated from lineage

- **Caching, persistence (memory, disks)**

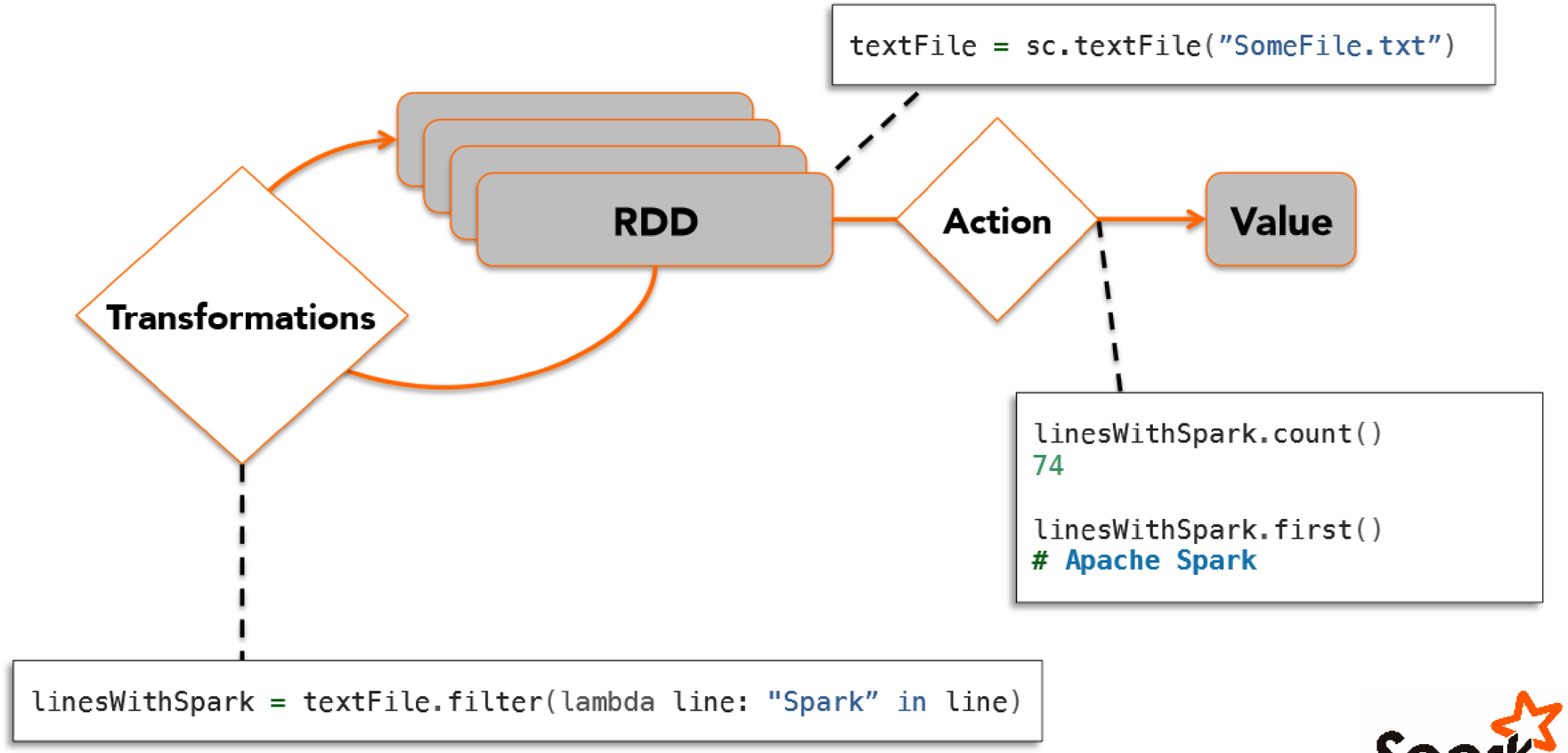
Resilient Distributed Dataset

Create RDDs:

- parallelize
- textFile
- Transformations

Get results:

- Actions



RDD Transformations

- Transformations are lazy evaluations
- Transformations return a pointer to the transformed RDD
- DAG (Directed Acyclic Graph) is generated by Spark

Transformation	Meaning
map(**func**)	Return a new dataset formed by passing each element of the source through a function **func**.
filter(**func**)	Return a new dataset formed by selecting those elements of the source on which **func** returns true.
flatMap(**func**)	Similar to map, but each input item can be mapped to 0 or more output items. So **func** should return a Seq rather than a single item.
join(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
reduceByKey(**func**)	When called on a dataset of (K, V) pairs, returns a dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function **func**.
sortByKey([ascending], [numTasks])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K,V) pairs sorted by keys in ascending or descending order.

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package

RDD Actions

- Actions return values or save RDD to disks

Action	Meaning
collect()	Return all the elements of the dataset as an array of the driver program. This is usually useful after a filter or another operation that returns a sufficiently small subset of data.
count()	Return the number of elements in a dataset.
first()	Return the first element of the dataset.
take(**n**)	Return an array with the first **n** elements of the dataset.
foreach(**func**)	Run a function **func** on each element of the dataset. Does not return any RDD.
saveAsTextFile(**path**)	Save the RDD into a TextFile at the given **path**.

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package

RDD Persistence

- Each node stores any partitions of the cache that it computes in memory
- Reuses them in other actions on that dataset (or datasets derived from it)

Storage Level	Meaning
MEMORY_ONLY	Store as deserialized Java objects in the JVM. If the RDD does not fit in memory, part of it will be cached. The other will be recomputed as needed. This is the default. The cache() method uses this.
MEMORY_AND_DISK	Same except also store on disk if it doesn't fit in memory. Read from memory and disk when needed.
MEMORY_ONLY_SER	Store as serialized Java objects (one byte array per partition). Space efficient, but more CPU intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_AND_DISK but stored as serialized objects.
DISK_ONLY	Store only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Store RDD in serialized format in Tachyon.

Spark Word Count

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.net.URI;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.Counter;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.StringUtils;

public class WordCount2 {

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

    static enum CountersEnum { INPUT_WORDS }

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

    private boolean caseSensitive;
    private Set<String> patternsToSkip = new HashSet<String>();

    private Configuration conf;
    private BufferedReader fis;

    @Override
    public void setup(Context context) throws IOException,
        InterruptedException {
      conf = context.getConfiguration();
      caseSensitive = conf.getBoolean("wordcount.case.sensitive", true);
      if (conf.getBoolean("wordcount.skip.patterns", true)) {
        URI[] patternsURIs = Job.getInstance(conf).getCacheFiles();
        for (URI patternsURI : patternsURIs) {
          Path patternsPath = new Path(patternsURI.getPath());
          String patternsFileName = patternsPath.getName().toString();
          parseSkipFile(patternsFileName);
        }
      }
    }

    private void parseSkipFile(String fileName) {
      try {
        fis = new BufferedReader(new FileReader(fileName));
        String pattern = null;
        while ((pattern = fis.readLine()) != null) {
          patternsToSkip.add(pattern);
        }
      } catch (IOException ioe) {
        System.err.println("Caught exception while parsing the cached file '"
            + StringUtils.stringifyException(ioe));
      }
    }

    @Override
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      String line = (caseSensitive) ?
          value.toString() : value.toString().toLowerCase();
      for (String pattern : patternsToSkip) {
        line = line.replaceAll(pattern, "");
      }
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
        Counter counter = context.getCounter(CountersEnum.class.getName(),
            CountersEnum.INPUT_WORDS.toString());
        counter.increment(1);
      }
    }
  }

  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();
    GenericOptionsParser optionParser = new GenericOptionsParser(conf, args);
    String[] remainingArgs = optionParser.getRemainingArgs();
    if (!(remainingArgs.length != 2 || remainingArgs.length != 4)) {
      System.err.println("Usage: wordcount <in> <out> [-skip skipPatternFile]");
      System.exit(2);
    }
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount2.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    List<String> otherArgs = new ArrayList<String>();
    for (int i=0; i < remainingArgs.length; ++i) {
      if ("-skip".equals(remainingArgs[i])) {
        job.addCacheFile(new Path(remainingArgs[++i]).toUri());
        job.getConfiguration().setBoolean("wordcount.skip.patterns", true);
      } else {
        otherArgs.add(remainingArgs[i]);
      }
    }
    FileInputFormat.addInputPath(job, new Path(otherArgs.get(0)));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs.get(1)));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

```
text_file = sc.textFile("hdfs://...")

text_file   .flatMap(lambda line: line.split(" "))
            .map(lambda word: (word, 1))
            .reduceByKey(lambda (a, b): a+b)
            .saveAsTextFile("hdfs://...")
```

Code Execution (1/5)

- Spark Context is provided as **sc** by spark-shell, pyspark or Bluemix


```
// Create RDD
quotes = sc.textFile("hdfs://tmp/sparkQuotes.txt")
// Transformations
danQuotes = quotes     .filter(lambda line: line.startswith("DAN"))
                       .map(lambda line: line.split(" "))
sparkQuotes = danQuotes.map(lambda line: line[1])
                       .filter(lambda word: “Spark” in word)

// Actions
sparkQuotes.count()
```

FILE: sparkQuotes.txt

```
DAN Spark is cool
BOB Spark is fun
BRIAN Spark is great
DAN Scala is awesome
BOB Scala is flexible
```

Code Execution (2/5)

```
// Create RDD
quotes = sc.textFile("hdfs://tmp/sparkQuotes.txt")
// Transformations
danQuotes = quotes     .filter(lambda line: line.startswith("DAN"))
                       .map(lambda line: line.split(" "))
sparkQuotes = danQuotes.map(lambda line: line[1])
                       .filter(lambda word: “Spark” in word)
// Actions
sparkQuotes.count()
```

FILE: sparkQuotes.txt

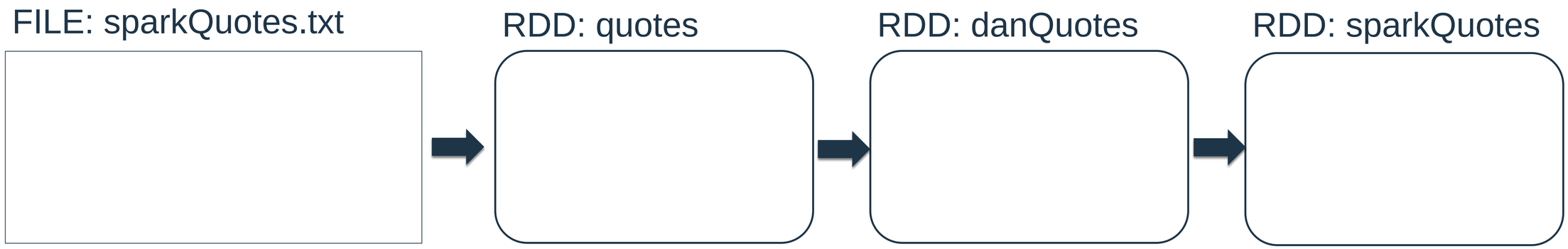
RDD: quotes

Code Execution (3/5)

```
// Create RDD
quotes = sc.textFile("hdfs://tmp/sparkQuotes.txt")
// Transformations
danQuotes = quotes     .filter(lambda line: line.startswith("DAN"))
                       .map(lambda line: line.split(" "))
sparkQuotes = danQuotes.map(lambda line: line[1])
                       .filter(lambda word: “Spark” in word)
// Actions
sparkQuotes.count()
```

Code Execution (4/5)

```
// Create RDD
quotes = sc.textFile("hdfs://tmp/sparkQuotes.txt")
// Transformations
danQuotes = quotes     .filter(lambda line: line.startswith("DAN"))
                       .map(lambda line: line.split(" "))
sparkQuotes = danQuotes.map(lambda line: line[1])
                       .filter(lambda word: “Spark” in word)
// Actions
sparkQuotes.count()
```



Code Execution (5/5)

```
// Create RDD
quotes = sc.textFile("hdfs://tmp/sparkQuotes.txt")
// Transformations
danQuotes = quotes     .filter(lambda line: line.startswith("DAN"))
                       .map(lambda line: line.split(" "))
sparkQuotes = danQuotes.map(lambda line: line[1])
                       .filter(lambda word: “Spark” in word)
// Actions
sparkQuotes.count()
```

FILE: sparkQuotes.txt

```
DAN Spark is cool
BOB Spark is fun
BRIAN Spark is great
DAN Scala is awesome
BOB Scala is flexible
```

RDD: quotes

```
DAN Spark is cool
BOB Spark is fun
BRIAN Spark is great
DAN Scala is awesome
BOB Scala is flexible
```

RDD: danQuotes

```
[“DAN”, “Spark”, “is”, “cool”]
[“DAN”, “Scala”, “is”, “awesome”]
```

RDD: sparkQuotes

```
Spark
```

1

Spark standalone application in Scala

```scala
/* SimpleApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your system
    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
  }
}
```

Import statements

Transformations and Actions

SparkConf and SparkContext

Spark Properties

- Set application properties via the SparkConf object:

```
val conf = new SparkConf()
                .setMaster("local")
                .setAppName("CountingSheep")
                .set("spark.executor.memory", "1g")
val sc = new SparkContext(conf)
```

- Dynamically setting Spark properties:
  SparkContext with an empty conf: val sc = new SparkContext(new SparkConf())
  Supply the configuration values during runtime:

```
./bin/spark-submit --name "My app" --master local[4] --conf spark.shuffle.spill=false myApp.jar conf/spark-defaults.conf
```

- Application web UI: http://<driver>:4040
- Spark History web UI: http://<driver>:18080

Executing a Spark job

- pyspark: Python REPL
- spark-shell: Scala REPL
- sparkR: R REPL
- spark-submit: execute a Scala/Java compiled .jar file, a Python .py script or a R .r script
- Example: ./bin/spark-submit examples/src/main/python/pi.py 10

- Deploy mode:
  - Standalone Mode
  - Apache Hadoop YARN
  - Apache Mesos

Spark Final Thoughts

- Spark is a good replacement for MapReduce:
  - Higher performance
  - Easier to use framework
  - Powerful RDD & DataFrames concepts
  - Big higher level libraries: SparkSQL, MLlib, Streaming, GraphX
  - Huge ecosystem adoption
  - Great community and enterprise support

- This is a very fast paced environment, so keep up!
  - Lot of new features at each new release (major release every 3 months)
  - Spark has the latest / best offer but things may change again

Spark API documentations

- Spark Scala API (Scaladoc)
- Spark Java API (Javadoc)
- Spark Python API (Sphinx)
- Spark R API (Roxygen2)

Spark Core – Practice

MovieLens

- http://grouplens.org/datasets/movielens/latest/
- MovieLens 20M dataset
- 138k users (Netflix announced 69M users)
- 27k movies (Netflix probably cast between 10k and 100k different movies)
- 20M ratings (Sparse Matrix)

Notebook

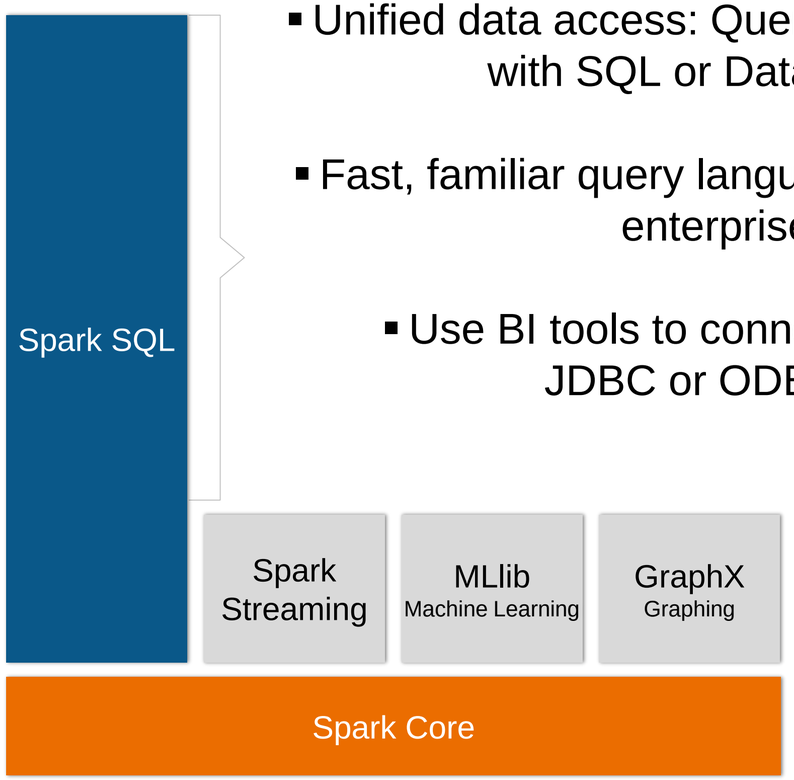
- Download the movielens-exercises.ipynb notebook
- Stop at step **3. Collaborative filtering**
- Solve the additional exercises from exercises.pdf

Exercises

- Use the timestamp column from the ratings
  - Convert it as a human-readable time/date
  - Check the evolution of ratings versus time?
- Is that something related to time or users with the .5 notation?
- Explode the genres column and compute statistics such as:
  - Distribution
  - Average rating by genre

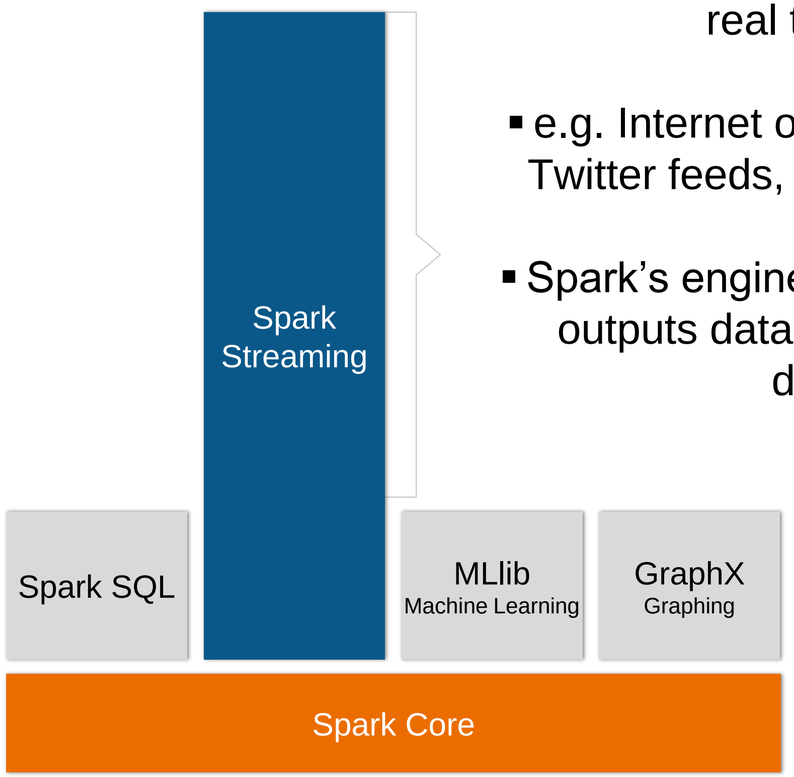
Introduction to Spark Libraries

Select Libraries to Meet Use-Case Challenges



- Unified data access: Query structured data sets with SQL or DataFrame APIs
- Fast, familiar query language across all of your enterprise data
- Use BI tools to connect and query via JDBC or ODBC drivers

Select Libraries to Meet Use-Case Challenges



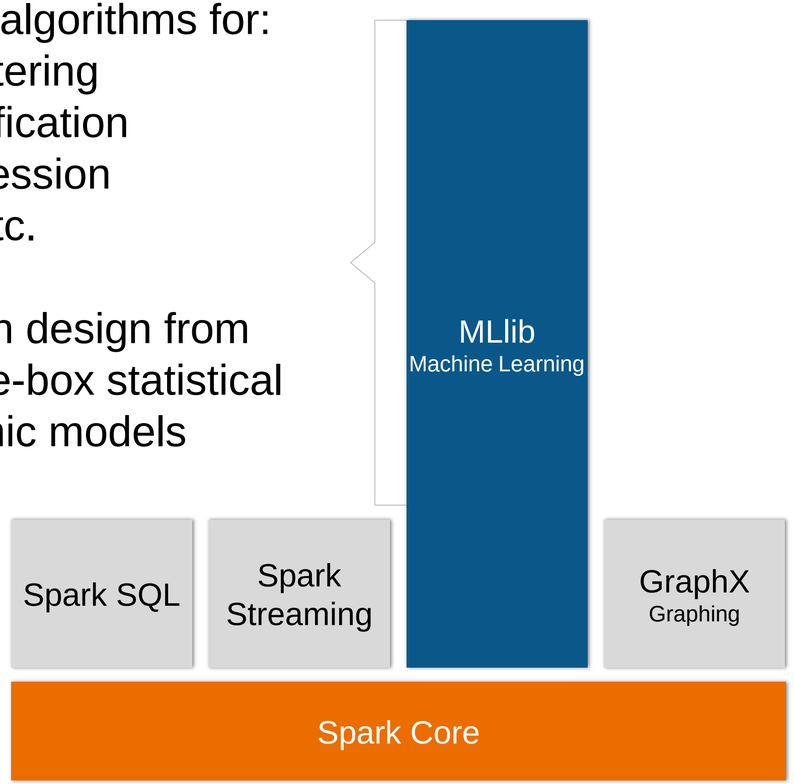
- Micro-batch event processing for near-real time analytics
- e.g. Internet of Things (IoT) devices, Twitter feeds, Kafka (event hub), etc.
- Spark's engine drives some action or outputs data in batches to various data stores

Select Libraries to Meet Use-Case Challenges

- Predictive and prescriptive analytics

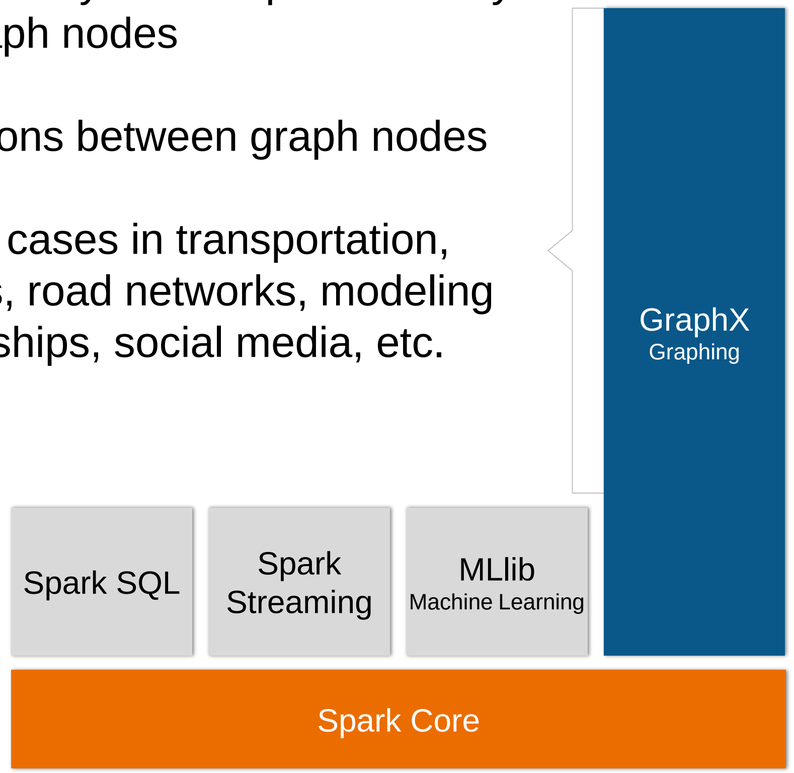
- Machine learning algorithms for:
  - Clustering
  - Classification
  - Regression
  - etc.

- Smart application design from pre-built, out-of-the-box statistical and algorithmic models



Select Libraries to Meet Use-Case Challenges

- Represent and analyze systems represented by graph nodes
- Trace interconnections between graph nodes
- Applicable to use cases in transportation, telecommunications, road networks, modeling personal relationships, social media, etc.



Spark SQL & Data Frames

What are structured / semi-structured / not structured data?

- Structured has a schema with type that is usually control on read/write
- Semi structured can have an easy schema on read JSON/XML or an implicit structure with no type
- Not structure is plain text or image

- Less structured data over time with IoT, Big Data, etc.

What is a schema?

- Schema can be inferred automatically by Spark
- Or defined by the programmer

What is SparkSQL?

- Spark SQL is Apache Spark's module for working with structured data
  - Define a DataFrame object for structured data
  - Let users seamlessly mix SQL queries with Spark programs
  - Use either SQL or a familiar DataFrame API
- Connect to multiple data sources and data formats from an unified API
  - Sources include Swift, S3, HDFS, Hive or any JDBC database
  - Formats include ORC, JSON, CSV or Parquet
- Provide a Hive compatibility
  - Reuse Hive data, queries, UDFs or SerDes
- Provide a standard connectivity
  - Connect third-party tools to Spark SQL through ODBC or JDBC

```
context = HiveContext(sc)
results = context.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Apply functions to results of SQL queries.

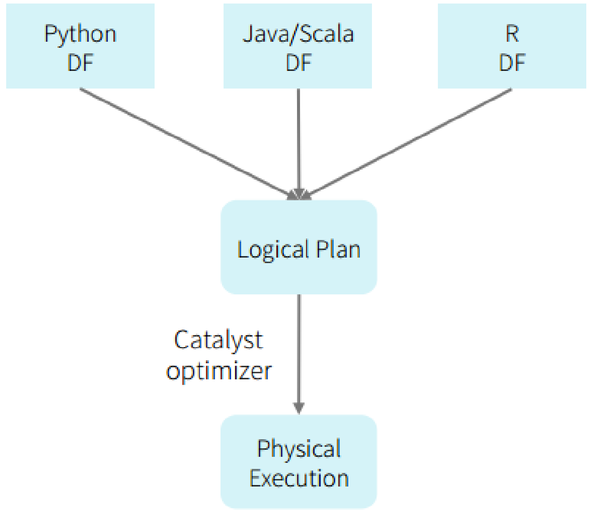
Spark DataFrames, released Spark 1.3

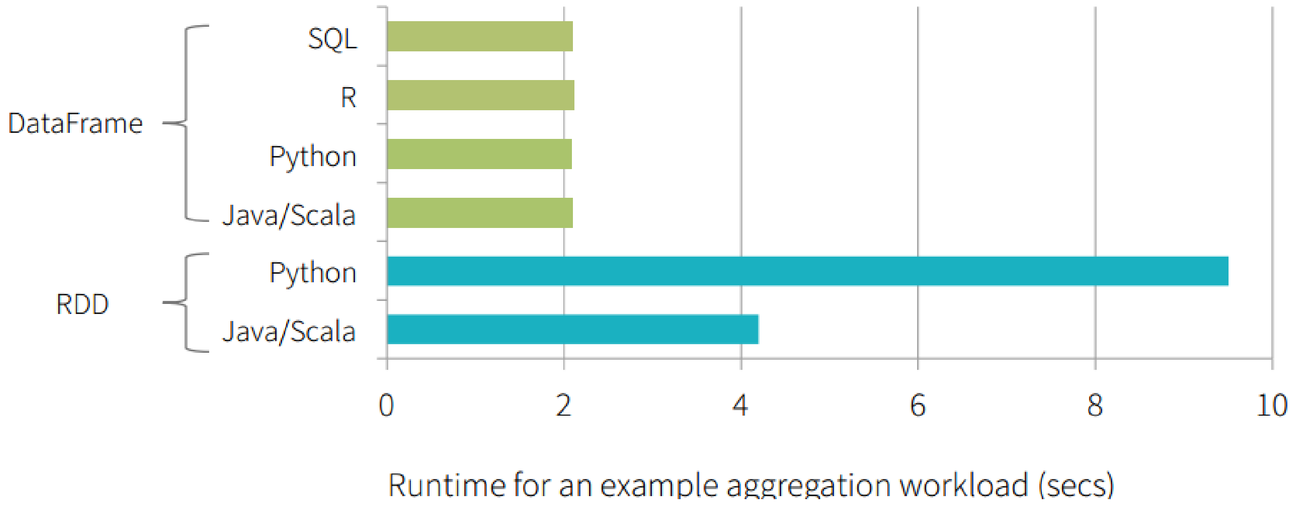
- A DataFrame is a distributed collection of data organized into named columns.

It is conceptually equivalent to a table in a relational database, an R dataframe or Python Pandas, but in a distributed manner and with query optimizations and predicate pushdown to the underlying storage.

- DataFrames can be constructed from a wide array of sources such as:

structured data files, tables in Hive, external databases, or existing RDDs.

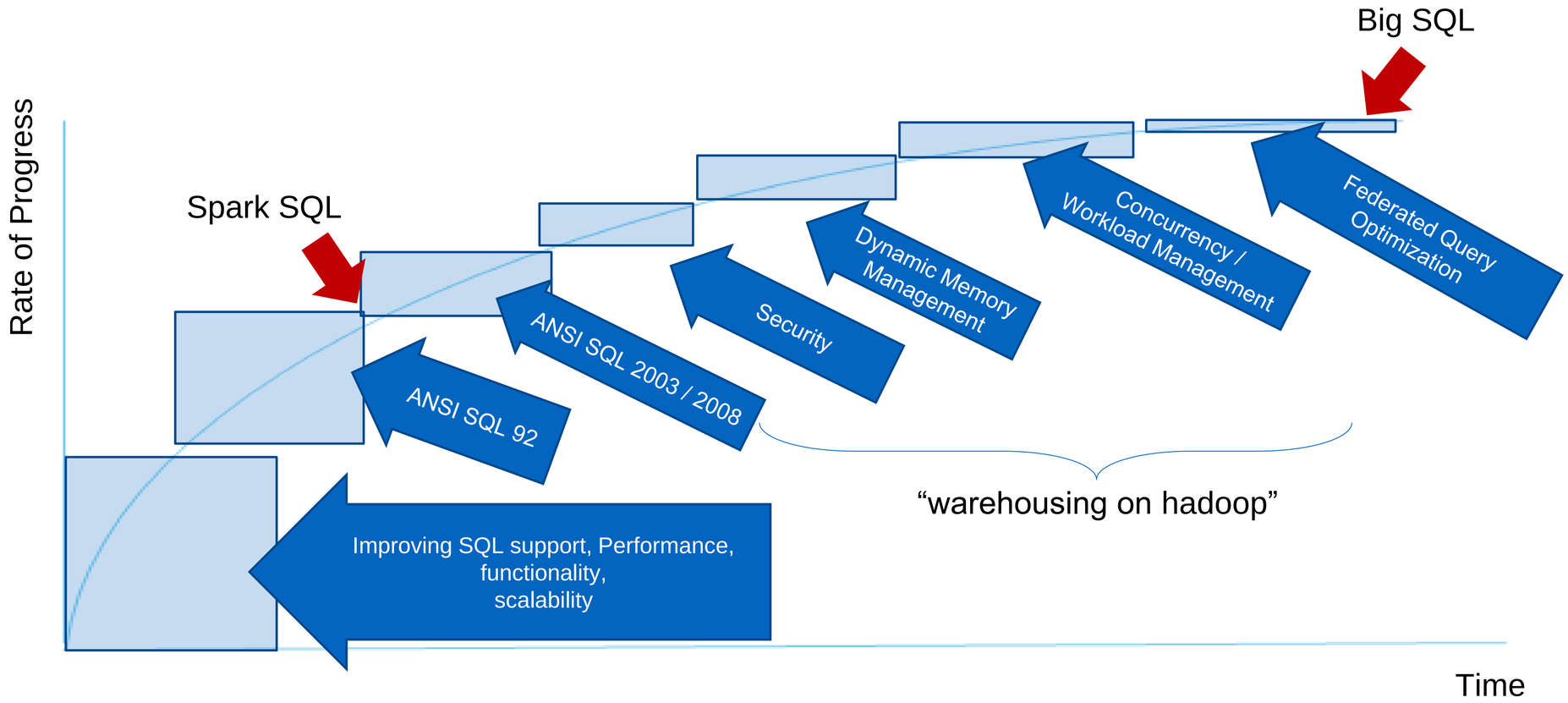




Spark DataFrames – Examples

```
// Create the DataFrame
val df = sqlContext.read.parquet("examples/src/main/resources/people.parquet")

// Show the content of the DataFrame
df.show()

// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()

// Select everybody, but increment the age by 1
df.select(df("name"), df("age") + 1).show()

// Select people older than 21
df.filter(df("age") > 21).show()

// Count people by age
df.groupBy("age").count().show()
```

SparkSQL is not a completely mature SQL engine on Hadoop yet



Exercises

- Perform the previous analysis with DataFrames & SQL
  - Define the schema
  - Infer the schema
- Performances:
  - Can you compare the performances with RDD?
  - Can you use the Parquet file format or compression to optimize performances?

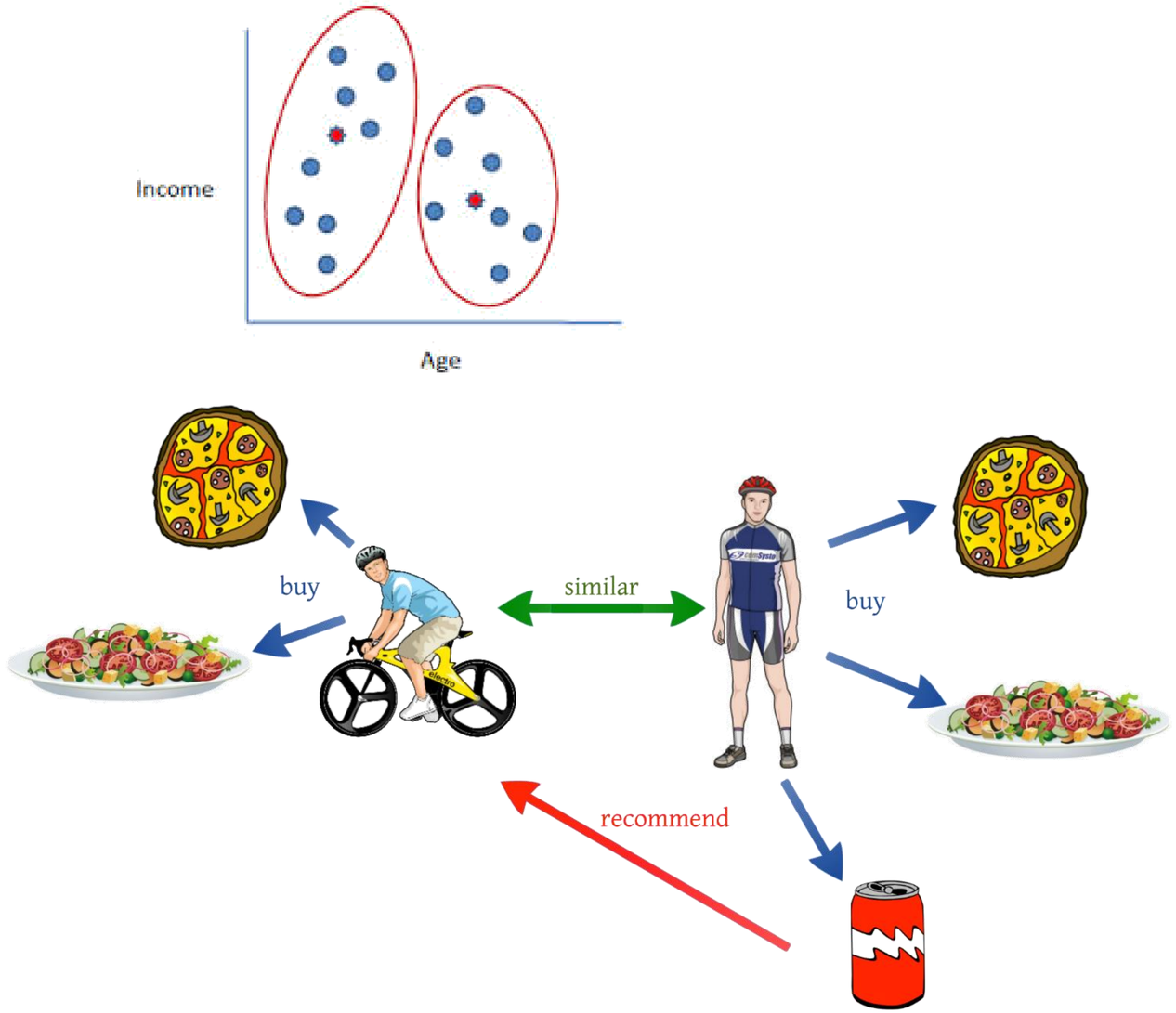
Spark MLlib & Spark ML

What is Machine Learning?

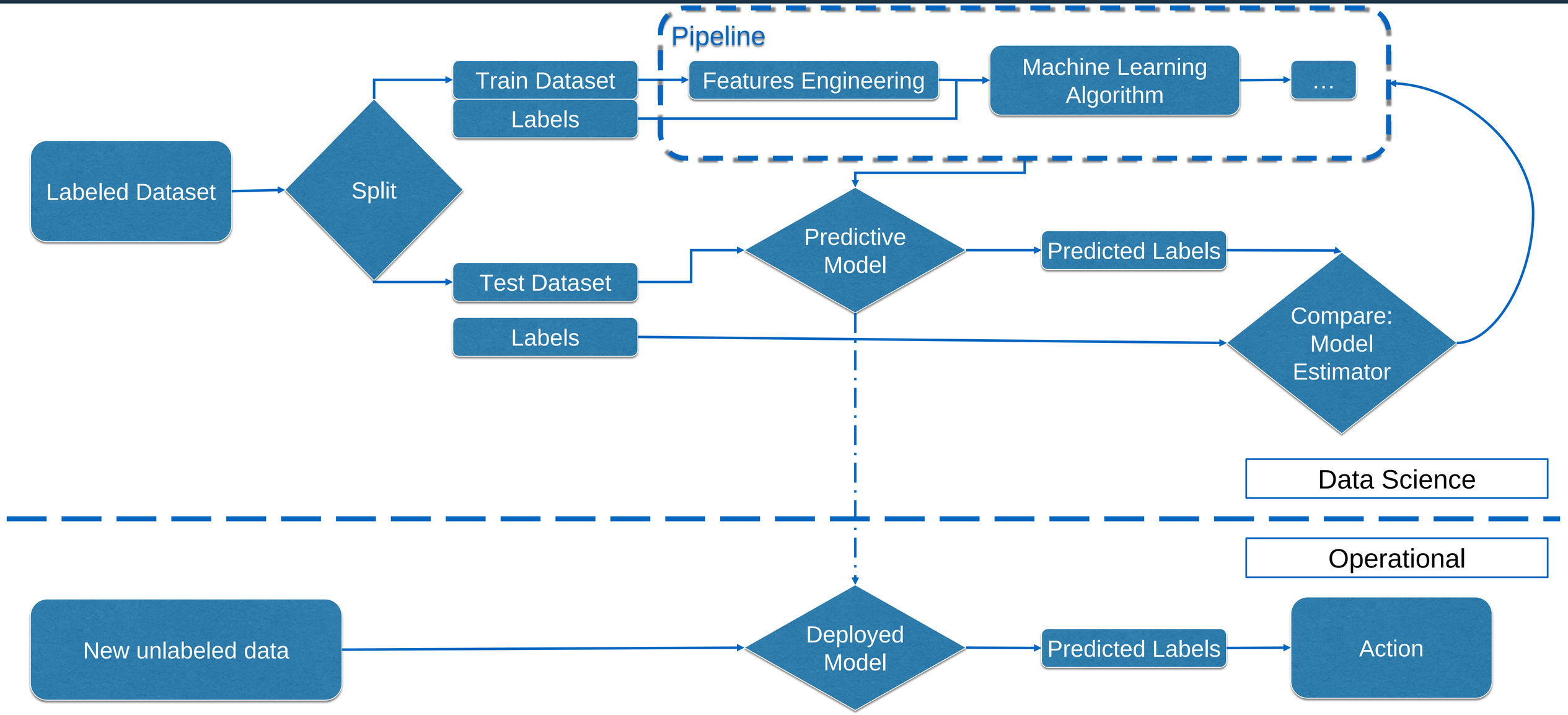
- Identify patterns to make decisions
- Use cases: prediction, classification, clustering, recommender systems, customer segmentation, fraud detection, sentiment analysis, customer churn, people relationships, mortgage loan, etc.
- Tools: scikit-learn, R, pandas, Spark

What is Machine Learning?

- Unsupervised
  - Dimensionality reduction
  - Clustering
  - Association
- Supervised
  - Regression
  - Decision Trees
  - Classification
  - Recommender systems



Supervised Machine Learning Standard Workflow



What is MLlib?

- MLlib is Apache Spark’s scalable machine learning library
- MLlib fits into Spark API and integrates with R and Python NumPy
- MLlib delivers high performance Machine Learning and especially excels at iterative computation with in-memory computation
- MLlib already implements tens of algorithms and growing fast
- MLlib algorithms are parallelized and scalable

SparkML vs MLlib

- **MLlib** is the RDD-API for Spark Machine Learning

- **SparkML** is the DataFrame-API for Spark Machine Learning
  - DataFrame API enables user-friendly APIs, SQL, Tungsten & Catalyst optimizations
  - SparkML provides a uniform / standardized API over languages and algorithms
  - DataFrames facilitate practical ML thanks to the pipeline feature

- Parity between SparkML & MLlib is expected around Spark 2.2

- For now, not all MLlib algorithms are implemented in SparkML

SparkML Pipelines

- A pipeline defines a Machine Learning Workflow
- Combination of:
  - **DataFrame**: the ML dataset to store text, features, true labels and predictions
  - **Transformer**: an algorithm to transform an input DataFrame into an other DataFrame, for instance with predictions
  - **Estimator**: an algorithm which can be fit on a DataFrame to produce a Transformer, for instance a learning algorithm that can be fitted on a dataset and produce a model (which is a Transformer)
  - **Pipeline**: chains multiple Transformers and Estimators together to build an ML workflow
  - **Parameter**: an API for specifying parameters to Transformers and Estimators

SparkML & DataFrames

- SparkML can benefit from all the data types supported by DataFrames
- SparkML also support a Vector type
- DataFrame can be created implicitly from a structured or semi-structured input file or explicitly from a RDD and a given schema
- DataFrame columns are named: text, features, label, prediction (for instance)

Transformers

- Transformers can be:
  - Feature transformers
    - e.g. a transformer that read a column (e.g. text) and map it into a new column (e.g. feature vectors), returning the complete DataFrame
  - Learning models
    - e.g. a learning model that read the feature vectors, predict the label for each feature vector and output the complete DataFrame with predicted labels

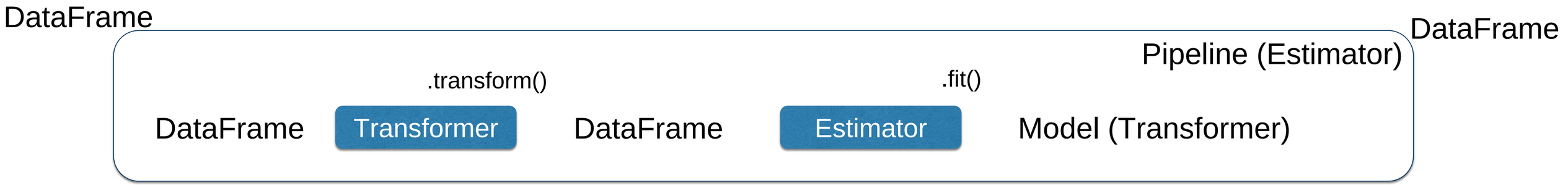
- Both transform a DataFrame into an other DataFrame, appending one or more columns

Estimators

- Estimators abstract the concept of a learning algorithm that fits or trains on a dataset
- Estimators implement the method `.fit()` which accept a DataFrame and produce a Model, which is a Transformer
  - e.g. **LogisticRegression** is an **Estimator**, **LogisticRegression.fit()** trains a **LogisticRegressionModel** which is a **Model** and hence a **Transformer**

Pipeline

- A Machine Learning workflow usually follows a pipeline of transformations and learning algorithms going through features engineering to model training
- In SparkML, this is represented as a Pipeline which is a sequence of PipelineStages (Transformers and Estimators) that are run in order
- For every Transformer, the .transform() method is called outputting a new DataFrame
- For every Estimator, the fit() method is called, outputting a new Model (Transformer)

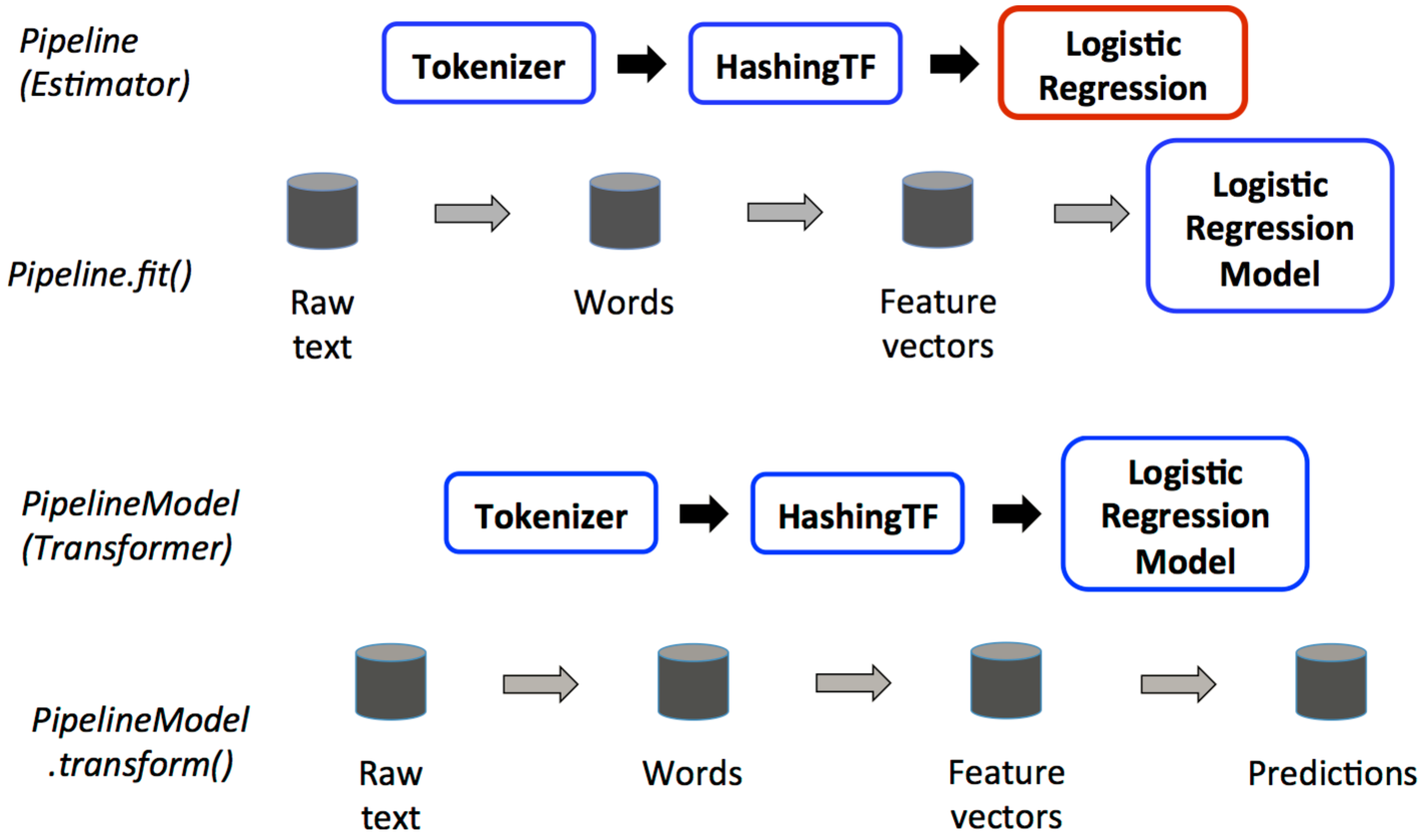


Pipeline Usage

- The built Pipeline is an Estimator and thus, can be fitted on a DataFrame (test data), resulting in a Model that can be then applied as a Transformer to a DataFrame (to-predict data)
- Pipeline.fit() produces a PipelineModel, which is a Transformer and can be used at test time
- The PipelineModel has the same numbers of steps as the Pipeline but every Estimator from the Pipeline is replaced in the PipelineModel by a Transformer (see following example)
- This unified Pipeline process helps ensure that training and test data go through the exact same steps

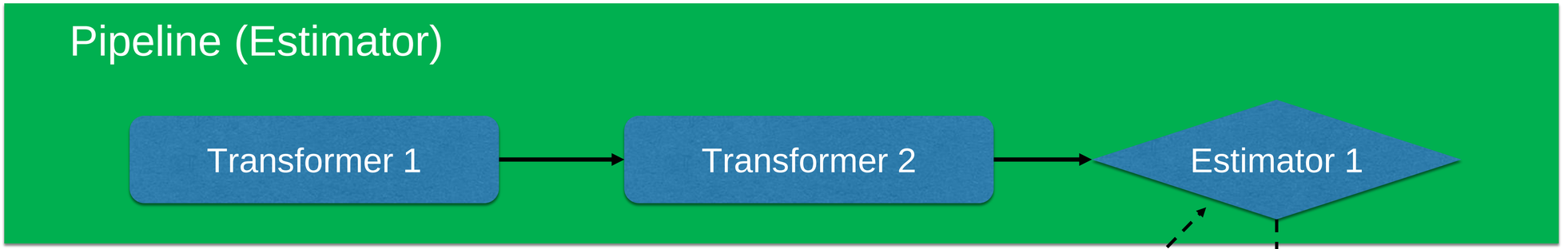
Pipeline Example

- Document classification
  - Transformers (in blue)
  - Estimator (in red)



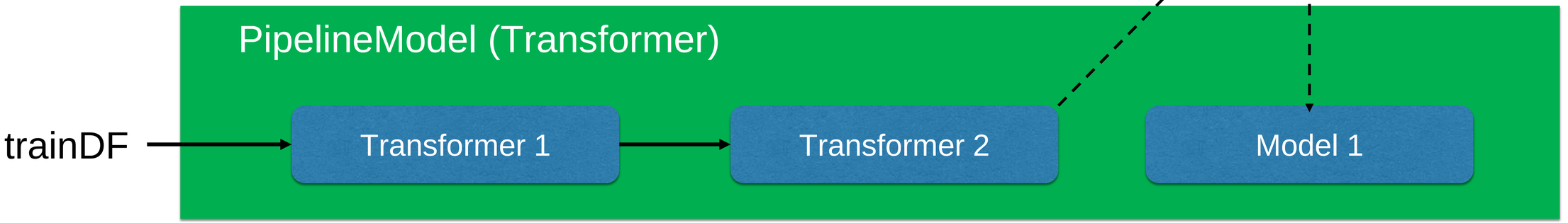
Pipeline Workflow

p = [Transformer 1, Transformer 2, Estimator 1]

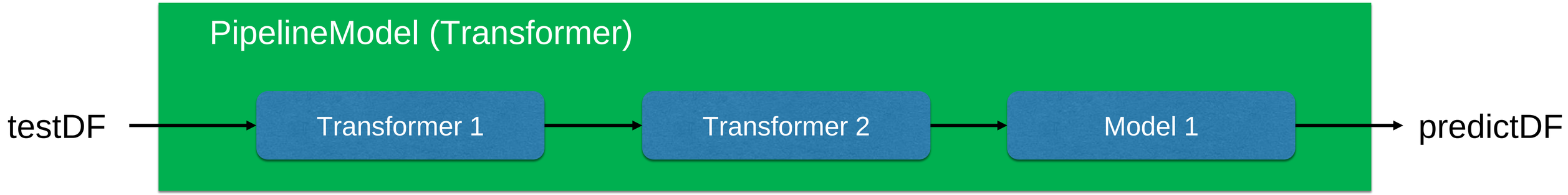


pm = p.fit(trainDataFrame)

.fit()



predictDataFrame = pm.transform(testDataFrame)



Model Selection: Hyperparameter Optimization

- Hyperparameter Optimization or Tuning is the process of chosing the **Estimator** or **Pipeline** parameters that produce the best model / results
- To tune your **Estimator** or **Pipeline**, you need:
  - a parameter grid: a set or **ParamMaps** to search over
  - a metric to measure how well a fitted Model is performing: an Evaluator
- Model Selection Tools such as **CrossValidator** and **TrainValidationSplit** use the following process to measure **Model** performance:
  - Split the input data between training and test datasets
  - For each (training, test) pair and for each **ParamMap**, fit the **Estimator** or **Pipeline** and evaluate the fitted **Model** using the specified **Evaluator**
  - Select the **Model** that show the best results
- **ParamGridBuilder** helps create a set of **ParamMaps**

CrossValidator example

- You have a train dataset of 10 elements
- You have a pipeline with an HashingTF and a LogisticRegression
- You want to test 2 values for lr.regParam and 3 values for hashingTF.numFeatures
- You want to choose the best model with CrossValidator on 2 folds
- There are (3x2)x2=12 models to consider

Spark MLlib

- Stands for Machine Learning library
- First release in Spark 0.8
- Provides common algorithms and utilities:
  - Classification
  - Regression
  - Clustering
  - Collaborative Filtering
  - Dimensionality Reduction
- Leverages in-memory cache of Spark to speed-up iteration processing

- Classification and regression
  - linear models (SVMs, logistic regression, linear regression)
  - naive Bayes
  - decision trees
  - ensembles of trees (Random Forests and Gradient-Boosted Trees)
  - isotonic regression
- Data types
- Basic statistics
  - summary statistics
  - correlations
  - stratified sampling
  - hypothesis testing
  - streaming significance testing
  - random data generation
- Collaborative filtering
  - alternating least squares (ALS)
- Clustering
  - k-means
  - Gaussian mixture
  - power iteration clustering (PIC)
  - latent Dirichlet allocation (LDA)
  - bisecting k-means
  - streaming k-means
- Dimensionality reduction
  - singular value decomposition (SVD)
  - principal component analysis (PCA)
- Feature extraction and transformation
- Frequent pattern mining
  - FP-growth
  - association rules
  - PrefixSpan
- Evaluation metrics
- PMML model export
- Optimization (developer)
  - stochastic gradient descent
  - limited-memory BFGS (L-BFGS)

Spark Streaming

What is Spark Streaming?

- Spark Streaming makes it easy to build scalable fault-tolerant streaming applications
- Spark Streaming brings Apache Spark's API to stream processing letting you write streaming jobs the same way you write batch jobs
- Spark Streaming provide fault-tolerance with stateful exactly-once semantics out of the box
- Spark Streaming is based on Spark and let user combine streaming with batch to join streams against historical data or build a so called lambda architecture

Spark Streaming

- Scalable, high-throughput, fault-tolerant stream processing of live data streams
- Write Spark streaming applications like Spark applications
- Recovers lost work and operator state (sliding windows) out-of-the-box
- Uses HDFS and Zookeeper for high availability
- Data sources also include TCP sockets, ZeroMQ or other customized data sources

Spark Streaming - Getting started

- Count the number of words coming in from the TCP socket
- Import the Spark Streaming classes:

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
```

- Create the StreamingContext object:

```
val conf = new SparkConf() .setMaster("local[2]")
                                        .setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

- Create a DStream:

```
val lines = ssc.socketTextStream("localhost", 9999)
```

- Split the lines into words:

```
val words = lines.flatMap(_.split(" "))
```

- Count the words:

```
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
```

- Print to the console:

```
wordCounts.print()
```

- No real processing happens until you tell it:

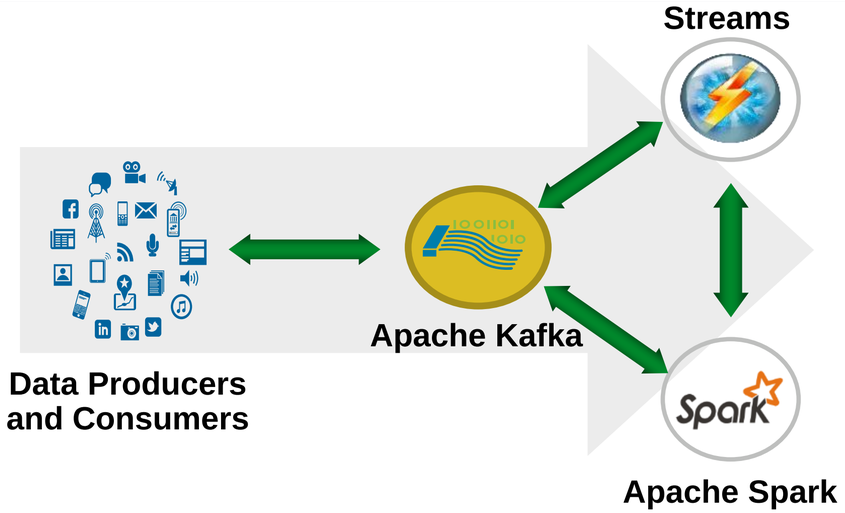
```
ssc.start()             // Start the computation
ssc.awaitTermination() // Wait for the computation to terminate
```

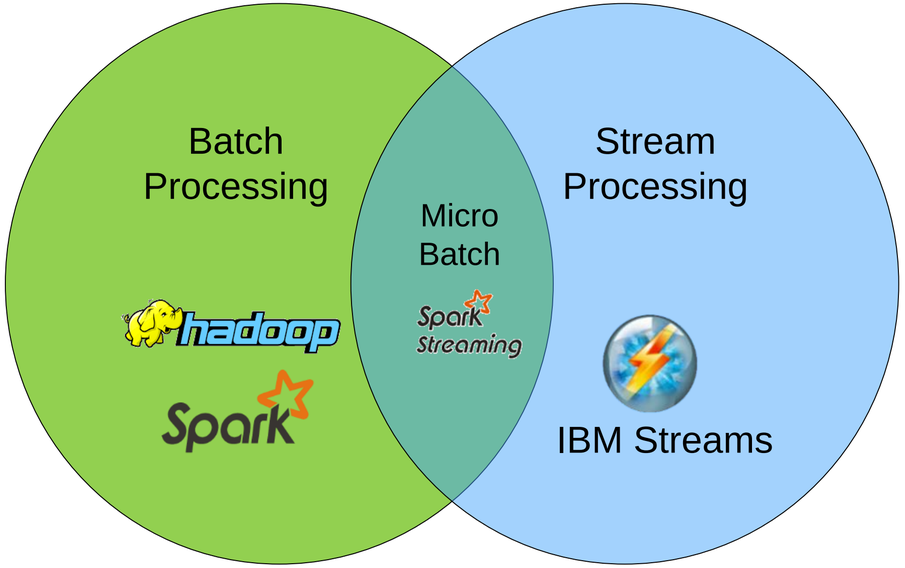
Spark and IBM Streams: Better Together

- **Spark for data-at-rest**
- A more expressive MapReduce
- Toward a better Hadoop ecosystem

- **Spark Streaming**
- Time-based batch processing
- Simplest approach for once only processing

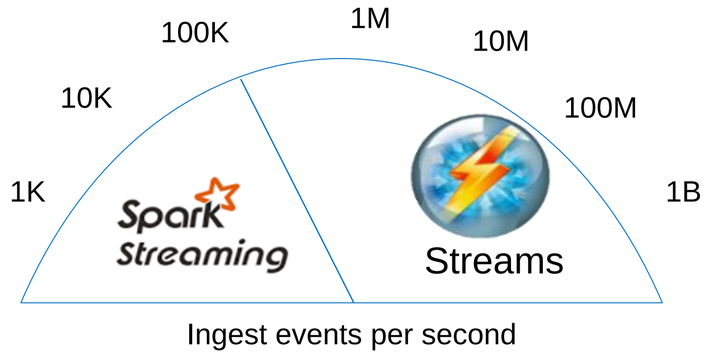
- **IBM Streams**
- Complements both capabilities
- Interoperate via Kafka & HDFS
- Enterprise integration
- Low latency
- Event processing
- Extended analytics



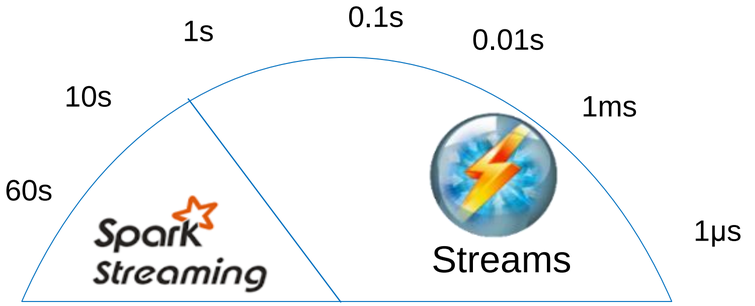


Streams event processing examples

Media Agency
- News Ingest and Analytics
  - News, Market Data, Meta Data to Streams to Hbase
- Signal App: Real time Technical Analysis
  - Bollinger Band, Simple moving average, etc.
- VolSurf: Real time volatility surfaces
  - 200k instruments, 100k mps,

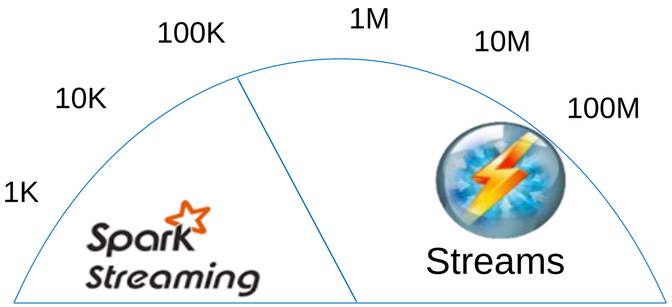


Ingest events per second



Latency required (seconds)

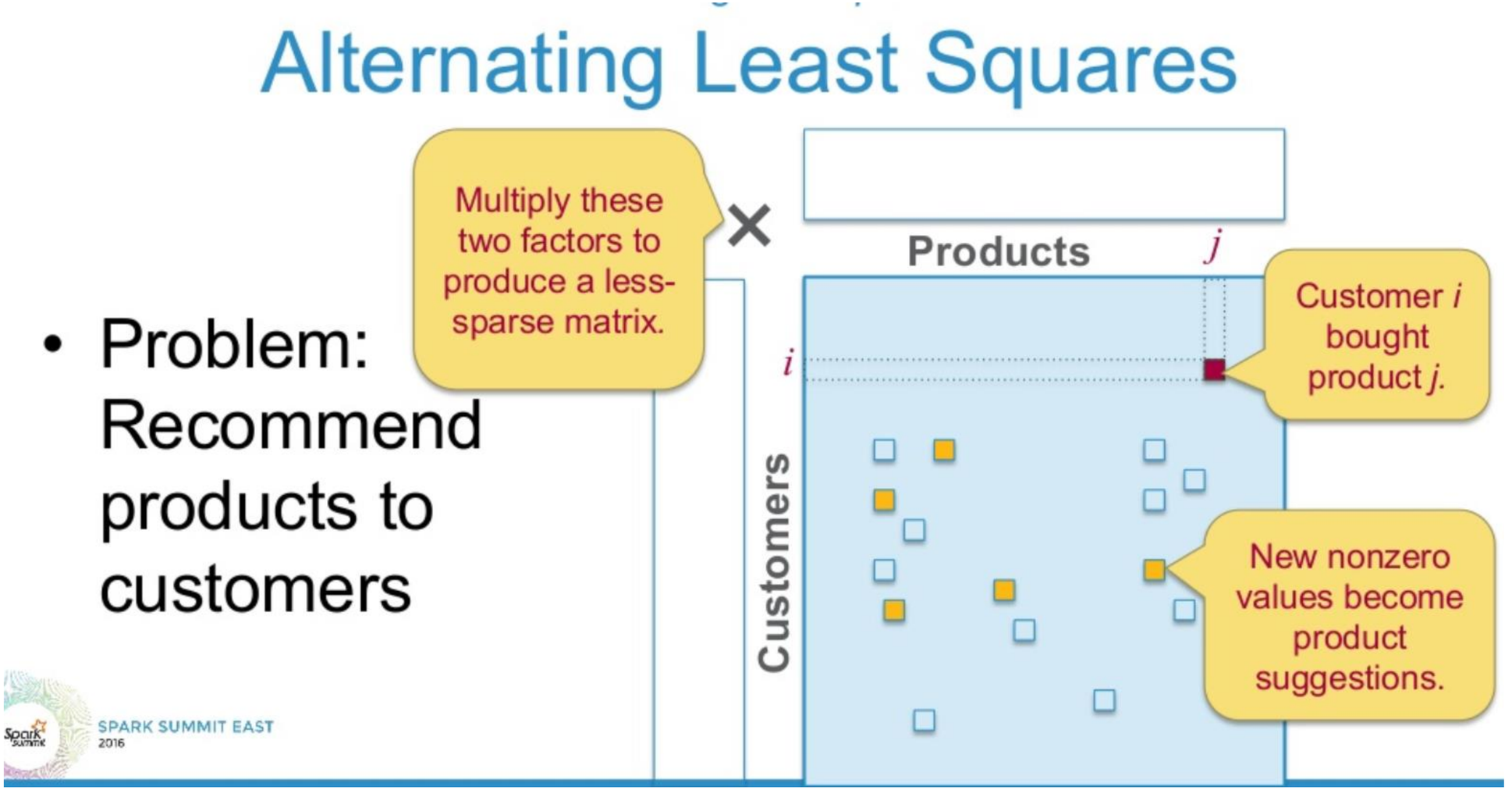
North American Telco real time advertising
- Click thru rate and Revenue up 50%
- ~30M in memory profiles, 500 SPSS models in real time
- Purchases, Web click stream, CDRs, IPTV viewing, Behavioral events
- Total events ~1.2B per day, 210K per second
- Average Latency 7ms



Models maintained

Data Science on Apache Spark

Alternating Least Squares



- Problem: Recommend products to customers

Questions

- What is the RMSE?

Data Science on Apache Spark – Exercises

- Pursue the movielens.ipynb notebook
- Automatize the cross-validation
- Draw a confusion matrix
- Build a category based recommendation
- Build a multi user recommendation application
- Use additional data to improve the model
- Use the DataFrames and the Pipelines