# Distributed Processing with Computer Clusters II

*Advanced Databases and Information Systems*

SS18

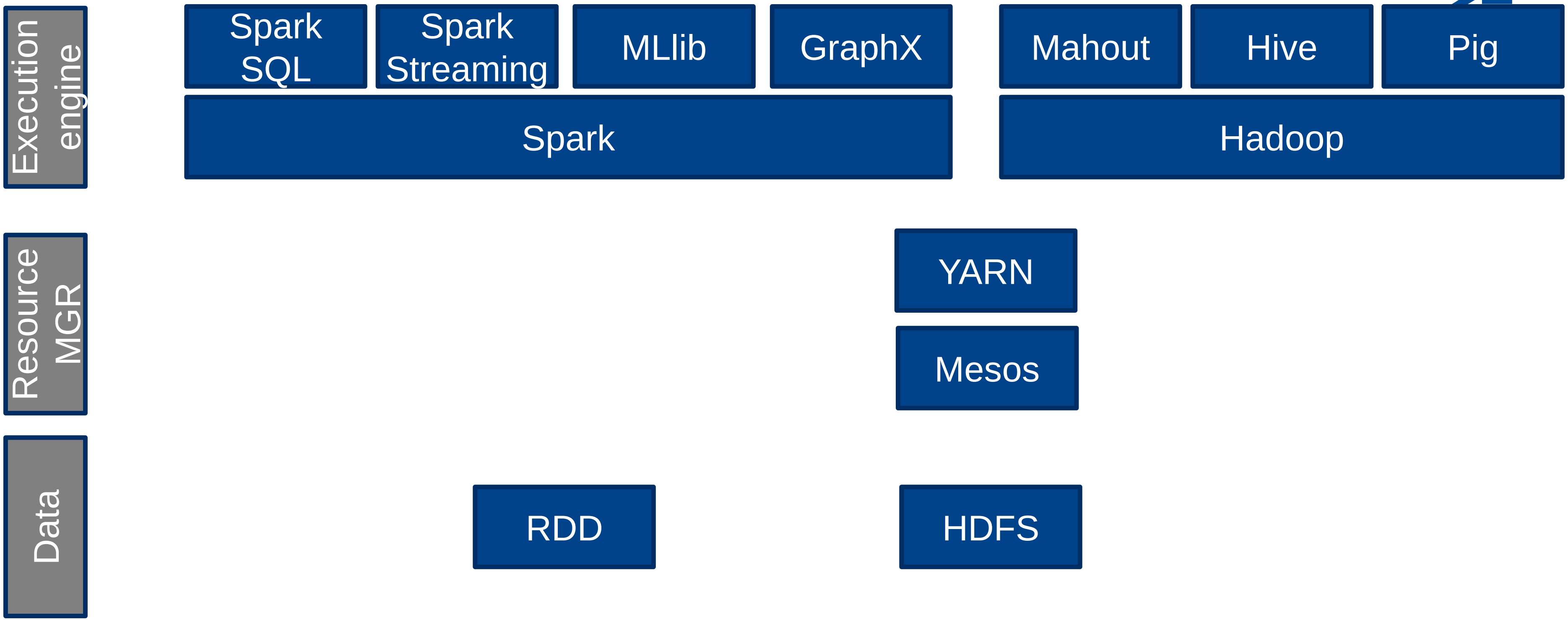
Albert-Ludwigs-Universität Freiburg

Prof. Dr. Georg Lausen
Patrick Philipp

# Apache Spark

- **Open-source, distributed, general-purpose cluster computing framework.**
- For processing large volumes of data, both *batch processing* and *streaming*.
- Supported features, among others:
  - analytics
  - machine learning
  - graph processing
- APIs for Scala, Python, Java, R, and SQL.
- Specialized modules that run on Spark core:
  - Spark SQL, Spark Streaming, MLlib (machine learning), GraphX (Graph processing)

# Spark Components



# Operations in Spark

- More general *functional* programming model than MapReduce: **Transformations** and **actions**
- Examples of transformations: *map, filter, groupBy*
- Examples of actions: *count, collect, save*
- (In MapReduce, transformation ≈ map(), action ≈ reduce())

# Data in Spark: RDDs

- **Resilient Distributed Datasets** (RDDs)
- RDD=Collection of elements spread across a cluster, which is
  - **immutable** (read-only) Nothing will be deleted in future
  - **resilient** (fault-tolerant: Automatically rebuilt in case of failure)
  - **distributed** (dataset spread out to more than one node)
- Stored in RAM or on disk.
- (Parallel) **Transformation**: Build a new RDD
- **Action**: Performed on RDD‘s elements or return result to program.

# Cluster Architecture

- **Job**: bunch of transformations & actions on RDDs
- **Cluster manager**: Allocates worker nodes

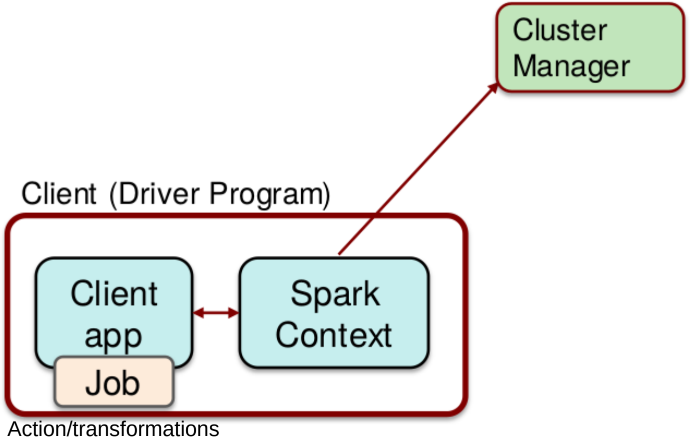



figure from Paul Krzyzanowski

# Cluster Architecture (2)

- **Driver** breaks the job into tasks
- Sends tasks to worker nodes where the data lives

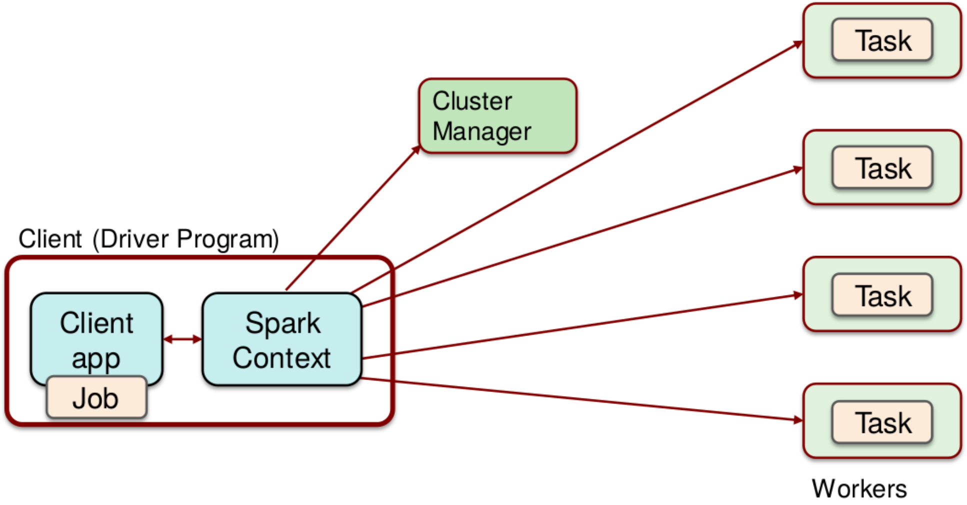
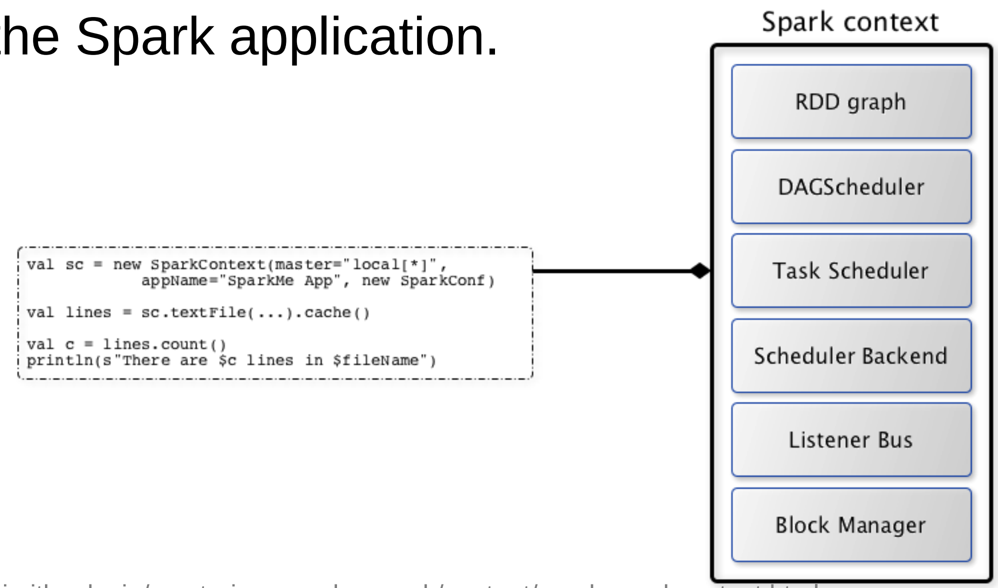



figure from Paul Krzyzanowski

# Cluster Architecture (3)

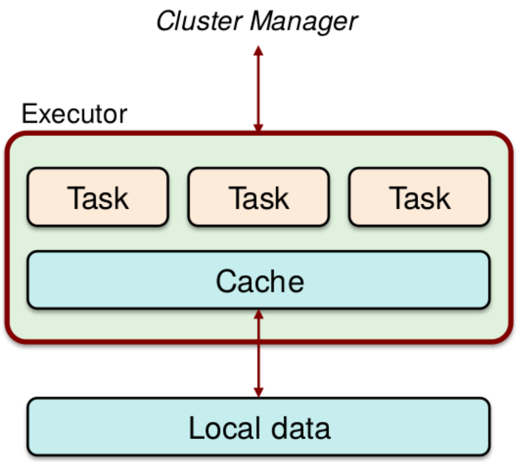
- **Spark context**: sets up internal services and establishes a connection to a Spark execution environment.
- masters the Spark application.




from: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-sparkcontext.html

# Worker Node

- Has one or more *executors*
- JVM process
- Talks with cluster manager
- Receives tasks
  - JVM code (e.g., compiled Java, Clojure, Scala, …)
  - Task = *transformation* or *action*
- *RDD*
  - *Data to be processed*
  - *local to the node*
- *Cache*
  - *frequently used data is kept in memory - for high performance!*



# Data Organization

- Organized in RDDs
- Idea: Partition (big) data across machines
- How are RDDs created:
  1. Create from any **file** stored in HDFS or other supported storage (Amazon S3, HBase, Cassandra, etc.)
     - Created externally (e.g., event stream, text files, database)
     - Examples:
       - Query a database & use results as RDD
       - Any Hadoop InputFormat, such as a list of files or a directory

# Data Organization (2)

2. *Streaming sources* (via Spark Streaming)
   - Fault-tolerant stream with a sliding window
3. An RDD can be the *output of a Spark transformation function.*
   - Example: filter out data, select key-value pairs

# Data Organization (3)

- Main properties of RDDs:
  - Immutable
    - You cannot change RDDs, only create new RDDs
    - The framework will eventually collect unused RDDs
  - Partitioned: RDDs distributed among servers
    - Default partitioning function: hash(key) mod server_count
- Optional properties of RDDs:
  - Typed: RDDs are not BLOBs
    - – Embedded data structure, e.g., key-value set
  - Ordered: Elements in an RDD can be sorted

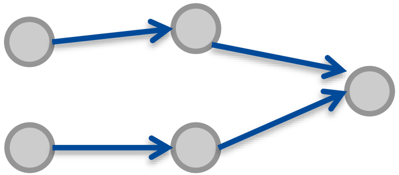
# Operations on RDDs

Two types of operations on RDDs:

1. *Transformations*
   - Lazy: not computed immediately, only if result required by driver program.
   - Transformed RDD is recomputed when an action is run on it
   - RDD can be persisted into memory or disk storage
2. *Actions*
   - *Finalizing* operations
     - e.g., reduce, count, grab samples, write to file

# Operations on RDDs (2)

- Transformations and Actions are generalizations of map() and reduce() of Hadoop
- In Spark: Directed Acyclic Graph (DAG) created and submitted to DAG scheduler
  - Nodes are RDDs
  - Arrows are transformations



- => more efficienct than MapReduce, since **DAG optimizer** rearranges the order of the operators and since data is cached.

# Spark Transformations

| Transformation | Description |
|---|---|
| **map**(func) | Pass each element through a function func |
| **filter**(func) | Select elements of the source on which func returns true |
| **flatmap**(func) | Each input item can be mapped to 0 or more output items |
| **sample**(withReplacement, fraction, seed) | Sample a *fraction* of the data, with or without replacement, using a given random number generator seed |
| **union**(otherdataset) | Union of the elements in the source data set and otherdataset |
| **distinct**([numtasks]) | The distinct elements of the source dataset |

# Spark Transformations (2)

| Transformation | Description |
|---|---|
| **groupByKey**([numtasks]) | When called on a dataset of (K,V) pairs, returns a dataset of (K,seq[V]) pairs |
| **reduceByKey**(func, [numtasks]) | Aggregate the values for each key using the given *reduce* function |
| **sortByKey**([ascending], [numtasks]) | Sort keys in ascending or descending order |
| **join**(otherDataset, [numTasks]) | Combines two datasets (K,V) and (K,W) into (K, (V,W)) |
| **cogroup**(otherDataset, [numtasks]) | Given (K,V) and (K,W), returns (K,Seq[V], Seq[W]) |
| **cartesian**(otherDataset) | For two datasets T and U, returns a dataset of (T,U) pairs |

# Spark Actions

| Action | Description |
|---|---|
| **reduce**(func) | Aggregate elements of the dataset using *func*. |
| **collect**(func, [numtasks]) | Return all elements of the dataset as an array |
| **count**() | Return the number of elements in the dataset |
| **first**() | Return the first element of the dataset |
| **take**(n) | Return an array with the first n elements of the dataset |
| **takeSample**(withReplacement, fraction, seed) | Return an array with a random sample of *num* elements of the dataset. |

# Spark Actions (2)

| Action | Description |
| --- | --- |
| **saveAsTextFile**(path) | Write dataset elements as a text file |
| **saveAsSequenceFile** (path) | Write dataset elements as a Hadoop SequenceFile |
| **countByKey**() | For (K,V) RDDs, return a map of (K, Int) pairs with the count of each key |
| **foreach**(func) | Run *func* on each element of the dataset |

Descriptions from Paul Krzyzanowski

# Spark in Action – Some Examples

- Obtain number of BSD licenses written in license file “LICENSE”

```
val licLines = sc.textFile("/usr/local/spark/LICENSE")
val bsdLines = licLines.filter(line => line.contains("BSD"))
bsdLines.count
bsdLines.foreach(bLine => println(bLine))
```

Download spark in action

# Spark in Action – map

```
val numbers = sc.parallelize(10 to 50 by 10)
numbers.foreach(x => println(x))
val numbersSquared = numbers.map(num => num * num)
numbersSquared.foreach(x => println(x))
```

# Spark in Action – distinct & flatMap

```
val lines = sc.textFile("/home/spark/client-ids.log")
val idsStr = lines.map(line => line.split(","))
ids.collect
idsStr.first

val ids = lines.flatMap(_.split(","))
ids.collect
ids.first

val intIds = ids.map(_.toInt)
intIds.collect

val uniqueIds = intIds.distinct
uniqueIds.collect
val finalCount = uniqueIds.count
```

# Spark in Action – sample

Prepare sample of 30% of clienIDs:

without replacement:

```
val s = uniqueIds.sample(false, 0.3)
s.collect
s.count
```

with replacement:

```
val swr = uniqueIds.sample(true, 0.5)
swr.collect
swr.count
```

# Spark in Action – take, takeSample

- **`takeSample`**:

Action!

```
val taken = uniqueIds.takeSample(false, 5)
```

- **`take`**:

```
uniqueIds.take(3)
```

# Spark in Action – Double values

- Implicit conversion of int to double in Scala
- Can be used for total sum of all of an RDD ’s elements and their mean value, standard deviation, and variance.
- Example code:

```
intIds.mean
intIds.sum
intIds.variance
intIds.stdev
```

# Note

- RDDs have new methods added to them automatically, depending on the type of data they hold.

# Data Storage

- Spark does not care how source data is stored
  - RDD connector determines that
- RDD fault tolerance
  - RDDs track the sequence of transformations used to create them
  - Enables recomputing of lost data
    - • Go back to the previous RDD and apply the transforms again

# Example: Log processing

- Transform (create new RDDs):
  1. Retrieve error message from a log file.
  2. Retrieve only ERROR messages and extract the source of error.
- Actions to perform: Count mysql errors and php errors

```
val lines = sc.textFile("hdfs://...”)
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

base RDD

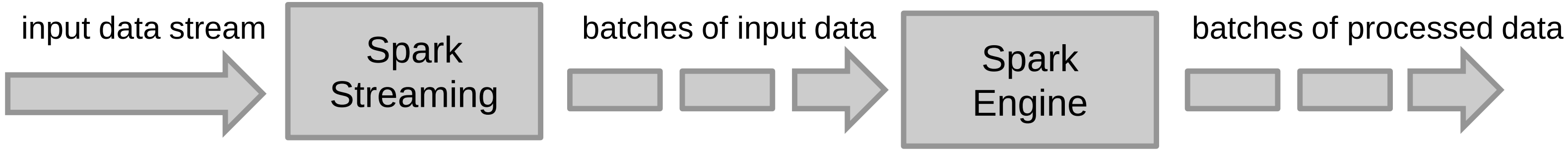
transformed RDDs

action 1

action 2

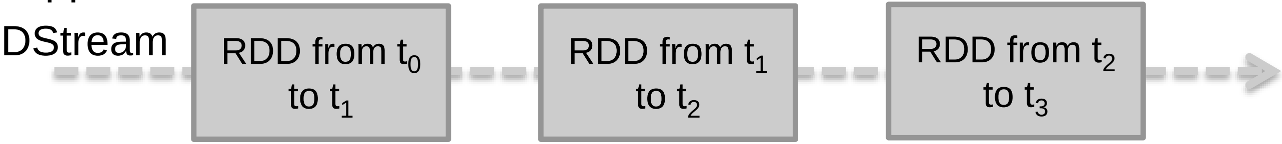
# Spark Streaming

- MapReduce, Pregel, etc. work on static data
- Spark Streaming enables processing of live data streams
  - Same programming operations
  - Input data is chunked into batches
    - Time interval specified in implementation



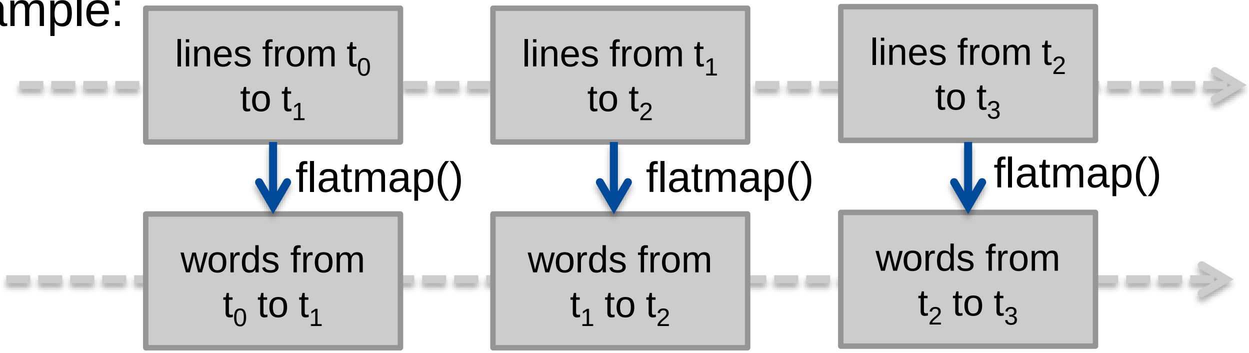
# Spark Streaming (2)

- Discretized Stream = DStream
  - Continuous stream of data (from source or a transformation)
  - Appears as a series of RDDs, each for a time interval




- Each operation on a DStream translates to operations on the RDDs

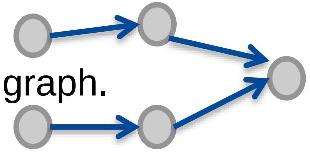
  Example:



- Join operations allow combining multiple streams

# Spark Core Summary

- Resilient Distributed Datasets (**RDDs**) as data collections.
  - RDDs are created from files (HDF etc.; storage agnostic), streaming sources, or output of Spark transformation function.
  - RDDs are immutable and distributed among servers.
  - Fault taulerant: RDDs can be regenerated.
- Spark operations: **Transformations** and **actions** on RDDs.
  - ensemble of different transformations and actions form a directed acyclic graph.



- Fast
  - Often up to 10x faster on disk and 100x faster in memory than MapReduce
  - General execution graph model
  - In-memory storage for RDDs
- Spark streaming: Handle continuous data streams via Spark Streaming

# MLlib: Machine Learning in Spark

- Example: **Linear SVM learning**
  from: https://spark.apache.org/docs/1.2.0/mllib-linear-methods.html#linear-support-vector-machines-svms

```
import ...

// Load training data in LIBSVM format.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")

// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4))
val training = splits(0).cache()
val test = splits(1)

// Run training algorithm to build the model
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)
```

# MLlib: Machine Learning in Spark (2)

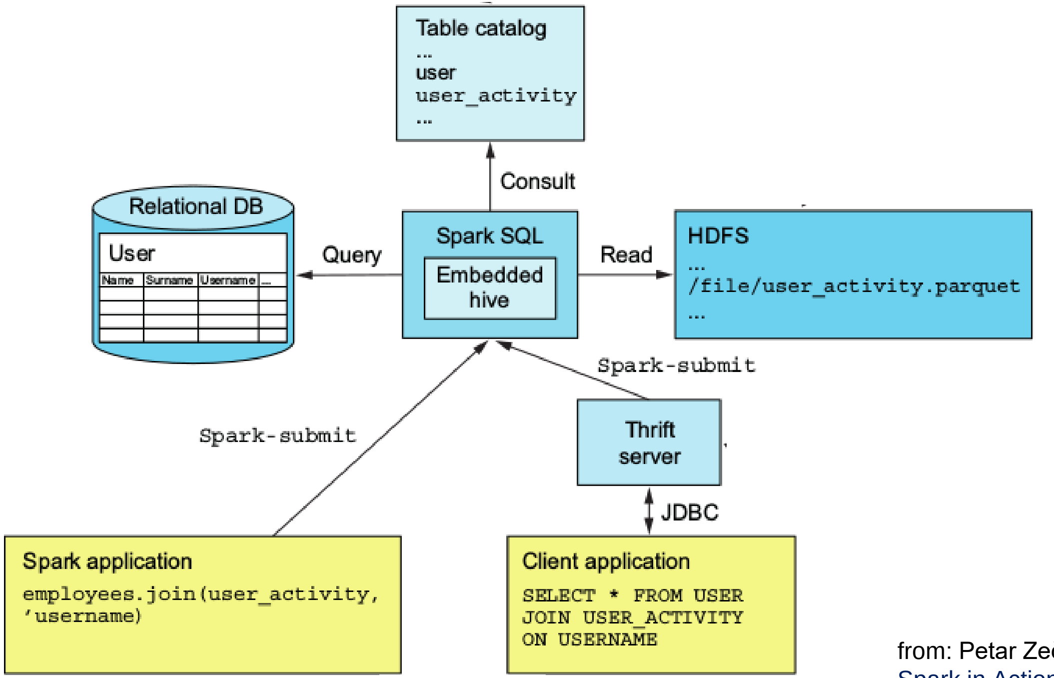
```
// Clear the default threshold.
model.clearThreshold()

// Compute raw scores on the test set.
val scoreAndLabels = test.map { point =>
  val score = model.predict(point.features)
  (score, point.label)
}

// Get evaluation metrics.
val metrics = new BinaryClassificationMetrics(scoreAndLabels)
val auROC = metrics.areaUnderROC()

// Print result (receiver operating characteristic curve)
println("Area under ROC = " + auROC)
```

# DataFrame in Spark

- Central component, supported since Spark 1.3
- Handle structured data in a table-like representation (with column names/types)
- Translate SQL code and domain-specific language (DSL) expressions into optimized low-level **RDD operations**
- Yield one common API for all languages (Scala, Java, Python, R)

- Since Spark 2.0: DataFrame is a row of a DataSet
- Registration of DataFrames in a table catalog to become visible/queryable
  - Only DataFrame name needed for querying.
- Spark exploits HiveQL, which is similar to SQL (see Hive in last lecture; not only used for MapReduce jobs, but also for Spark jobs)
  - It is also possible to query from remote via JDBC/ODBC

# Spark SQL Overview




from: Petar Zečević and Marko Bonaći: Spark in Action, Manning: 2016.

# How to create DataFrames?

1. Converting existing RDDs
2. Running SQL queries
3. Loading external data

# From RDDs to DataFrames

- Possibilities for creating a DataFrame from RDD:

a) Using RDDs containing row data as tuples

b) Using case classes

c) Specifying a schema

Schema is only infered for Option a) and b).

Option c) mostly used; there, explicitly specifying a schema.

# From RDDs to DataFrames

- Needed for transformation of RDDs to DataFrames:
  - **`SparkSession`**
    - **`SparkSession`** is a wrapper around **`SparkContext`** and **`SQLContext`** (which was directly used for creating DataFrames until Spark 2.0)
  - Implicit methods for transforming RDDs to DataFrames:

    ```
    import spark.implicits._
    ```

    - allows to call additional method **`toDF`**.

# a) Using RDDs containing row data as tuples

- Example in Scala:

```
scala> val itPostsRDD = itPostsSplit.map(x => (x(0),x(1),x(2),x(3),x(4),
x(5),x(6),x(7),x(8),x(9),x(10),x(11),x(12)))
itPostsRDD: org.apache.spark.rdd.RDD[(String, String, ...
scala> val itPostsDFrame = itPostsRDD.toDF()
itPostsDF: org.apache.spark.sql.DataFrame = [_1: string, ...
```

not very elegant

- Then one can work with it:

```
scala> itPostsDFrame.show(10)
+---+--------------------+---+--------------------+---+-------------------
| _1|                  _2| _3|                  _4| _5|                  _6
+---+--------------------+---+--------------------+---+-------------------
| 4|2013-11-11 18:21:...| 17|&lt;p&gt;The infi...| 23|2013-11-10 19:37:...
| 5|2013-11-10 20:31:...| 12|&lt;p&gt;Come cre...| 1|2013-11-10 19:44:...
| 2|2013-11-10 20:31:...| 17|&lt;p&gt;Il verbo...| 5|2013-11-10 19:58:...
...
```

# a) Using RDDs containing row data as tuples (2)

- Recommended: column names given automatically, rename column names with toDF function:

```
scala> val itPostsDF = itPostsRDD.toDF("commentCount", "lastActivityDate",
"ownerUserId", "body", "score", "creationDate", "viewCount", "title",
"tags", "answerCount", "acceptedAnswerId", "postTypeId", "id")
```

- Check (print) column names:

```
scala> itPostsDF.printSchema
root
|-- commentCount: string (nullable = true)
|-- lastActivityDate: string (nullable = true)
|-- ownerUserId: string (nullable = true)
|-- body: string (nullable = true)
|-- score: string (nullable = true)
```

# b) RDDs to DataFrames using Case Classes

- In columns all strings so far => Introduce data types via case classes.
- Steps: (a) define case class (b) map each row in RDD to case class (c) use toDF method.
- Example for (a):

```
import java.sql.Timestamp
case class Post(
 commentCount:Option[Int],
 lastActivityDate:Option[java.sql.Timestamp],
 ownerUserId:Option[Long],
 body:String,
 score:Option[Int],
 creationDate:Option[java.sql.Timestamp],
 ...
 postTypeId:Option[Long],
 id:Long)
```

# b) RDDs to DataFrames using Case Classes (2)

- For better conversion (of different data types, i.e., to not only have strings), define class and methods as follows:

```
object StringImplicits {
 implicit class StringImprovements(val s: String) {
 import scala.util.control.Exception.catching
 def toIntSafe = catching(classOf[NumberFormatException]) opt s.toInt
 def toLongSafe = catching(classOf[NumberFormatException]) opt s.toLong
 def toTimestampSafe = catching(classOf[IllegalArgumentException]) opt
 Timestamp.valueOf(s)
 }
}
```

# b) RDDs to DataFrames using Case Classes (3)

- Actual transformation:

```
import StringImplicits._
def stringToPost(row:String):Post = {
val r = row.split("~")
Post(r(0).toIntSafe,
r(1).toTimestampSafe,
r(2).toLongSafe,
r(3),
r(4).toIntSafe,
r(5).toTimestampSafe,
r(6).toIntSafe,
r(7),
...
r(11).toLongSafe,
r(12).toLong)
}
val itPostsDFCase = itPostsRows.map(x => stringToPost(x)).toDF()
```

# b) RDDs to DataFrames using Case Classes (4)

- Check if tranformation successful:

```
scala> itPostsDFCase.printSchema
root
|-- commentCount: integer (nullable = true)
|-- lastActivityDate: timestamp (nullable = true)
|-- ownerUserId: long (nullable = true)
|-- body: string (nullable = true)
|-- score: integer (nullable = true)
|-- creationDate: timestamp (nullable = true)
|-- viewCount: integer (nullable = true)
|-- title: string (nullable = true)
|-- tags: string (nullable = true)
|-- answerCount: integer (nullable = true)
|-- acceptedAnswerId: long (nullable = true)
|-- postTypeId: long (nullable = true)
|-- id: long (nullable = false)
```

# c) From RDDs to DataFrames by Specifying a Schema

- Use **SparkSession**'s **createDataFrame** method (needed: objects of type **Row** and a **StructuredType** (=schema)).
- **StructuredType** definition example:

```
import org.apache.spark.sql.types._
val postSchema = StructType(Seq(
 StructField("commentCount", IntegerType, true),
 StructField("lastActivityDate", TimestampType, true),
 StructField("ownerUserId", LongType, true),
 StructField("body", StringType, true),
 StructField("score", IntegerType, true),
 StructField("creationDate", TimestampType, true),
 StructField("viewCount", IntegerType, true),
 ...
 StructField("id", LongType, false))
)
```

Supported data types: strings, integers, shorts, floats, doubles, bytes, dates, timestamps, binary values, arrays, maps, structs.

# c) From RDDs to DataFrames by Specifying a Schema (2)

- Actual transformation (now **`Row`**, not **`Post`** type; the rest is the same as above):

```
def stringToRow(row:String):Row = {
val r = row.split("~")
Row(r(0).toIntSafe.getOrElse(null),
r(1).toTimestampSafe.getOrElse(null),
r(2).toLongSafe.getOrElse(null),
r(3),
r(4).toIntSafe.getOrElse(null),
r(5).toTimestampSafe.getOrElse(null),
r(6).toIntSafe.getOrElse(null),
r(7),
r(8),
r(9).toIntSafe.getOrElse(null),
r(10).toLongSafe.getOrElse(null),
r(11).toLongSafe.getOrElse(null),
r(12).toLong)
}
```

# c) From RDDs to DataFrames by Specifying a Schema (3)

```
val rowRDD = itPostsRows.map(row => stringToRow(row))
val itPostsDFStruct = spark.createDataFrame(rowRDD, postSchema)
```

- Check ifeverything is fine, e.g., by **`.columns`**-method (=print column names) or by **`.dtypes`**-method (column names plus data types):

```
scala> itPostsDFCase.columns
res0: Array[String] = Array(commentCount, lastActivityDate, ownerUserId,
body, score, creationDate, viewCount, title, tags, answerCount,
acceptedAnswerId, postTypeId, id)
```

# How to Use the DataFrame API

- Example:

```
scala> val postsDf = itPostsDFStruct
scala> val postsIdBody = postsDf.select("id", "body")
postsIdBody: org.apache.spark.sql.DataFrame = [id: bigint, body: string]
```

- Using `col`-method:

```
val postsIdBody = postsDf.select(postsDf.col("id"), postsDf.col("body"))
```

- To retrieve all columns except one: e.g.

```
val postIds = postsIdBody.drop("body")
```

  - removes the `body` column from the `postsIdBody` DataFrame

# How to Use the DataFrame API (2)

- Filtering data:
  - using **`where`** or **`filter`** functions (synonymous!)
- Examples:

```
scala> postsIdBody.filter('body contains "Italiano").count
res0: Long = 46
```

Hyphon needed for scala parser

```
scala> val noAnswer = postsDf.filter(('postTypeId === 1) and
('acceptedAnswerId isNull))
```

- Return top n elements, using limit:

```
scala> val firstTenQs = postsDf.filter('postTypeId === 1).limit(10)
```

# SQL Functions in Spark

- SQL functions are available through
  1. the *DataFrame API and*
  2. *SQL expressions*.

# SQL Functions Using the DataFrame API

- 4 kinds of SQL functions when using the DataFrame API:
  a) *Scalar functions* return a single value for each row based on calculations on one or more columns.
  b) *Aggregate functions* return a single value for a group of rows.
  c) *Window functions* return several values for a group of rows.
  d) *User-defined functions* include custom scalar or aggregate functions.

# a) Scalar Functions

- Math calculations:
  - **abs** (calculates absolute value),
  - **hypot** (calculates hypote-nuse based on two columns or scalar values),
  - **log** (calculates logarithm),
  - **cbrt** (computes cube root), and others
- String operations:
  - **length** (calculates length of a string),
  - **trim** (trims a string value left and right),
  - **concat** (concatenates several input strings), and others
- Date-time operations:
  - **year** (returns the year of a date column),
  - **date_add** (adds a number of days to a date column), and others.

# c) Window Functions

- Window functions: for making selects and joins simpler.
- First import by

```
import org.apache.spark.sql.expressions.Window
```

# Using SQL Commands in Spark

- Idea: Use SQL, since SQL is widely used and easier
- SQL commands get translated into DataFrames
- Spark supports (1) Spark's SQL dialect and (2) Hive Query Language (HQL).
- HQL is recommended, since richer functionalities and more support from community.
- One can store DataFrames as tables in a *table catalog* – either temporarily or permanently.
  - *Temporary* registration:

```
postsDf.createOrReplaceTempView("posts_temp")
```

  Afterwards, you are ably to query the data with SQL commands
  - Permanent registration:

```
postsDf.write.saveAsTable("posts")
votesDf.write.saveAsTable("votes")
```

SparkSession with Hive support can be used to register table definitions that will survive your application's restarts

# Using SQL Commands in Spark (2)

- Checking the table catalog:

```
scala> spark.catalog.listTables().show()
```

- Execute SQL queries then:

```
val resultDf = sql("select * from posts")
```

- Result is a DataFrame again.
- Spark SQL shell (command: `spark-sql`), which provides additional functions beyond spark-shell and spark-submit.
- Example:

```
spark-sql> select substring(title, 0, 70) from posts where
postTypeId = 1 order by creationDate desc limit 3;
```

UNI FREIBURG

# JDBC/ODBC Connection

- Note: Besides executing SQL queries directly from programs or through SQL shell: SQL commands via JDBC (or ODBC) possible, namely via Thrift (which is a special Spark application).

# Acknowledgements.

- Slides are partially based on
  - Petar Zečević and Marko Bonaći: Spark in Action, Manning: 2016.
  - Paul Krzyzanowski, Rutgers University, 2016.