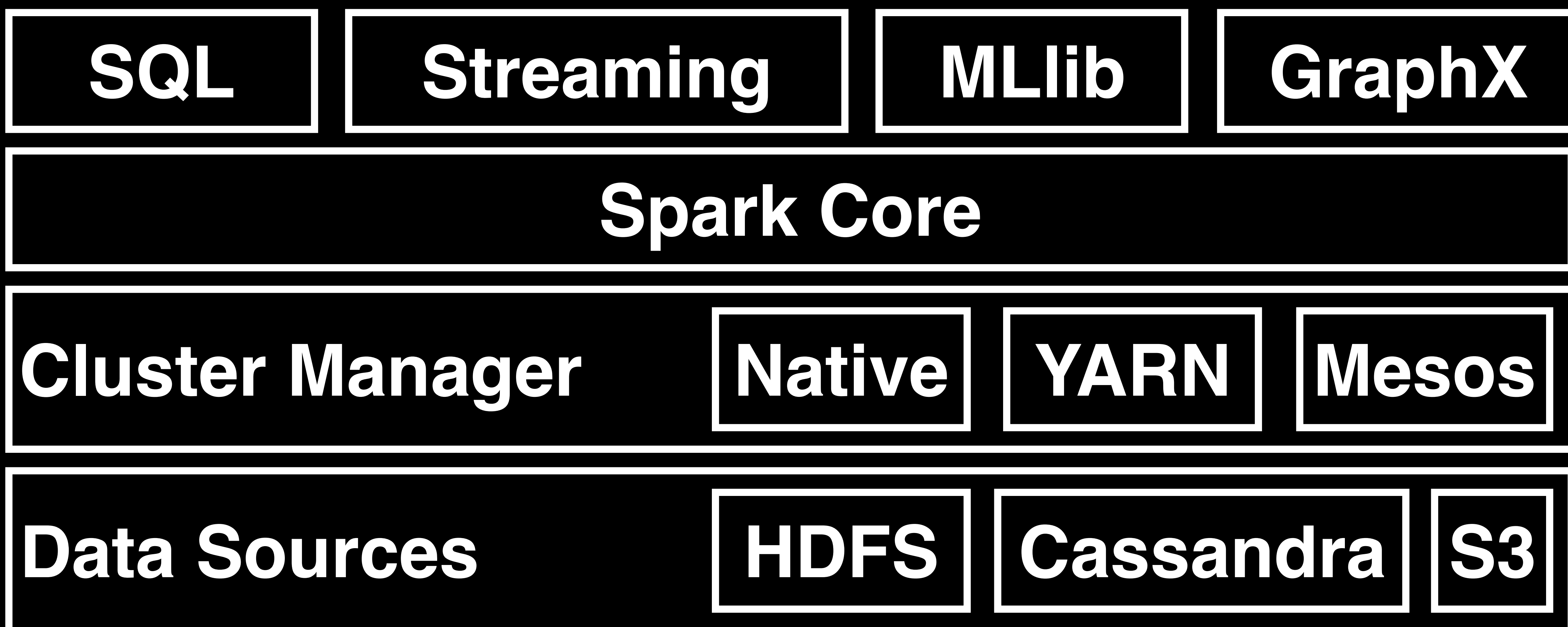
Spark Architecture



Spark SQL

- Is not only about Databases
- Can deal with Json, and other formats too
- Introduces ***DataFrame*** which is an RDD of ***Row*** objects.

Loading Data

```
import org.apache.spark.sql.hive.HiveContext
val sc = …
val hiveCtx = new HiveContext(sc)
val input = hiveCtx.jsonFile("../data/tweets/*/*")
```

Loading Data

```
import org.apache.spark.sql.SQLContext
val sc = …
val sqlCtx = new SQLContext(sc)
val input = sqlCtx.jsonFile(“../data/tweets/*/*”)
```

Loading Data

- Note that shell creates sqlContext for you

```
scala> sqlContext
res1: org.apache.spark.sql.SQLContext =
org.apache.spark.sql.hive.HiveContext@dc72335

val hiveCtx =
sqlContext.asInstanceOf[org.apache.spark.sql.hive.HiveContext]
```

- Usually it is an instance of HiveContext, you need to cast it yo use additional functionality in hive

Schema

- Spark can predict the schema for you!

```
input.printSchema
```

```
root
 |-- contributorsIDs: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- createdAt: string (nullable = true)
 |-- currentUserRetweetId: long (nullable = true)
 |-- geoLocation: struct (nullable = true)
 |    |-- latitude: double (nullable = true)
 |    |-- longitude: double (nullable = true)
 |-- hashtagEntities: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- end: long (nullable = true)
 |    |    |-- start: long (nullable = true)
 |    |    |-- text: string (nullable = true)
```

DataFrame Operations

- DataFrame introduces new operations

```
input.count
input.select("id").take(5)
```

Traditional RDD Operations

- DataFrame still supports map, flatMap, filter

```
input.rdd.filter(_.getString(1).startsWith("Oct
11")).first
```

RDD[Row]

- DataFrame is basically an RDD of Row objects

```
scala> val row = input.rdd.first
row: org.apache.spark.sql.Row = …
```

Row object

- Row objects know their schema

```
scala> row.schema
res39: org.apache.spark.sql.types.StructType =
StructType(
  StructField(createdAt,StringType,true),
  StructField(currentUserRetweetId,LongType,true),
…
```

Row object

- You still need to specify the types (casting)

```
scala> row.get(1)
res1: Any = Oct 11, 2015 5:22:29 PM

scala> row.getString(1)
res2: String = Oct 11, 2015 5:22:29 PM

scala> row.getAs[String](1)
res2: String = Oct 11, 2015 5:22:29 PM

scala> row.getBoolean(1)
java.lang.ClassCastException: java.lang.String cannot be
cast to java.lang.Boolean
```

Row object

- Row objects can be nested

```
StructField(
  geoLocation,
  StructType(
    StructField(
      latitude,DoubleType,true),
    StructField(
       longitude,DoubleType,true)),true),
```

Row object

- Row objects can be nested

```
scala> row.getStruct(15).getString(1)
res68: String = Oct 11, 2015 2:26:26 PM
```

Python API

- Row looks prettier using Python
- Because Python is already dynamically typed

```
from pyspark.sql import SQLContext, Row
sqlCtx = SQLContext(sc)
input = sqlCtx.jsonFile("../data/tweets/*/*")
row = input.rdd.first

row[1]

row.createdAt
```

Top 10 Tweets

- Let’s print top 10 tweets in terms for retweets
- scala> row.getLong(14)
- python> row[17]
- python> row.retweetCount
- python> row.text

Top Tweet

- Traditional RDD transformations

```python
input.rdd.sortBy(lambda r: r.retweetCount).first().text
```

Python

Scala

```scala
input.rdd.sortBy(_.getLong(14)).first.getString(17)
```

Top Tweet

- Traditional RDD transformations

```python
input.orderBy("retweetCount").first().text
```

Python

Scala

```scala
input.orderBy("retweetCount").first.getString(17)
```

Top Tweet

- DataFrame transformations

```scala
input
  .orderBy(“retweetCount")
  .select(“text")
  .first
  .get(0)
```

Scala

```python
input
  .orderBy(“retweetCount")
  .select(“text")
  .first()
  .text
```

Python

Top Tweet

- Using SQL

```
input.registerTempTable("tweets")

sqlCtx.sql("""
SELECT text FROM tweets ORDER BY retweetCount LIMIT 1
“”").first()
```

Scala

Python

SQL

- Nested fields

```
sqlCtx.sql("""
SELECT user.favouritesCount FROM tweets
""").first()
```

SQL

- multiple aggregations

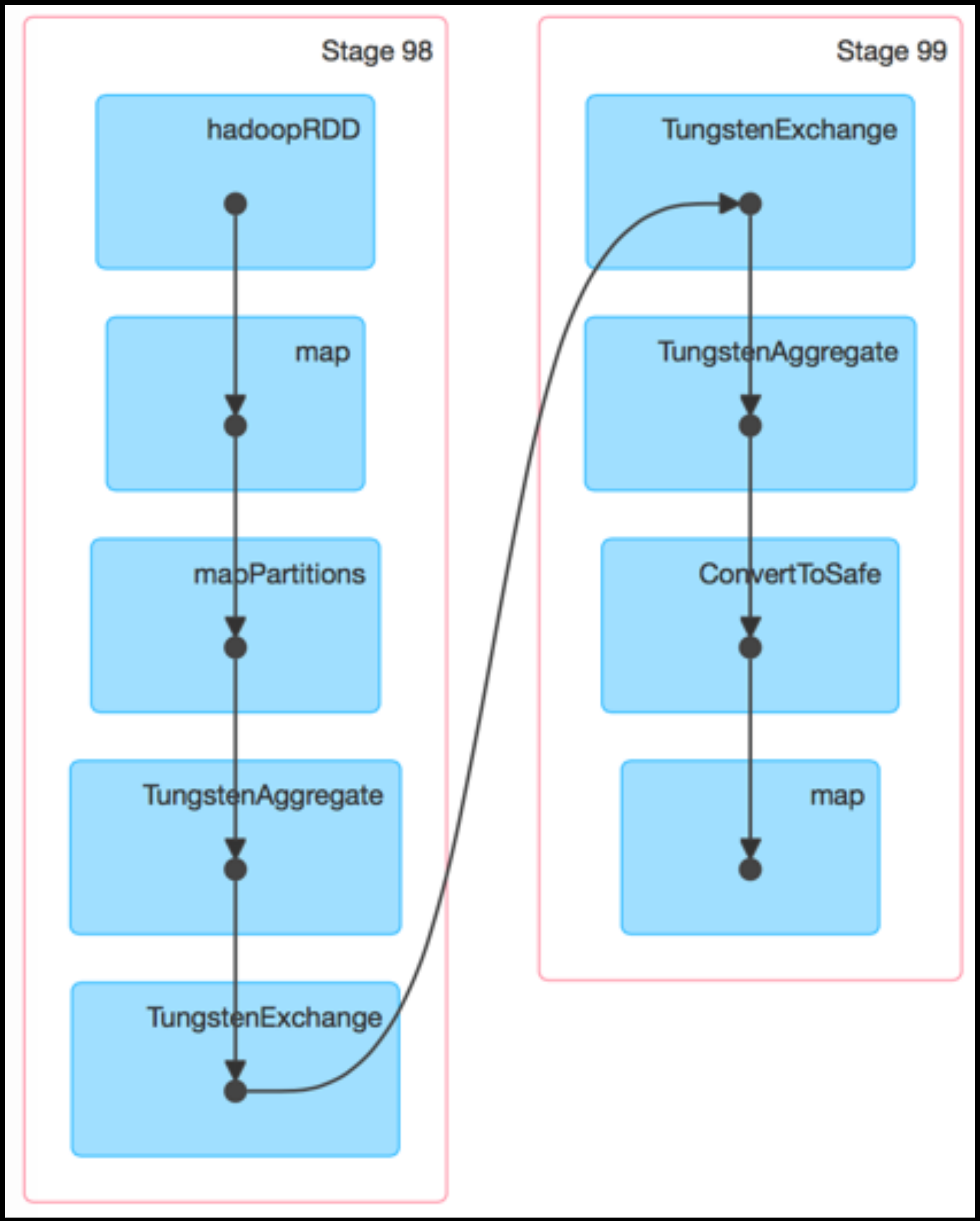
```
sqlCtx.sql("""

SELECT sum(user.favouritesCount), sum(retweetCount)
 FROM tweets

""").first()
```

Performance

- SQL is compiled into normal map and reduce
- SQL is higher level that functional style
- Spark can optimise what you “want”



Performance

- This can be extremely inefficient if written manually
- Spark can optimise this by filtering before joining

```
sqlCtx.sql("""
SELECT t1.id, t2.id
FROM table1 t1 JOIN table2 t2
WHERE t1.field < 10
“”").first()
```

Performance

- Even more, It can “push predicates down”
- Push query to data source to return data already filtered
- Depending on data source (if real database)

```
sqlCtx.sql("""
SELECT t1.id, t2.id
FROM table1 t1 JOIN table2 t2
WHERE t1.field < 10
“”").first()
```

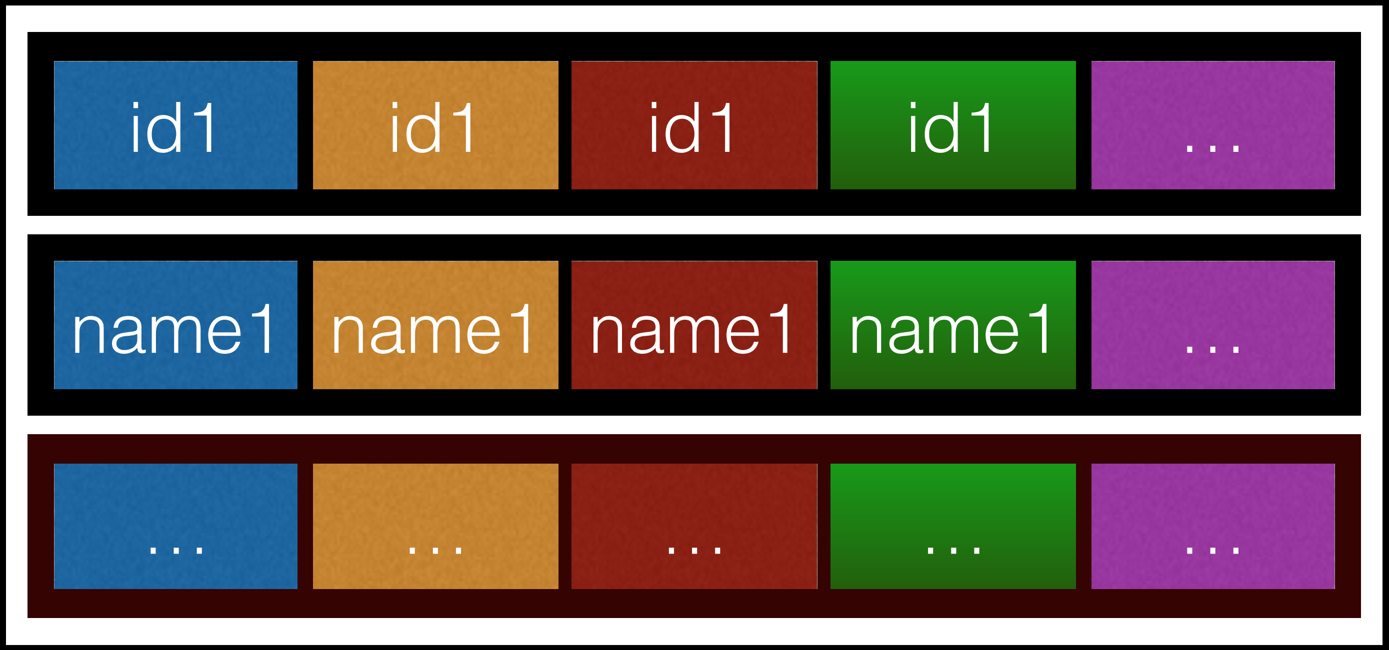
Performance

- Also, It can load only required fields
- Depending on data format (like Apache Parquet)
- Resulting in less network traffic

```
sqlCtx.sql("""
SELECT t1.id, t2.id
FROM table1 t1 JOIN table2 t2
WHERE t1.field < 10
“”").first()
```

Performance

- Row objects are very compact
- Uses multiple arrays



Caching

- Using SQL

```
sqlCtx.sql(…).cache()
sqlCtx.cacheTable("tableName")
sqlCtx.sql("""cache table tableName""")
```

```
sqlCtx.sql(…).map(…).cache()
```

Spark 1.5.1

Jobs Stages Storage Environme

Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Externa
In-memory table tweets	Memory Deserialized 1x Replicated	55	100%	19.1 MB	0.0 B
MapPartitionsRDD	Memory Deserialized 1x Replicated	55	100%	137.1 MB	0.0 B

DataFrames from RDDs

```
case class MyClass(field: String)
val rdd = sc.parallelize(List(MyClass("value")))
val dataframe = sqlCtx.createDataFrame(rdd)
dataframe.registerTempTable("myTable")
```

```
rdd = sc.parallelize([Row(field="value", ...)])
datafram = sqlCtx.inferSchema(rdd)
rdd.registerTempTable("myTable")
```

DataFrames from RDDs

- You can now use normal RDDs in SQL

```
sqlCtx.sql(“""select * from myTable join anotherTable”"")
```

User Defined Functions

- You can use your Scala or Python function inside SQL!

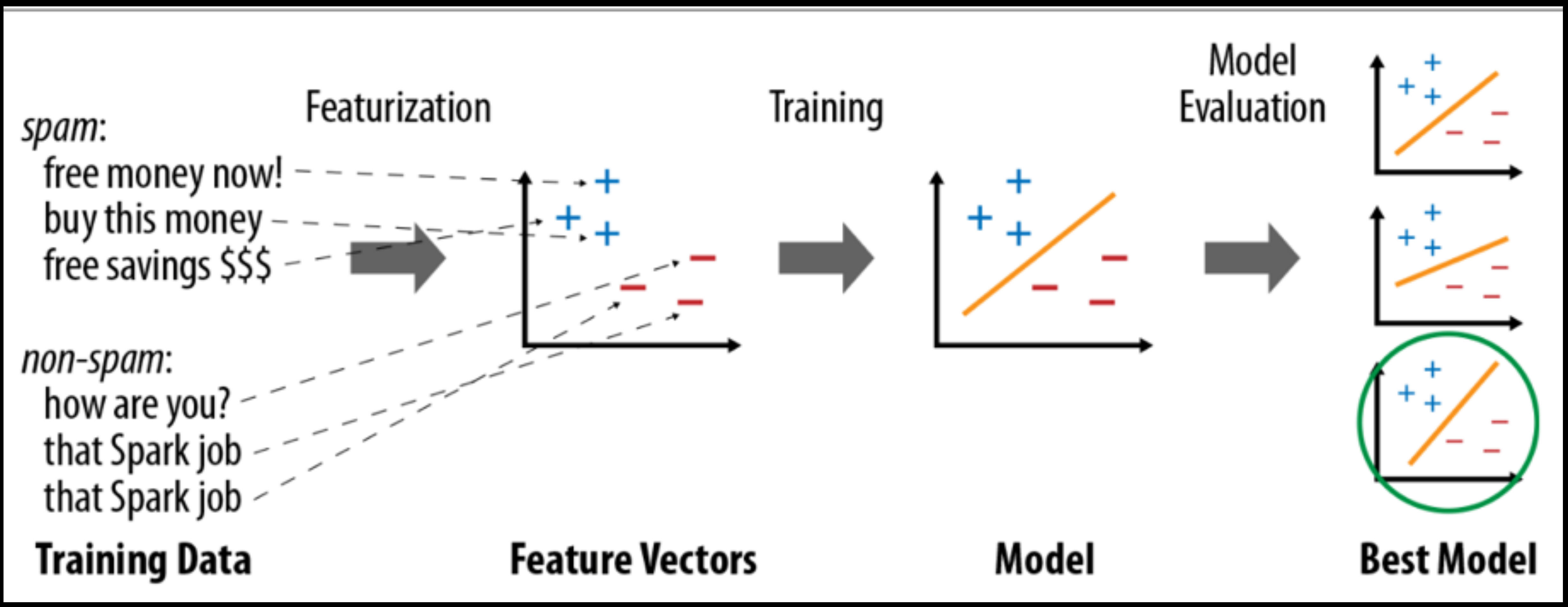
```
sqlContext.udf.register("len", (x: String) => x.length)
sqlContext.sql("""
select sum(len(text)) from tweets
""").collect
```

Array[org.apache.spark.sql.Row] = Array([725031])

MLLib

- Spark provides library for Machine learning
- K-means is a popular ML algorithm
- Used to classify/cluster data into groups based on similarity (called features)

Classification



K-means

```
val texts = sqlContext.sql("SELECT text from
tweets").map(_.getString(0))
```

- Read data and extract only tweet’s text

K-means

```
import org.apache.spark.mllib.feature.HashingTF
val tf = new HashingTF(1000)
def featurize(s: String) =
  tf.transform(s.sliding(2).toSeq)

val vectors = texts.map(featurize).cache
```

- Converts string into Vector of numbers

```
scala> “abcd".sliding(2).toArray
res48: Array[String] = Array(ab, bc, cd)
```

K-means

```
import org.apache.spark.mllib.clustering.KMeans
val model =
  KMeans.train(vectors, numClusters, numIterations)
```

- Now we have a trained model
- We can use it to classify new tweets

K-means

- Let’s have a look on the data

```
val groups =
  texts.groupBy(t => model.predict(featurize(t)))
groups.foreach(_.println)

model.predict(featurize(“test”))
```