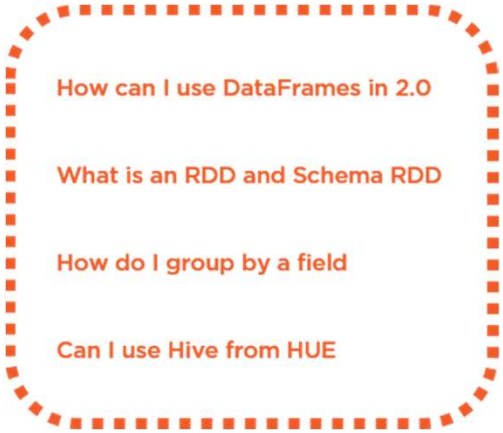
Increasing Proficiency with Spark: DataFrames & Spark SQL

Xavier Morera

HELPING DEVELOPERS UNDERSTAND SEARCH & BIG DATA

@xmorera www.xaviermorera.com

Increasing Proficiency with Spark: DataFrames & Spark SQL



Increasing Proficiency with Spark: DataFrames & Spark SQL

History repeats itself...

Have you ever heard?

Before Hadoop

Early Days of Hadoop

Early Days of Hadoop

Lingua franca for data analysis

SQL (Structured Query Language)

Everyone Uses SQL

Extremely popular for many years

Business analysts to developers

Easy to learn, understand and use

Supported by many applications

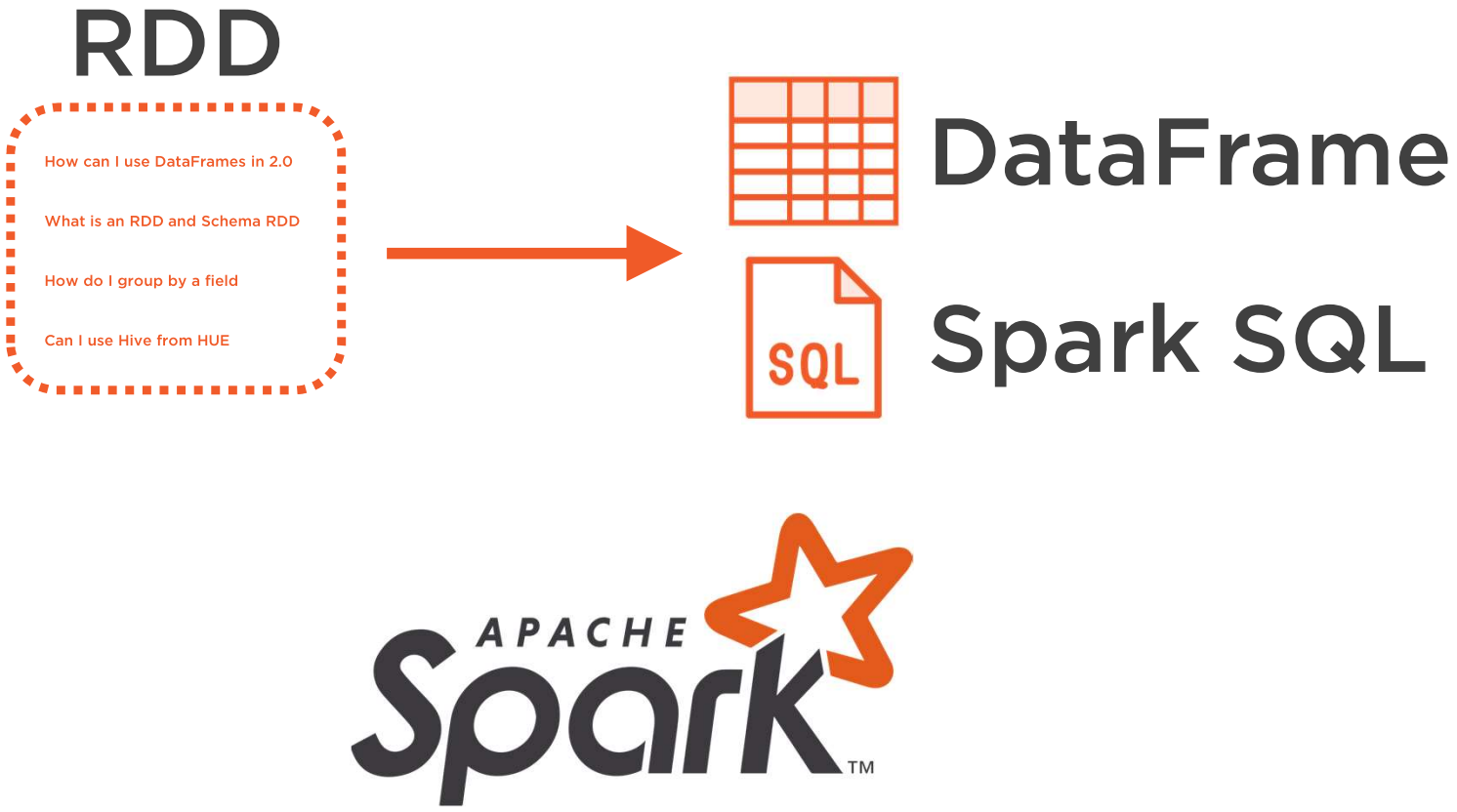
- Beyond databases

Early Days of Hadoop

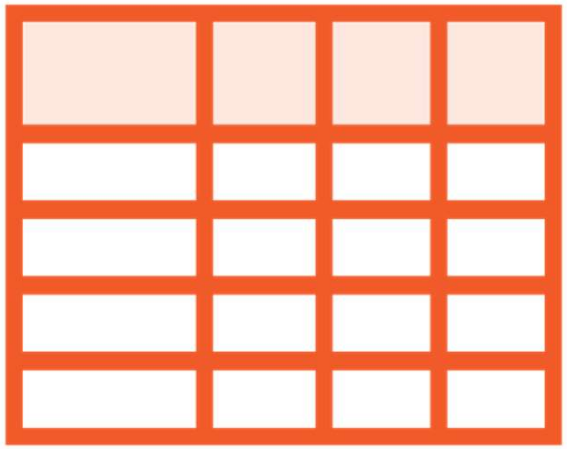
Early Days of Hadoop

Java

Spark


RDD
How can I use DataFrames in 2.0
What is an RDD and Schema RDD
How do I group by a field
Can I use Hive from HUE
DataFrame
SQL
Spark SQL
APACHE
Spark™

The Beginnings of the API



Shark

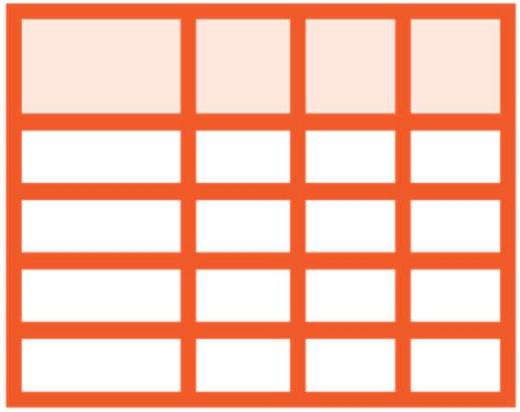
- SQL using Spark execution engine
- Evolved into Spark SQL in 1.0

SchemaRDD

- RDD with schema information
- For unit testing & debugging Spark SQL
- Drew attention by Spark developers
- Released as DataFrame API in 1.3

Hello DataFrames & Spark SQL

Spark SQL



DataFrame

Spark SQL

Module for structured data processing

Schema

Give Spark more information on the data

Optimizations

Spark SQL

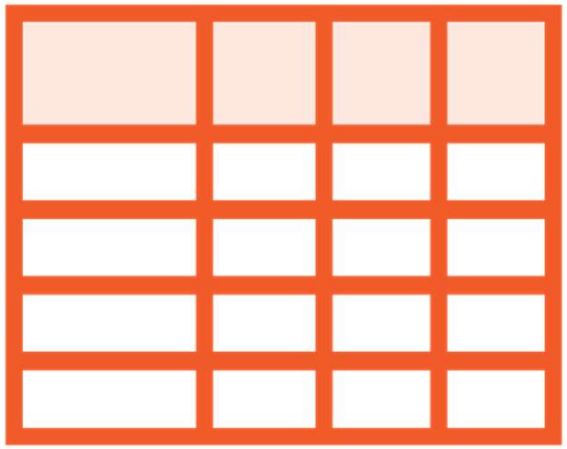
Interact via SQL queries

- ANSI SQL 2003 support

Works with Hive

Any data source compatible with Spark

DataFrame



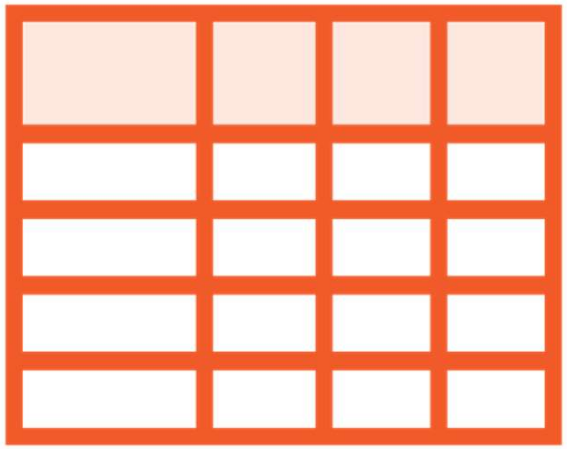
Distributed collection of Row objects

- Dataset[Row]

Equivalent to a database table

- Rows and columns,
- Known schema

DataFrame



Structured or unstructured data

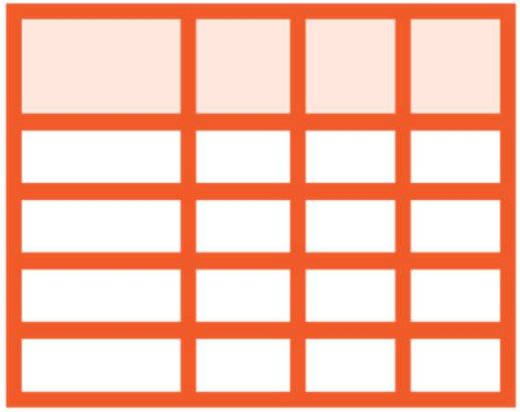
- Conversion to and from RDD possible

Queries

- Domain Specific Language (DSL)
- Relational
- Allows for optimizations

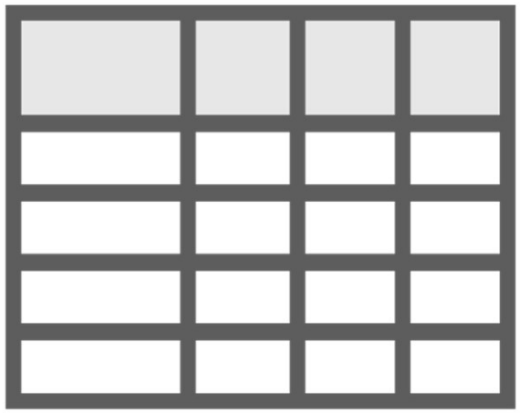
DataFrames & Spark SQL

Spark SQL



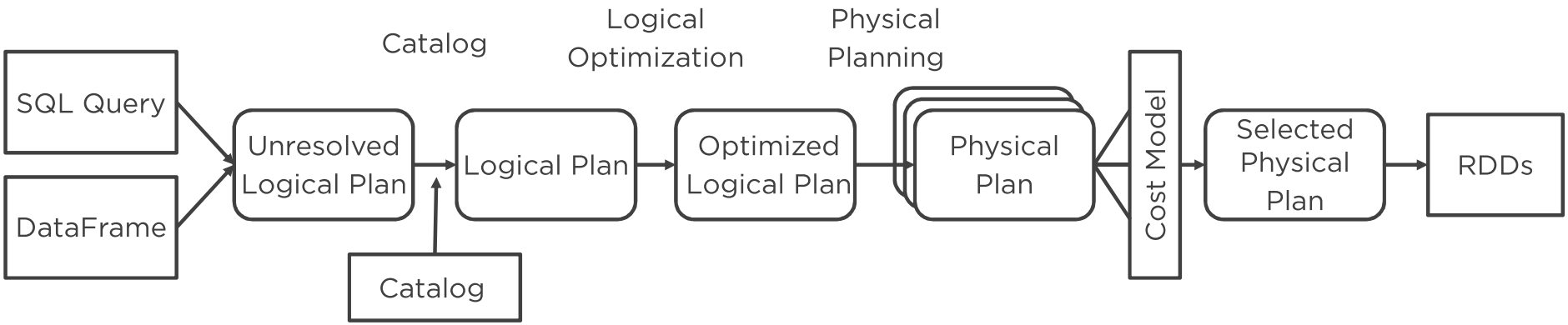
DataFrame

Spark SQL



DataFrame

Execution


Catalog
Logical Optimization
Physical Planning
SQL Query
DataFrame
Unresolved Logical Plan
Logical Plan
Optimized Logical Plan
Physical Plan
Cost Model
Selected Physical Plan
RDDs
Catalog

In a previous module...

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder.master("yarn")
            .appName("StackOverflowTest")
            .config("spark.executor.memory", "2g")
            .getOrCreate()
```

SparkSession

Entry point to Spark SQL

- Merges SQLContext, HiveContext. References SparkContext

```
# spark2-shell
```

Getting a SparkSession

Created automatically for you in spark2-shell (REPL)

But you need to create it for self contained applications

```
[root@dn01 no_session]# sbt package
[info] Loading project definition from /spark-scala-cloudera-demos/2 - Getting an Environment and
 Data - CDH + StackOverflow/no_session/project
[info] Loading settings from build.sbt ...
[info] Set current project to No Session (in build file:/spark-scala-cloudera-demos/2%20-%20Getti
ng%20an%20Environment%20and%20Data%20-%20CDH%20+%20StackOverflow/no_session/)
[info] Compiling 1 Scala source to /spark-scala-cloudera-demos/2 - Getting an Environment and Dat
a - CDH + StackOverflow/no_session/target/scala-2.11/classes ...
[error] /spark-scala-cloudera-demos/2 - Getting an Environment and Data - CDH + StackOverflow/no_
session/src/main/scala/simple_no_session.scala:18:33: not found: value spark
[error]   print ("Application name: " + spark.sparkContext.appName + " / Version: " + spark.versi
on)
[error]                                 ^
[error] /spark-scala-cloudera-demos/2 - Getting an Environment and Data - CDH + StackOverflow/no_
session/src/main/scala/simple_no_session.scala:18:79: not found: value spark
[error]   print ("Application name: " + spark.sparkContext.appName + " / Version: " + spark.versi
on)
[error]                                                                               ^
[error] /spark-scala-cloudera-demos/2 - Getting an Environment and Data - CDH + StackOverflow/no_
session/src/main/scala/simple_no_session.scala:19:3: not found: value spark
[error]   spark.stop()
[error]   ^
[error] three errors found
[error] (Compile / compileIncremental) Compilation failed
[error] Total time: 3 s, completed Mar 6, 2018 10:23:11 AM
[root@dn01 no_session]#
```

```
n.engine=mr.
18/03/06 10:22:19 INFO client.HiveClientImpl: Warehouse location for Hive client (version 1.1.0)
is /user/hive/warehouses
18/03/06 10:22:19 INFO state.StateStoreCoordinatorRef: Registered StateStoreCoordinator endpoint
*** Testing simple_with_session.scala ***
Application name: StackOverflowTest / Version: 2.2.0.cloudera118/03/06 10:22:19 INFO server.Abstr
actConnector: Stopped Spark@2d0566ba{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
18/03/06 10:22:19 INFO ui.SparkUI: Stopped Spark web UI at http://10.0.2.101:4040
18/03/06 10:22:19 INFO cluster.YarnClientSchedulerBackend: Interrupting monitor thread
18/03/06 10:22:19 INFO cluster.YarnClientSchedulerBackend: Shutting down all executors
18/03/06 10:22:19 INFO cluster.YarnSchedulerBackend$YarnDriverEndpoint: Asking each executor to s
hut down
18/03/06 10:22:19 INFO cluster.SchedulerExtensionServices: Stopping SchedulerExtensionServices
(serviceOption=None,
 services=List(),
 started=false)
18/03/06 10:22:19 INFO cluster.YarnClientSchedulerBackend: Stopped
18/03/06 10:22:19 INFO spark.MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopp
ed!
18/03/06 10:22:19 INFO memory.MemoryStore: MemoryStore cleared
18/03/06 10:22:19 INFO storage.BlockManager: BlockManager stopped
18/03/06 10:22:19 INFO storage.BlockManagerMaster: BlockManagerMaster stopped
18/03/06 10:22:19 INFO scheduler.OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputC
ommitCoordinator stopped!
18/03/06 10:22:19 INFO spark.SparkContext: Successfully stopped SparkContext
18/03/06 10:22:19 INFO util.ShutdownHookManager: Shutdown hook called
18/03/06 10:22:19 INFO util.ShutdownHookManager: Deleting directory /tmp/spark-943f720a-8b1d-4f31
-873c-e511f47d9f19
[hdfs@dn01 with_session]$
```

```
spark

spark_two=spark.newSession()

val spark_two = spark.newSession()

spark_two
```

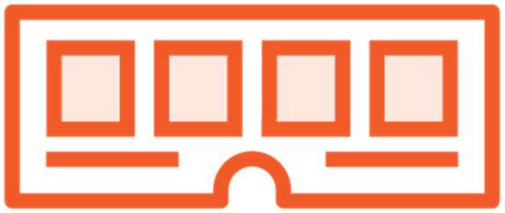
Multiple SparkSession Objects

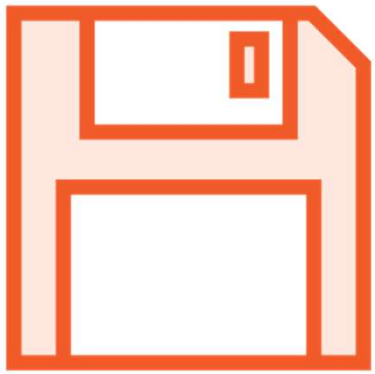
Possible to have multiple SparkSession objects

Independent SQLConf, UDFs and registered temporary views

- Shared SparkContext and table cache

Creating & Loading DataFrames





```
val qa_listDF =
 spark.createDataFrame(Array((1,"Xavier"),(2,"Irene"),(3,"Xavier")))
```

Creating a DataFrame Manually

Remember parallelize()? Similar idea

Use createDataFrame() and pass a list or array

And we have created a DataFrame

```
qa_listDF.collect()
sc.parallelize([(1,'Xavier'),(2,'Irene',(3,'Xavier')])
.collect()
```

Returning Data to the Driver

You can still call collect()

Worth noting the Row() objects!

Compare with RDD

```
qa_listDF.collect()
qa_listDF.take(3)
qa_listDF.show()
qa_listDF.show(1)
```

.collect() → [Row(_1=1, _2=u'Xavier')]

Collect does not output very nicely. Is that all we have?

Use show() instead

Nice formatting, with a few a couple of available parameters

```
qa_listDF.limit(1)
qa_listDF.limit(1).show()
qa_listDF.head()
qa_listDF.first()
qa_listDF.take(1)
qa_listDF.sample(false, .3, 42).collect()
```

More Options for Returning Data to the Driver

Let's check using tab completion

Test some of the available functionality

i.e. limit(), head(), take(), first() , sample() ...

Something is Still Bugging Me

org.apache.spark.sql.DataFrame = [_1: int, _2: string]

```
qa_listDF.show()
val qaDF = qa_listDF.toDF("Id", "QA")
qaDF.show()
```

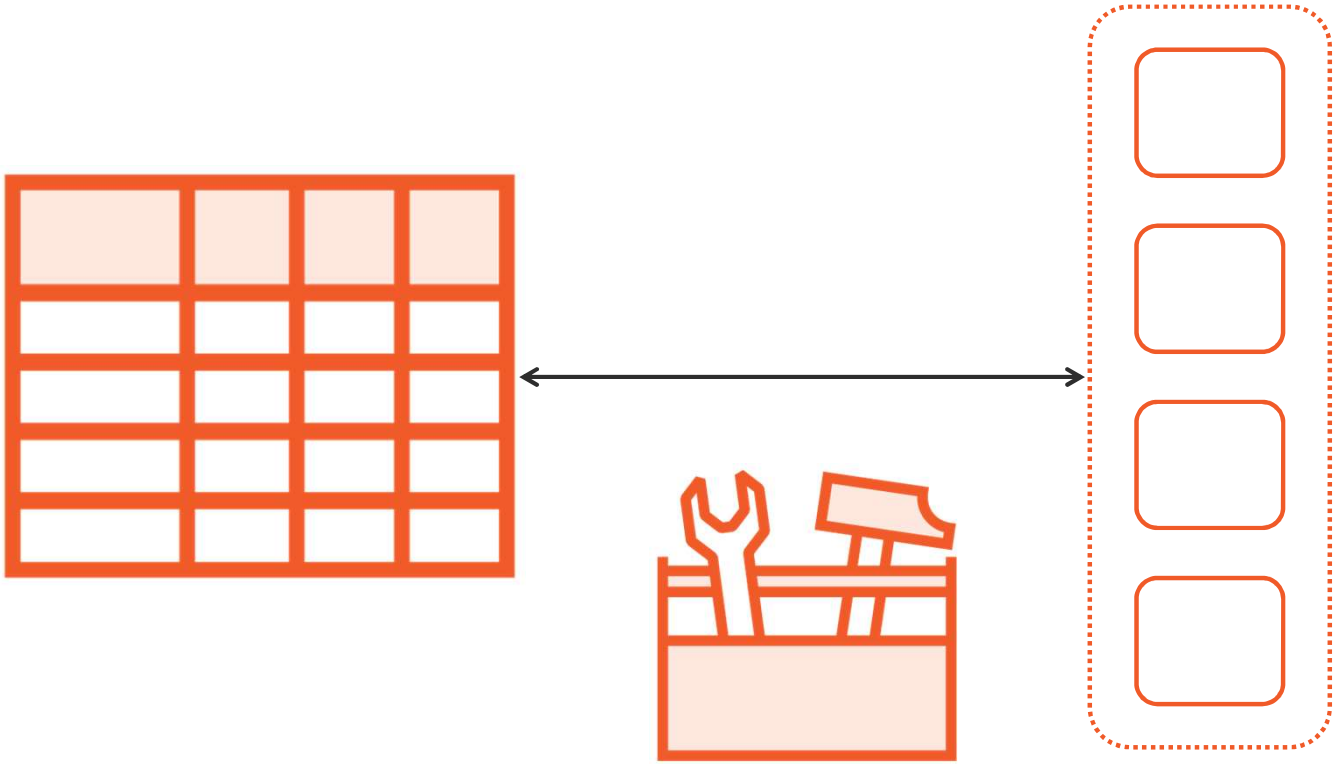
Nicer Column Names

Create a new DataFrame

But with nicer column names!

Use toDF()

DataFrames to RDDs & Viceversa



```
val qa_rdd = sc.parallelize(Array((1,"Xavier"), (2,"Irene"),
  (3,"Xavier")))

val qa_with_ToDF = qa_rdd.toDF()

val qa_with_create = spark.createDataFrame(qa_rdd)

qa_rdd.collect()

qa_with_ToDF.show()

qa_with_ToDF.rdd.collect()
```

DataFrames to RDDs & Viceversa

Create an RDD

Use toDF() on RDD to get a DataFrame

Use rdd on DataFrame to get an RDD

Prerequisite

```
cd badges
sbt package
spark2-submit --class "PrepareBadgesCSVApp"
              target/scala-2.11/badges-project_2.11-1.0.jar
```

Badges Data *(loaded in a previous module)*

Data preparation step

```
val badges_from_rddDF = badges_columns_rdd.toDF()
badges_from_rddDF.show(5)
val badges_tuple = badges_columns_rdd.map(x => (x(0), x(1),
  x(2), x(3), x(4), x(5))).toDF()
badges_tuple.show(5)
badges_from_rddDF.printSchema()
badges_tuple.printSchema()
```

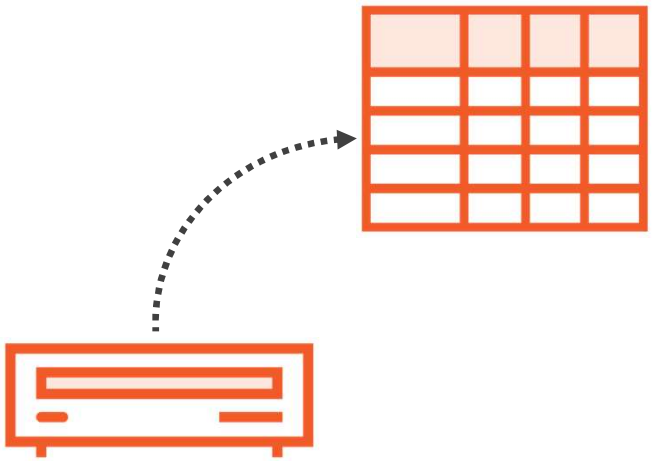
DataFrames

Use our StackOverflow / StackExchange RDDs

Badges data

Schema territory

Loading DataFrames

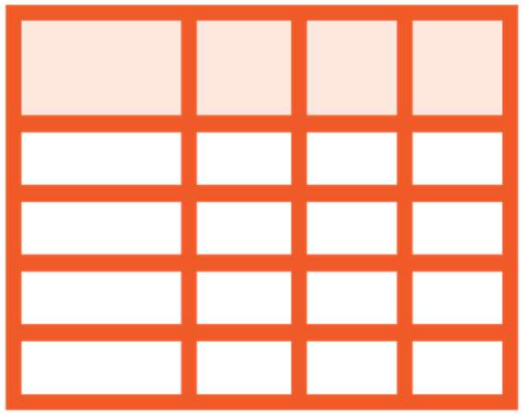


Data stored somewhere

Want to load it into a DataFrame

How do you do it?

DataFrameReader



Load data from external data sources

- Natively or using connectors

Supports multiple data formats

- Native support
- Define custom file formats

```
val posts_no_schemaTxtDF =
  spark.read.format("text").load("/user/cloudera/
  stackexchange/posts_all_csv")

posts_no_schemaTxtDF.printSchema()

posts_no_schemaTxtDF.show(5)

posts_no_schemaTxtDF.show(5, truncate=false)
```

Loading DataFrames

Use DataFrameReader, with SparkSession.read

- You specify format() and load()
- Let's start with text

Set truncate=false to get all text

```
val posts_no_schemaTxtDF =
  spark.read.text("/user/cloudera/
  stackexchange/posts_all_csv")

posts_no_schemaTxtDF.printSchema()

posts_no_schemaTxtDF.show(2)

posts_no_schemaTxtDF.show(2, truncate=false)
```

Specifying Format

We explicitly did with read.format("text").load()

- There is a quicker and more intuitive way

Use text()

```
val posts_no_schemaCSV = spark.read.csv("/user/cloudera/
  stackexchange/posts_all_csv")

posts_no_schemaCSV.show(5)

posts_no_schemaCSV.printSchema()
```

DataFrameReader Format: CSV

Specify a more structured format, in this case csv

Spark understands that we have columnar data

```
def csv(paths: String*): DataFrame
```

Loads CSV files and returns the result as a `DataFrame`.

This function will go through the input once to determine the input schema if `inferSchema` is enabled. To avoid going through the entire data once, disable `inferSchema` option or specify the schema explicitly using `schema`.

You can set the following CSV-specific options to deal with CSV files:

- `sep` (default `,`): sets a single character as a separator for each field and value.
- `encoding` (default `UTF-8`): decodes the CSV files by the given encoding type.
- `quote` (default `"`): sets a single character used for escaping quoted values where the separator can be part of the value. If you would like to turn off quotations, you need to set not `null` but an empty string. This behaviour is different from `com.databricks.spark.csv`.
- `escape` (default `\`): sets a single character used for escaping quotes inside an already quoted value.
- `charToEscapeQuoteEscaping` (default `escape` or `\0`): sets a single character used for escaping the escape for the quote character. The default value is escape character when escape and quote characters are different, `\0` otherwise.
- `comment` (default empty string): sets a single character used for skipping lines beginning with this character. By default, it is disabled.
- `header` (default `false`): uses the first line as names of columns.
- `inferSchema` (default `false`): infers the input schema automatically from data. It requires one extra pass over the data.
- `ignoreLeadingWhiteSpace` (default `false`): a flag indicating whether or not leading whitespaces from values being read should be skipped.
- `ignoreTrailingWhiteSpace` (default `false`): a flag indicating whether or not trailing whitespaces from values being read should be skipped.
- `nullValue` (default empty string): sets the string representation of a null value. Since 2.0.1, this applies to all supported types including the string type.
- `nanValue` (default `NaN`): sets the string representation of a non-number" value.
- `positiveInf` (default `Inf`): sets the string representation of a positive infinity value.
- `negativeInf` (default `-Inf`): sets the string representation of a negative infinity value.
- `dateFormat` (default `yyyy-MM-dd`): sets the string that indicates a date format. Custom date formats follow the formats at `java.text.SimpleDateFormat`. This applies to date type.
- `timestampFormat` (default `yyyy-MM-dd'T'HH:mm:ss.SSSXXX`): sets the string that indicates a timestamp format. Custom date formats follow the formats at `java.text.SimpleDateFormat`. This applies to timestamp type.
- `maxColumns` (default `20480`): defines a hard limit of how many columns a record can have.
- `maxCharsPerColumn` (default `-1`): defines the maximum number of characters allowed for any given value being read. By default, it is -1 meaning unlimited length
- `mode` (default `PERMISSIVE`): allows a mode for dealing with corrupt records during parsing. It supports the following case-insensitive modes.
- `PERMISSIVE` : sets other fields to `null` when it meets a corrupted record, and puts the malformed string into a field configured by `columnNameOfCorruptRecord`. To keep corrupt records, an user can set a string type field named `columnNameOfCorruptRecord` in an user-defined schema. If a schema does not have the field, it drops corrupt records during parsing. When a length of parsed CSV tokens is shorter than an expected length of a schema, it sets `null` for extra fields.
- `DROPMALFORMED` : ignores the whole corrupted records.
- `FAILFAST` : throws an exception when it meets corrupted records.
- `columnNameOfCorruptRecord` (default is the value specified in `spark.sql.columnNameOfCorruptRecord`): allows renaming the new field having malformed string created by `PERMISSIVE` mode. This overrides `spark.sql.columnNameOfCorruptRecord`.
- `multiLine` (default `false`): parse one record, which may span multiple lines.

Annotations	@varargs()
Since	2.0.0

```
def csv(csvDataset: Dataset[String]): DataFrame
```

Loads an `Dataset[String]` storing CSV rows and returns the result as a `DataFrame`.

```
def csv(path: String): DataFrame
```

```
postsNoSchemaDF=spark.read
.csv('/user/cloudera/stackexchange/posts_all_csv')

postsNoSchemaDF.printSchema

postsNoSchemaDF.printSchema()

postsNoSchemaDF.show()
```

DataFrameReader Format: CSV

Specify a more structured format, in this case csv

Spark understands that we have columnar data

Use printSchema()

```
root
 |-- _c0: string (nullable = true)
 |-- _c1: string (nullable = true)
 |-- _c2: string (nullable = true)
 |-- _c3: string (nullable = true)
 |-- _c4: string (nullable = true)
 |-- _c5: string (nullable = true)
 |-- _c6: string (nullable = true)
 |-- _c7: string (nullable = true)
 |-- _c8: string (nullable = true)
 |-- _c9: string (nullable = true)
 |-- _c10: string (nullable = true)
 |-- _c11: string (nullable = true)
 |-- _c12: string (nullable = true)
 |-- _c13: string (nullable = true)
 |-- _c14: string (nullable = true)
```

- All columns as string
- Column names?
- But columns none-the-less!

```
val posts_inferred =
spark.read.option("inferSchema","true").csv("/user/cloudera
/stackexchange/posts_all_csv")

posts_inferred.printSchema()
```

Infer Schema

Spark can infer the schema

Specify inferSchema and set to true

```
root
 |-- _c0: integer (nullable = true)
 |-- _c1: integer (nullable = true)
 |-- _c2: integer (nullable = true)
 |-- _c3: timestamp (nullable = true)
 |-- _c4: integer (nullable = true)
 |-- _c5: integer (nullable = true)
 |-- _c6: integer (nullable = true)
 |-- _c7: integer (nullable = true)
 |-- _c8: timestamp (nullable = true)
 |-- _c9: string (nullable = true)
```

- I see integer
- I see timestamp
- I see string
- I like inferred types
- I want to celebrate!

Remember?

Spark is Lazy

Warning!

With DataFrames it
reads file on load

```
val an_rdd = sc.textFile("/thisdoesnotexist")
val a_df = spark.read.text("/thisdoesnotexist")
```

Lazy Reading of Data(?)

RDD is lazy

DataFrames read ahead

```
spark.read.csv("/user/cloudera/stackexchange/posts_all_csv").printSchema

spark.read.option("inferSchema","true").csv("/user/cloudera/stackexchange/
  posts_all_csv").printSchema

spark.read.option("inferSchema","true").option("sep","|")
  .csv("/user/cloudera/stackexchange/posts_all_csv").printSchema

spark.read.options(Map("inferSchema"->"true",
  "sep"->"|")).csv("/user/cloudera/stackexchange/posts_all_csv").printSchema
```

Option

Use option() to pass parameters to DataFrameReader

Chain multiple option()

Or use options()

Prerequisite

```
cd posts_header

sbt package

spark2-submit --class "PreparePostsHeaderCSVApp"
              target/scala-2.11/posts-project_2.11-1.0.jar
```

Prepare Posts.xml as CSV with Headers

Data preparation step

```
posts_inferred.printSchema()

val posts_headersDF = spark.read.option("inferSchema",
"true").option("header",
true).csv("/user/cloudera/stackexchange/posts_all_csv_with_
header")

posts_headersDF.printSchema

posts_headersDF.show(5)
```

Column Names

Get column names from data

Takes first row as column name

```
root
 |-- Id: integer
 |-- PostTypeId: integer
 |-- AcceptedAnswerId: integer
 |-- CreationDate: timestamp
 |-- Score: integer
 |-- ViewCount: integer
 |-- OwnerUserId: integer
 |-- LastEditorUserId: integer
 |-- LastEditDate: timestamp
 |-- Title: string
 |-- LastActivityDate: timestamp
 |-- Tags: string
 |-- AnswerCount: integer
 |-- CommentCount: integer
 |-- FavoriteCount: integer
```

- Now this looks a lot better
- Wasn't this simple?
- However, not always 100% right
- Case of inferred incorrectly
- Corrupt data
- What if we wanted different column names?
- Or use different types?

```
import org.apache.spark.sql.types._

val postsSchema =
  StructType(Array(
  StructField("Id", IntegerType),
  StructField("PostTypeId", IntegerType), …
```

Explicit Schemas

Import types

StructType holds the schema

StructField is each field

Specify Schema

```
import org.apache.spark.sql.types._

val postsSchema =
StructType(Array(
StructField("Id", IntegerType),
StructField("PostTypeId", IntegerType),
StructField("AcceptedAnswerId", IntegerType),
StructField("CreationDate", TimestampType),
StructField("Score", IntegerType),
StructField("ViewCount", IntegerType),
StructField("OwnerUserId", IntegerType),
StructField("LastEditorUserId", IntegerType),
StructField("LastEditDate", TimestampType),
StructField("Title", StringType),
StructField("LastActivityDate", TimestampType),
StructField("Tags", StringType),
StructField("AnswerCount", IntegerType),
StructField("CommentCount", IntegerType),
StructField("FavoriteCount", IntegerType)))
```

```
val postsDF = spark.read.schema(postsSchema)
  .csv("/user/cloudera/stackexchange/posts_all_csv")

postsDF.printSchema()

postsDF.schema

postsDF.dtypes

postsDF.columns
```

Provide the Schema

Use schema()

Schema according to your specifications

```
default_formatDF = spark.read
  .load('/user/cloudera/stackexchange/posts_all_csv')
```

Quick Quiz

Which one do you think is the default format with DataFrames?

- Let's test

Parquet is assumed as default file format

Parquet

Columnar File Format

- Efficient

Supported by many Big Data systems

- Spark, MapReduce, Hive, Pig, Impala, ...

Default for higher level API

Parquet

Preserves schema of original data

Optimizes data storage

- Increasing performance
- Especially on large amounts of data

Cloudera and Twitter → Apache

Prerequisite

```
cd comments

sbt package

spark2-submit --class "PrepareCommentsParquetApp"
              target/scala-2.11/comments-project_2.11-1.0.jar
```

Convert Comments.xml to Parquet

Data preparation step

```
val comments_parquetDF = spark.read.parquet("/user/cloudera
  /stackexchange/comments_parquet")

comments_parquetDF.printSchema()

comments_parquetDF.show(5)
```

Loading Parquet

Use parquet()

Schema preserved, no need to specify inferSchema

Better than text

```
cd tags
sbt package
spark2-submit --class "PrepareTagsJSON"
              target/scala-2.11/tags-project_2.11-1.0.jar
```

Convert Tags.xml to JSON

Data preparation step

```
val tags_jsonDF =
  spark.read.json("/user/cloudera/stackexchange/tags_json")

tags_jsonDF.printSchema()

comments_parquetDF.show(5)
```

Loading JSON

Load JSON into DataFrame, with json()

- Schema inferred

JSON file vs. JSON rows in a file (JSON Lines file format)

JSON Lines

Documentation for the JSON Lines text file format

Home Examples On the web json.org

This page describes the JSON Lines text format, also called newline-delimited JSON. JSON Lines is a convenient format for storing structured data that may be processed one record at a time. It works well with unix-style text processing tools and shell pipelines. It's a great format for log files. It's also a flexible format for passing messages between cooperating processes.

The JSON Lines format has three requirements:

1. UTF-8 Encoding

JSON allows encoding Unicode strings with only ASCII escape sequences, however those escapes will be hard to read when viewed in a text editor. The author of the JSON Lines file may choose to escape characters to work with plain ASCII files.

Encodings other than UTF-8 are very unlikely to be valid when decoded as UTF-8 so the chance of accidentally misinterpreting characters in JSON Lines files is low.

2. Each Line is a Valid JSON Value

The most common values will be objects or arrays, but any JSON value is permitted.

See json.org for more information about JSON values.

Query | Search data and saved documents... | Jobs 1 | hdfs

File Browser

View as binary

Edit file

Download

View file location

Refresh

Last modified
01/11/2018 2:32 PM

User
hdfs

Group
supergroup

Size
22.07 KB

Mode
100644

Home

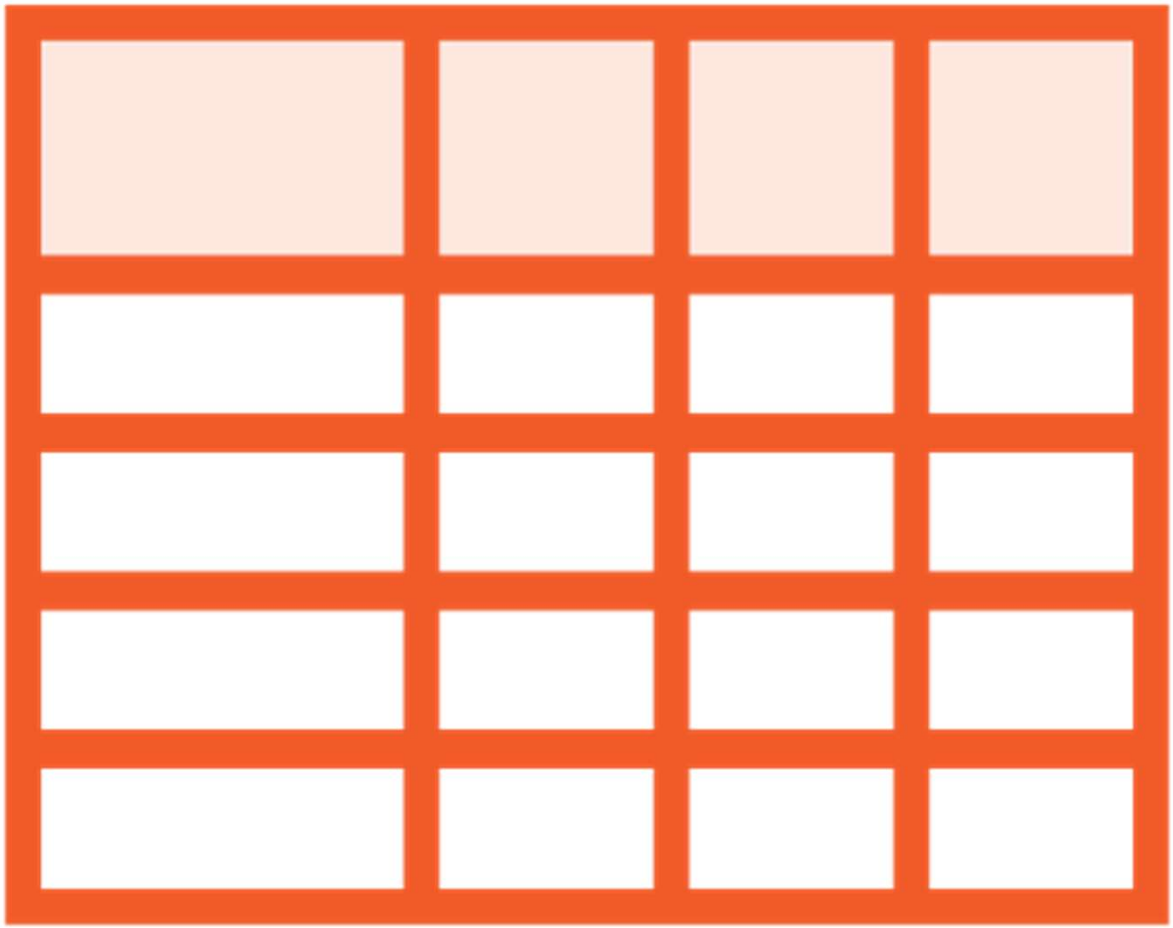
Page 1 to 6 of 6

/ user / cloudera / stackexchange / tags_json /
part-00000-c55fa897-0cea-442e-859d-123d18a64497-c000.json

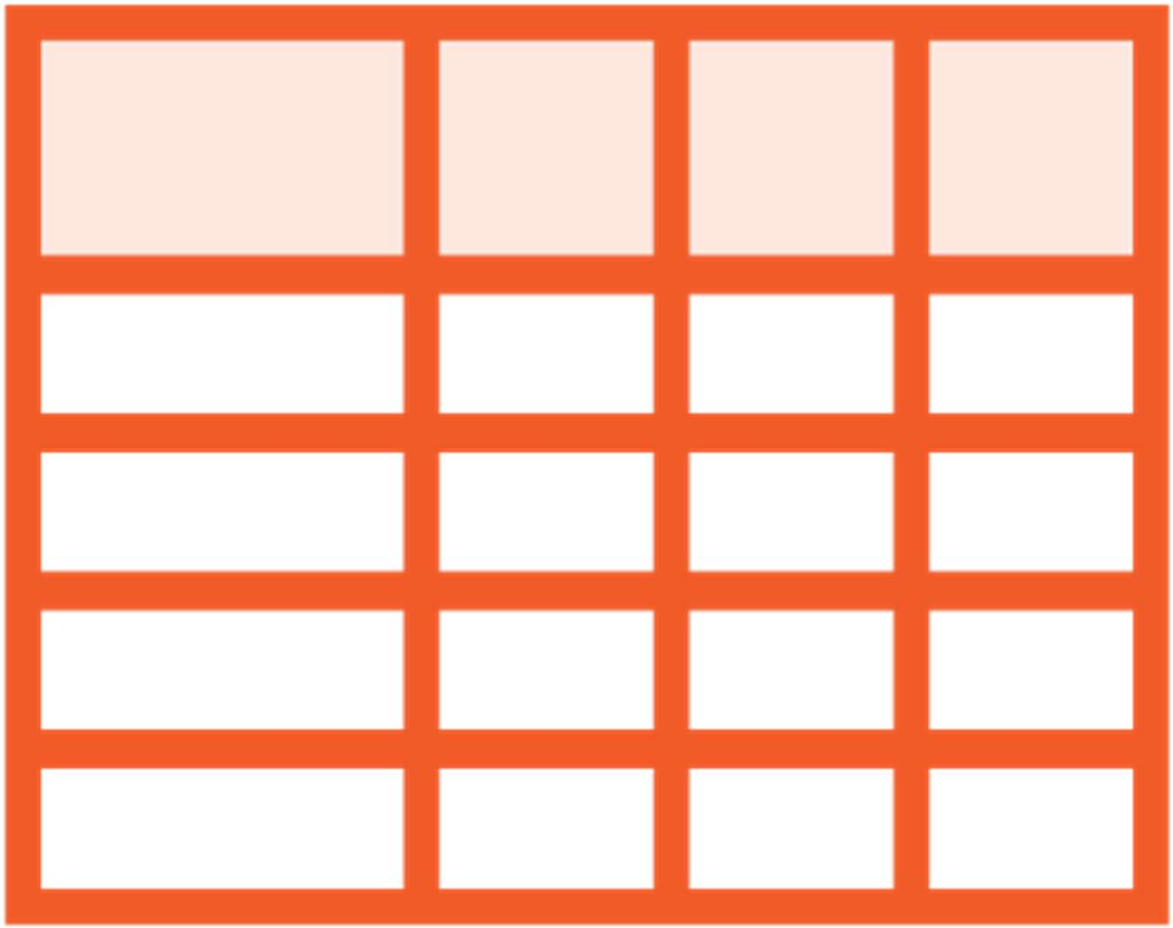

```
{"Id":"1","TagName":"line-numbers","Count":"33","ExcerptPostId":"4450","WikiPostId":"4449"}
{"Id":"2","TagName":"indentation","Count":"166","ExcerptPostId":"3239","WikiPostId":"3238"}
{"Id":"6","TagName":"macro","Count":"68","ExcerptPostId":"856","WikiPostId":"855"}
{"Id":"7","TagName":"text-generation","Count":"23","ExcerptPostId":"6549","WikiPostId":"6548"}
{"Id":"12","TagName":"search","Count":"198","ExcerptPostId":"4216","WikiPostId":"4215"}
{"Id":"18","TagName":"cursor-movement","Count":"153","ExcerptPostId":"4214","WikiPostId":"4213"}
{"Id":"19","TagName":"vimrc","Count":"514","ExcerptPostId":"316","WikiPostId":"315"}
{"Id":"22","TagName":"syntax-highlighting","Count":"221","ExcerptPostId":"2070","WikiPostId":"2069"}
{"Id":"23","TagName":"neovim","Count":"156","ExcerptPostId":"597","WikiPostId":"596"}
{"Id":"24","TagName":"folding","Count":"86","ExcerptPostId":"2112","WikiPostId":"2111"}
{"Id":"27","TagName":"filesystem","Count":"50","ExcerptPostId":"2040","WikiPostId":"2039"}
{"Id":"28","TagName":"filetype","Count":"78","ExcerptPostId":"4321","WikiPostId":"4320"}
{"Id":"30","TagName":"split","Count":"74","ExcerptPostId":"828","WikiPostId":"827"}
{"Id":"32","TagName":"save","Count":"56","ExcerptPostId":"1956","WikiPostId":"1955"}
{"Id":"34","TagName":"buffers","Count":"151","ExcerptPostId":"643","WikiPostId":"642"}
{"Id":"35","TagName":"crash-recovery","Count":"6"}
{"Id":"37","TagName":"autocompletion","Count":"153","ExcerptPostId":"2154","WikiPostId":"2153"}
{"Id":"40","TagName":"vimscript","Count":"487","ExcerptPostId":"272","WikiPostId":"271"}
{"Id":"43","TagName":"large-documents","Count":"6"}
{"Id":"44","TagName":"count","Count":"10","ExcerptPostId":"2712","WikiPostId":"2711"}
{"Id":"45","TagName":"abbreviations","Count":"30","ExcerptPostId":"4461","WikiPostId":"4460"}
{"Id":"46","TagName":"wrapping","Count":"48","ExcerptPostId":"4329","WikiPostId":"4328"}
{"Id":"48","TagName":"cut-copy-paste","Count":"164","ExcerptPostId":"817","WikiPostId":"816"}
```

So Far We Have Loaded

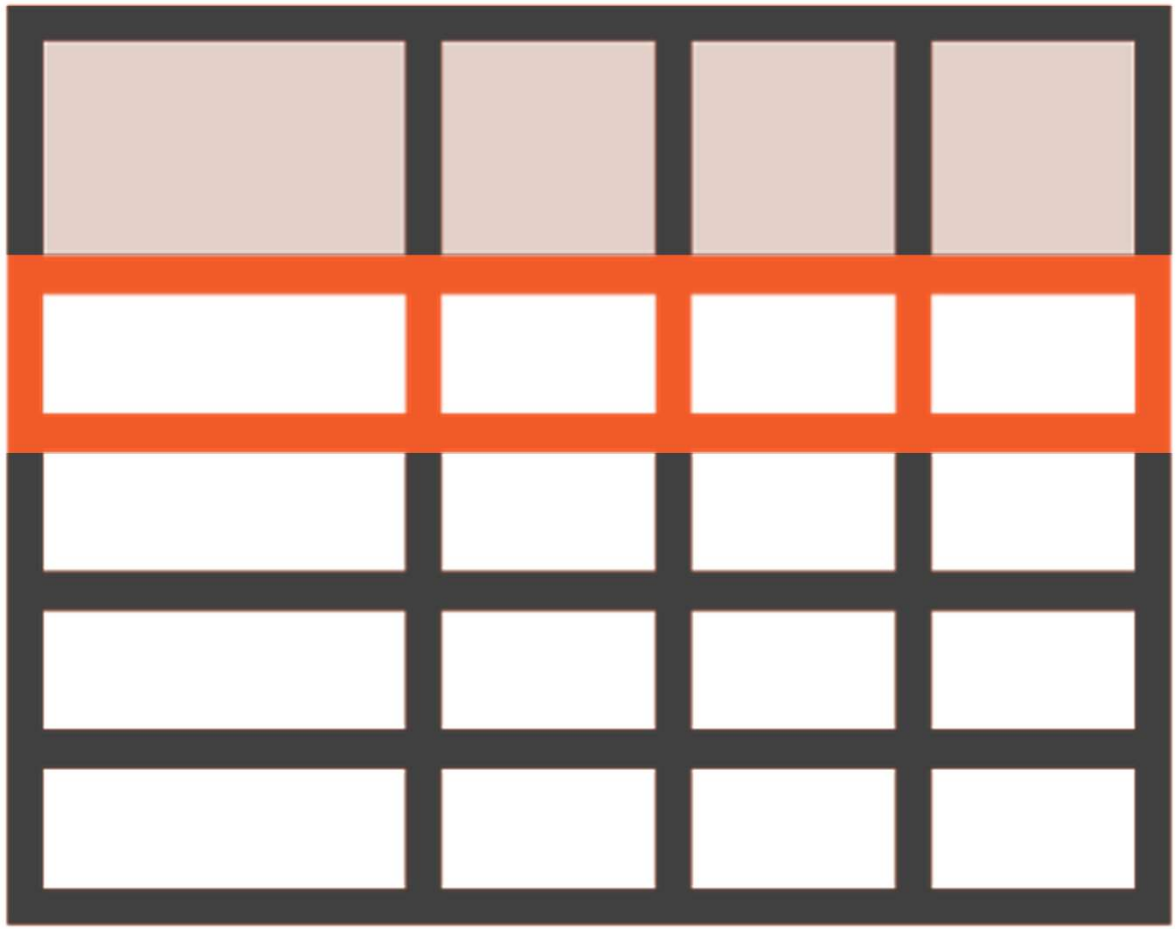
Rows, Columns, Expressions and Operators



Rows



Rows

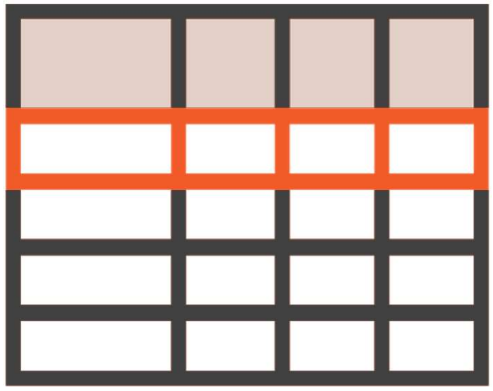


```
postsDF.take(1)

postsDF.show(1)

tags_jsonDF.show(5)

postsDF.columns
```

Row

Inspect one row

Column names with their values



Columns, Column Expressions and Column Operators

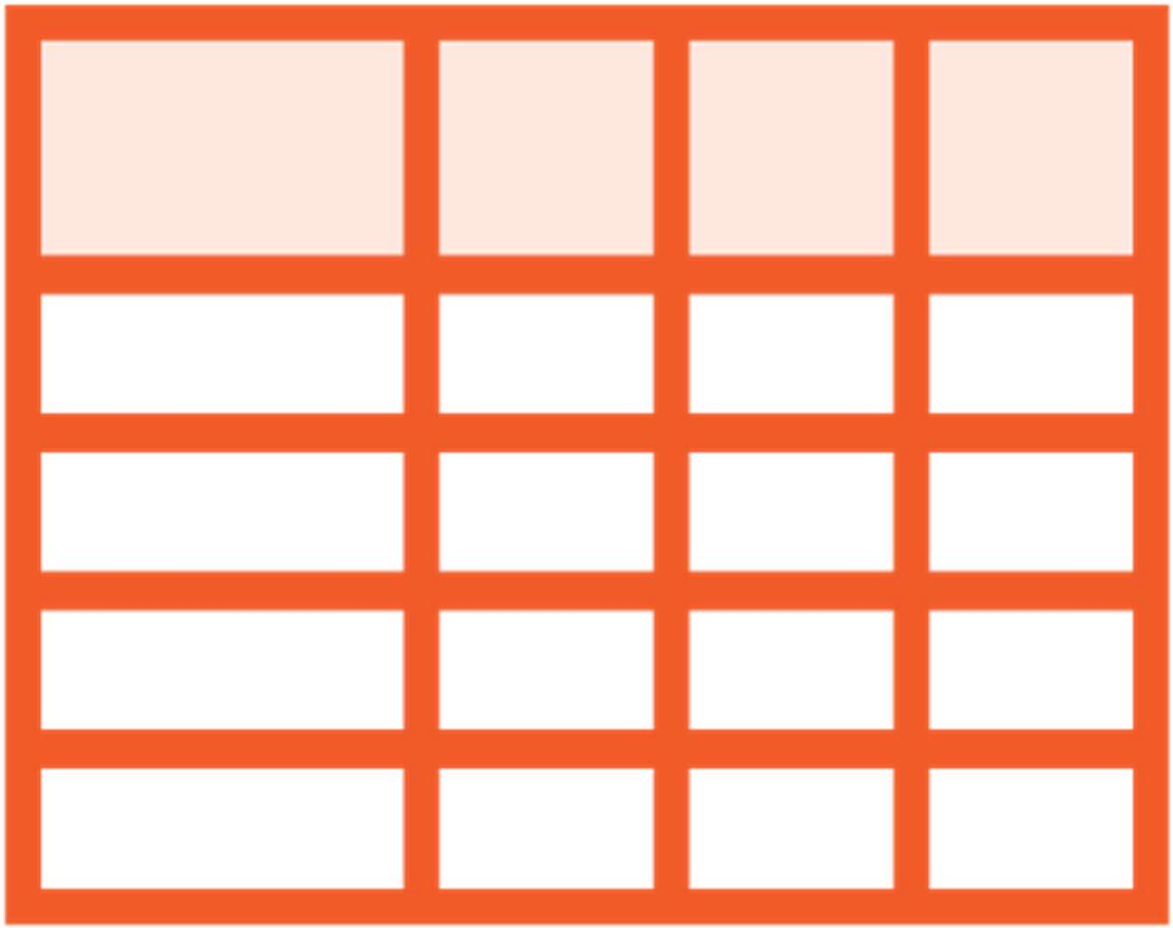
DataFrame

- Named columns

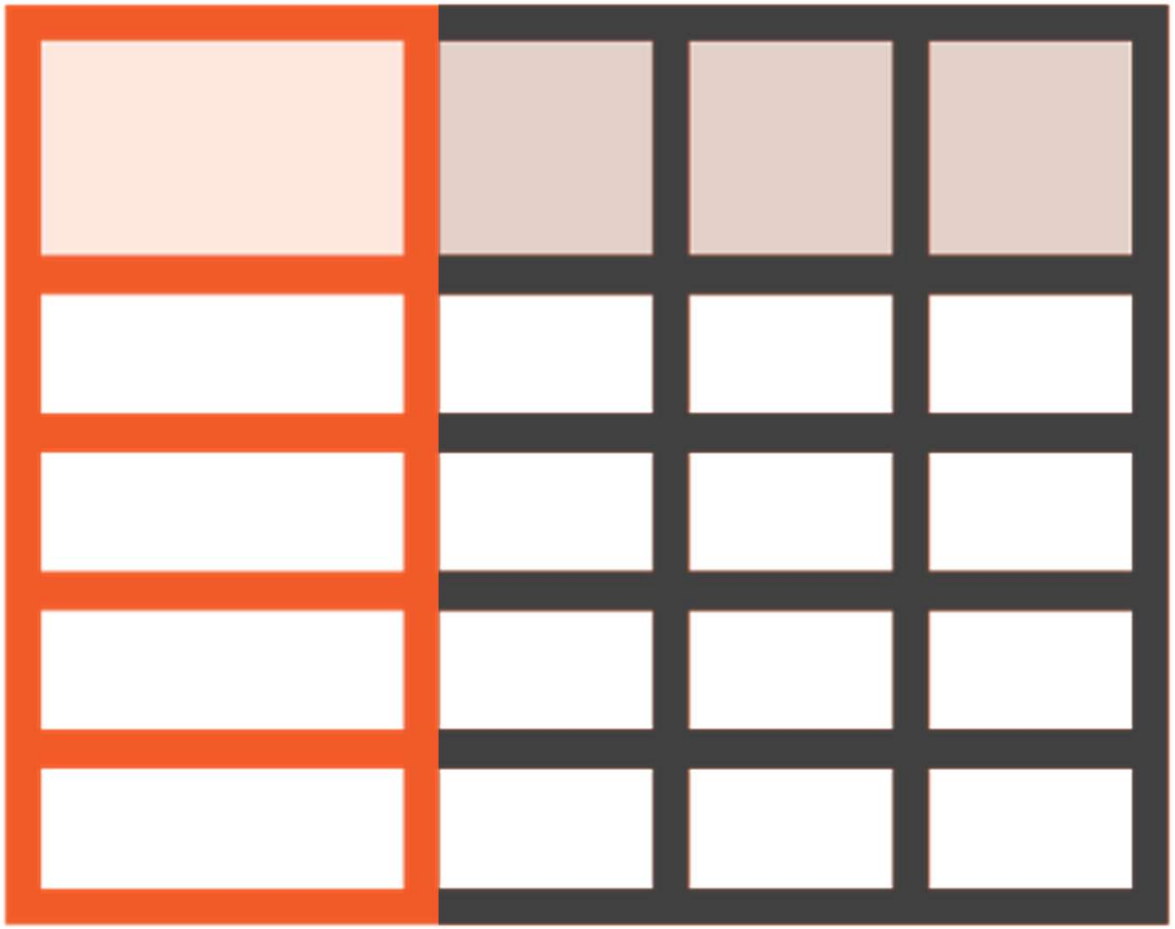
Does not simply store values

- What then?

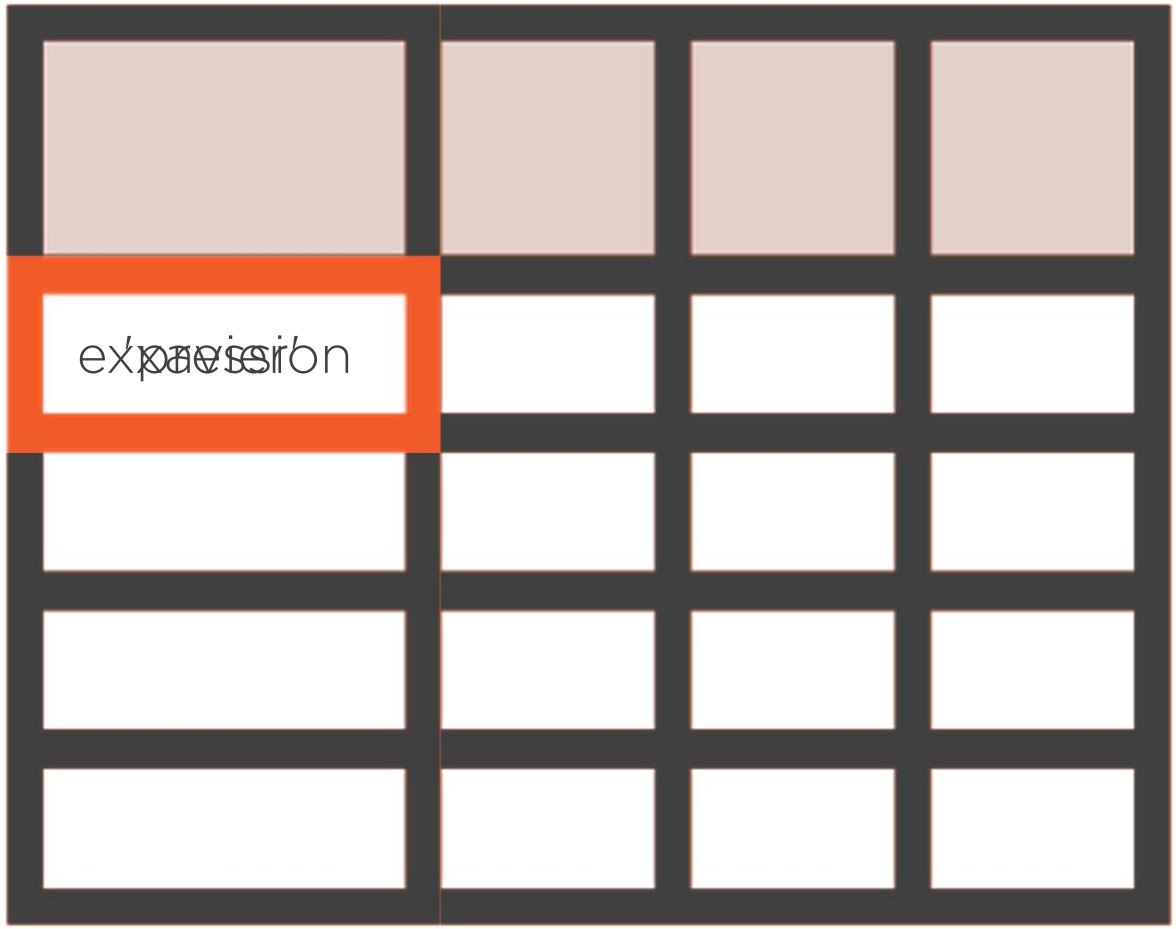
DataFrame



Columns



Columns



Expression

Catalyst expression

Produces a value per row

Based on a value, function or operation

Referring to Columns

Canonical

```
scala> postsDF("Title")
res248: org.apache.spark.sql.Column = Title

scala>
```

Just like a Map

Using parenthesis

Shortcut Syntax

```
scala> $"Title"
res249: org.apache.spark.sql.ColumnName = Title

scala>
```

Specify name of column with $

Not fully resolved until used in transformation

Can be ambiguous

```
col("Title")
```

```
postsDF.select(postsDF("Title"))
postsDF.select(postsDF("Title")).show(1)
postsDF.select($"Title").show(1)
postsDF.select(col("Title")).show(1)
postsDF.select($"Title", postsDF("Id"), col("CreationDate").show(1)
postsDF.select("Title", "Id").show(1)
postsDF.select("Title", $"CreationDate").show(1)
postsDF.select($"Score" * 1000).show(1)
postsDF.select("Score" * 1000).show(1)
```

Working with Columns

Specify which column or columns you want to return

Pass in a column expression, i.e. use a function or use some math

Check org.apache.spark.sql.functions for full list

Return true iff the column is NaN.

▶ `def isnull(e: Column): Column`

Return true iff the column is null.

▶ `def least(columnName: String, columnNames: String*): Column`

Returns the least value of the list of column names, skipping null values.

▶ `def least(exprs: Column*): Column`

Returns the least value of the list of values, skipping null values.

▼ `def lit(literal: Any): Column`

Creates a Column of literal value.

The passed in object is returned directly if it is already a Column. If the object is a Scala Symbol, it is converted into a Column also. Otherwise, a new Column is created to represent the literal value.

Since 1.3.0

▶ `def map(cols: Column*): Column`

Creates a new map column.

▶ `def monotonically_increasing_id(): Column`

A column expression that generates monotonically increasing 64-bit integers.

▶ `def nanvl(col1: Column, col2: Column): Column`

Returns col1 if it is not NaN, or col2 if col1 is NaN.

▶ `def negate(e: Column): Column`

Unary minus, i.e.

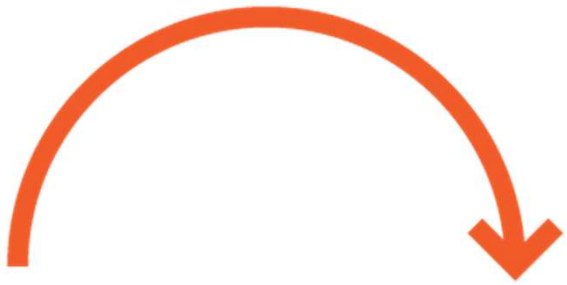
▶ `def not(e: Column): Column`

Inversion of boolean expression, i.e.

▶ `def rand(): Column`

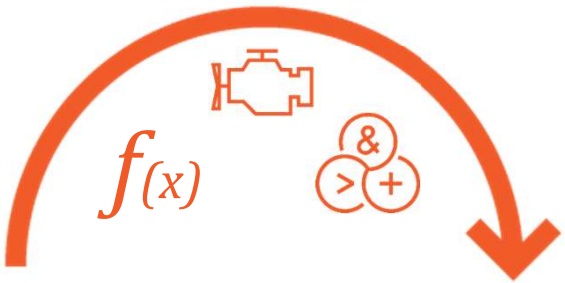
```
val postsSchema = StructType(Array(
  StructField("Id", IntegerType),
  StructField("PostTypeId", IntegerType),
  StructField("AcceptedAnswerId", IntegerType),
  StructField("CreationDate", TimestampType),
  StructField("Score", IntegerType),
  StructField("ViewCount", StringType),
  StructField("OwnerUserId", IntegerType),
  StructField("LastEditorUserId", IntegerType),
  StructField("LastEditDate", TimestampType),
  StructField("Title", StringType),
  StructField("LastActivityDate", TimestampType),
  StructField("Tags", StringType),
  StructField("AnswerCount", IntegerType),
  StructField("CommentCount", IntegerType),
  StructField("FavoriteCount", IntegerType)))
```

Column Expressions



```
ViewCount  StringType()
```

Column Expressions




```
ViewCount  IntegerType()
```

```
postsDF.dtypes
postsDF.dtypes.find(x => x._1 == "ViewCount")
postsDF.schema("ViewCount")
postsDF.select("ViewCount").printSchema()
val posts_viewDF = postsDF.withColumn("ViewCount",
  postsDF("ViewCount").cast("integer"))
posts_viewDF.select("ViewCount").printSchema()
posts_viewDF.select("ViewCount").show(5)
```

Casting Columns

Change type of a column

Use withColumn() and cast()

Besides casting, you can apply functions

```
val postsVCDF = postsDF.withColumnRenamed("ViewCount",
  "ViewCountStr")

postsVCDF.printSchema()

val posts_twoDF = postsDF.withColumnRenamed("ViewCount",
  "ViewCountStr").withColumnRenamed("Score", "ScoreInt")

posts_twoDF.printSchema()
```

Renaming Columns

Using withColumnRenamed()

Returns a new DataFrame

What if we wanted to rename two columns?

```
val posts_ticksDF = postsDF.withColumnRenamed("ViewCount",
  "ViewCount.Str")

posts_ticksDF.select("ViewCount.Str").show(3)

posts_ticksDF.select("`ViewCount.Str`").show()
```

A Thing or Two on Column Names

Use valid column names

i.e. dot can cause issue, means column path instead of column name

Use `backticks` to escape

```
val posts_wcDF = postsDF.
  withColumn("TitleClone1", $"Title")

posts_wcDF.printSchema()

postsDF.withColumn("Title", concat(lit("Title: "),
  $"Title")).select($"Title").show(5)
```

Copy Columns

Use withColumn()

Replace if name already exists

```
posts_wcDF.columns
posts_wcDF.columns.contains("TitleClone1")
val posts_no_cloneDF = posts_wcDF.drop("TitleClone1")
posts_no_cloneDF.columns.contains("TitleClone1")
posts_no_cloneDF.printSchema()
```

Dropping Columns

Remove columns from DataFrame

Use drop()

Computes the Levenshtein distance of the two given string columns.

▶ `def locate(substr: String, str: Column, pos: Int): Column`

Locate the position of the first occurrence of substr in a string column, after position pos.

▶ `def locate(substr: String, str: Column): Column`

Locate the position of the first occurrence of substr.

▼ `def lower(e: Column): Column`

Converts a string column to lower case.

Since 1.3.0

▶ `def lpad(str: Column, len: Int, pad: String): Column`

Left-pad the string column with pad to a length of len.

▶ `def ltrim(e: Column, trimString: String): Column`

Trim the specified character string from left end for the specified string column.

▶ `def ltrim(e: Column): Column`

Trim the spaces from left end for the specified string value.

▶ `def regexp_extract(e: Column, exp: String, groupIdx: Int): Column`

Extract a specific group matched by a Java regex, from the specified string column.

▶ `def regexp_replace(e: Column, pattern: Column, replacement: Column): Column`

Replace all substrings of the specified string value that match regexp with rep.

▶ `def regexp_replace(e: Column, pattern: String, replacement: String): Column`

Computes the Levenshtein distance of the two given string columns.

▶ `def locate(substr: String, str: Column, pos: Int): Column`

Locate the position of the first occurrence of substr in a string column, after position pos.

▶ `def locate(substr: String, str: Column): Column`

Locate the position of the first occurrence of substr.

▼ `def give_me_list(e: Column): Column`

Splits a string representing an array into an array.

Since	Created for Pluralsight's Spark + Scala on Cloudera course

---

▶ `def lpad(str: Column, len: Int, pad: String): Column`

Left-pad the string column with pad to a length of len.

▶ `def ltrim(e: Column, trimString: String): Column`

Trim the specified character string from left end for the specified string column.

▶ `def ltrim(e: Column): Column`

Trim the spaces from left end for the specified string value.

▶ `def regexp_extract(e: Column, exp: String, groupIdx: Int): Column`

Extract a specific group matched by a Java regex, from the specified string column.

▶ `def regexp_replace(e: Column, pattern: Column, replacement: Column): Column`

Replace all substrings of the specified string value that match regexp with rep.

▶ `def regexp_replace(e: Column, pattern: String, replacement: String): Column`

```
val questionsDF = postsDF.filter(col("PostTypeId") === 1)
questionsDF.select("Tags").show(20, truncate=false)
```

More Than org.apache.spark.sql.functions

There are many functions available

Sometimes you may need more

```
def give_me_list(str_lst: String): List[String] = {

if(str_lst == "()" || Option(str_lst).getOrElse("").isEmpty
|| str_lst.length < 2) return List[String]()

val elements = str_lst.slice(1, str_lst.length - 1)

elements.split(",").toList

}


val list_in_string = "(indentation,line-numbers)"

give_me_list(list_in_string)
```

```
import org.apache.spark.sql.functions.udf

val udf_give_me_list = udf(give_me_list _)
val questions_id_tagsDF =
questionsDF.withColumn("Tags",udf_give_me_list(
  questionsDF("Tags"))).select("Id", "Tags")
questions_id_tagsDF.printSchema()

questions_id_tagsDF.select("Tags").show(10, truncate=false)
```

User Defined Functions

Great because you can extend functionality

Create function and register the UDF

Cannot be optimized as Spark SQL functions, use only when needed

```
import org.apache.spark.sql.functions.explode

questions_id_tagsDF.select(explode(questions_id_tagsDF("Tags"))).
  show(10)

questions_id_tagsDF.select(explode(questions_id_tagsDF("Tags"))).count()

questions_id_tagsDF.select(explode(questions_id_tagsDF("Tags"))).
  distinct().count()
```

Distinct Tags

Function on column we just applied our UDF

With explode() you get one entry per item on the array

Takeaway

Earlier we learned about RDDs

Increase proficiency

- With DataFrames and Spark SQL

"Everyone knows SQL"

History of the higher level API

Takeaway

Create and load DataFrames

- From RDDs
- From Data in memory or storage

Many supported file formats

- Text, CSV, Parquet, JSON...
- Schema

Takeaway

- Schemas
  - Each column of particular type
  - Inferred or explicitly defined
- Columns
  - Instead of a value
  - Catalyst expressions
  - Column operations
- UDFs

More DataFrames and Spark SQL