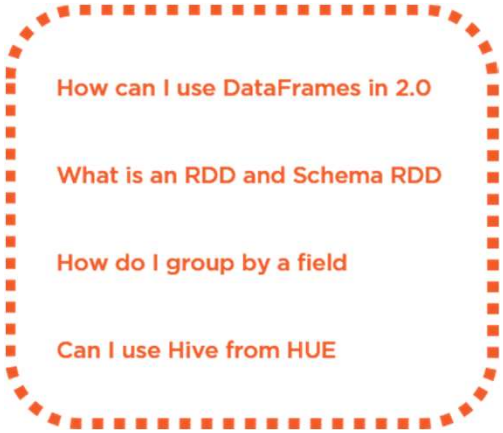
Increasing Proficiency with Spark: DataFrames & Spark SQL

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

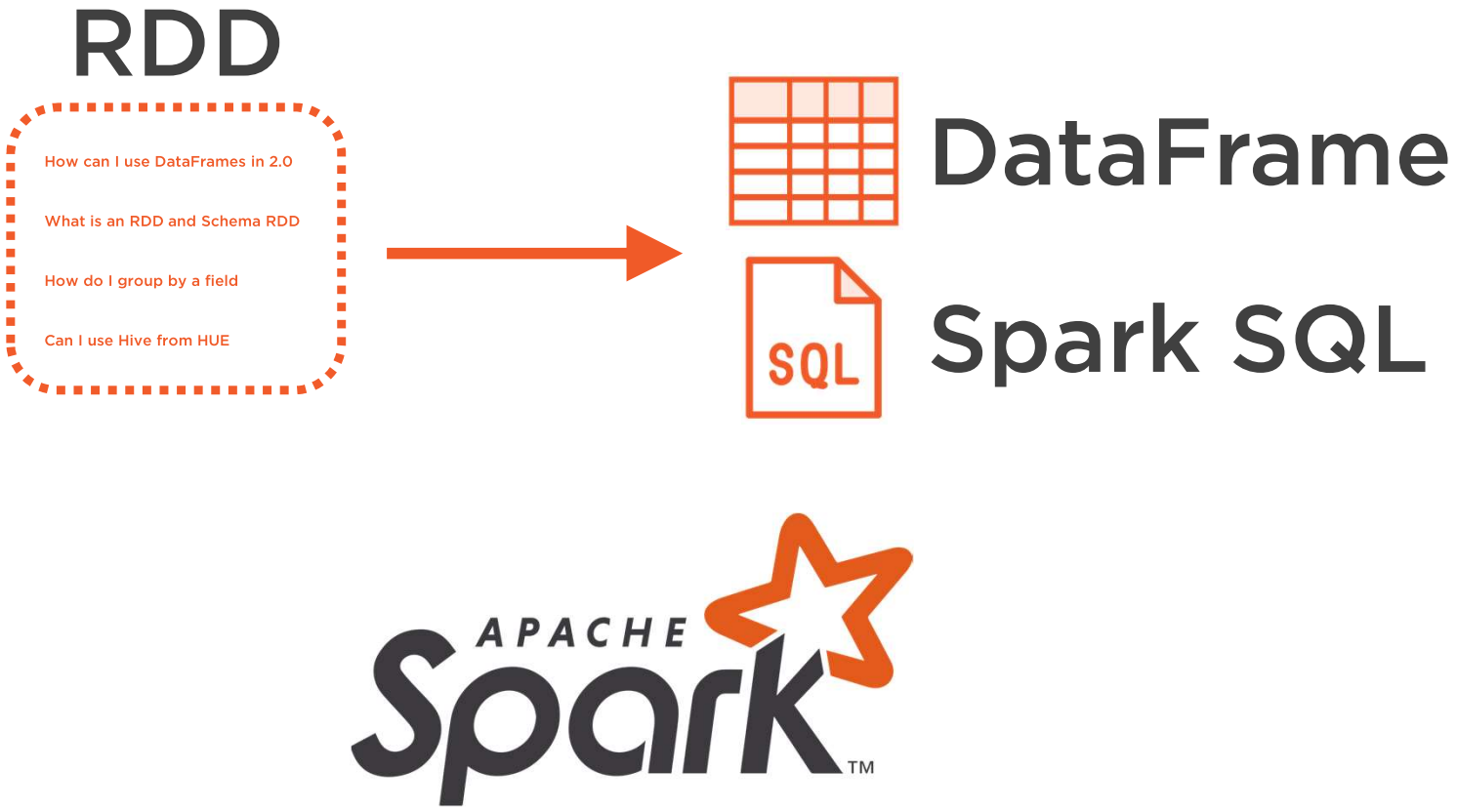
Supported by many applications

- Beyond databases

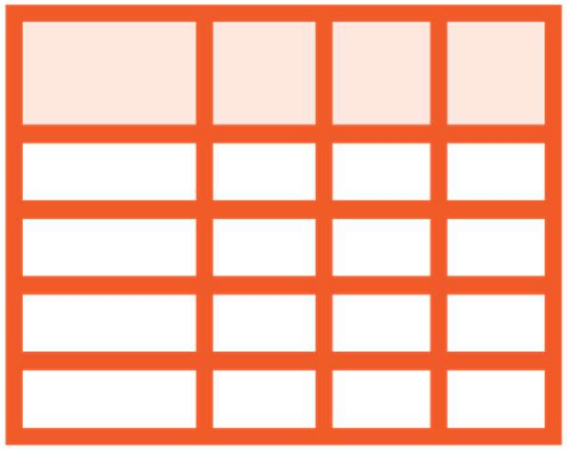
Early Days of Hadoop

Early Days of Hadoop

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

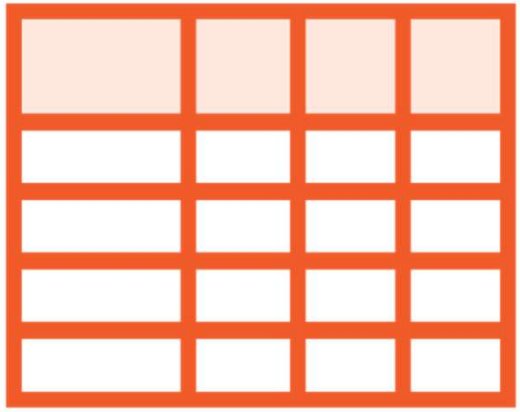
- SQL using Spark execution engine
- Evolved into Spark SQL in 1.0

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

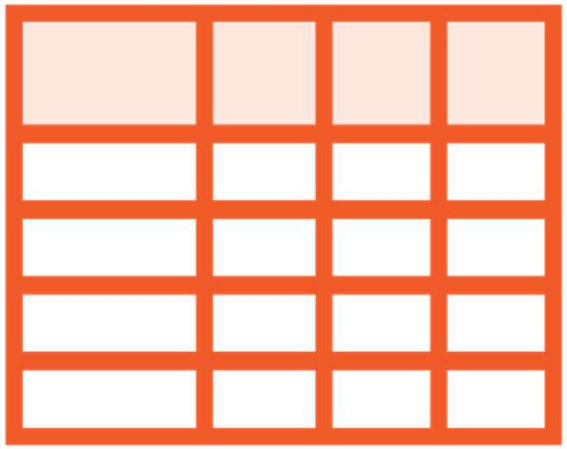
Interact via SQL queries

- ANSI SQL 2003 support

Works with Hive

Any data source compatible with Spark

DataFrame



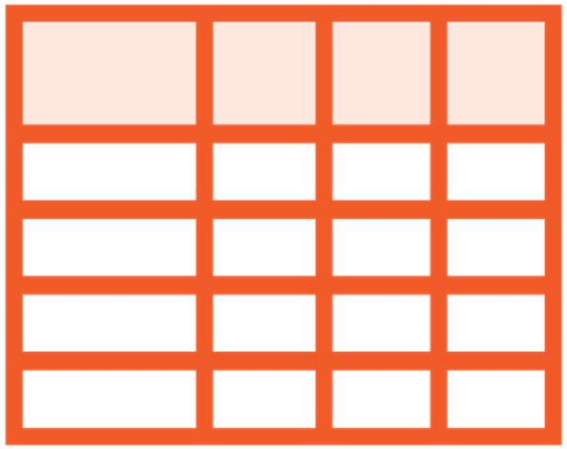
Distributed collection of Row objects

- Dataset[Row]

Equivalent to a database table

- Rows and columns,
- Known schema
- Or dataframe in Python

DataFrame



Structured or unstructured data

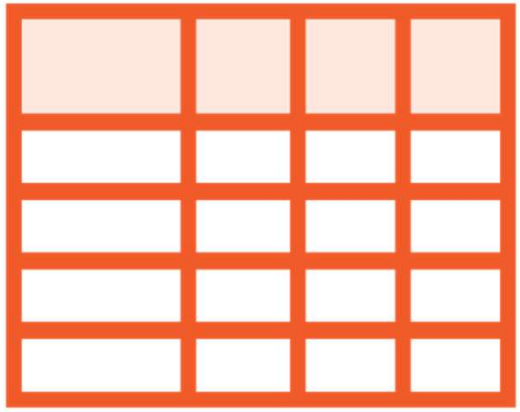
- Conversion to and from RDD possible

Queries

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

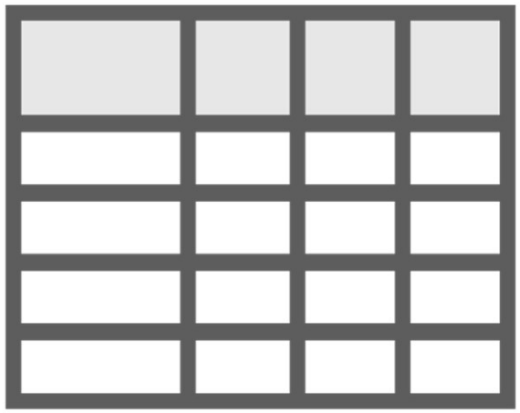
DataFrames & Spark SQL

Spark SQL



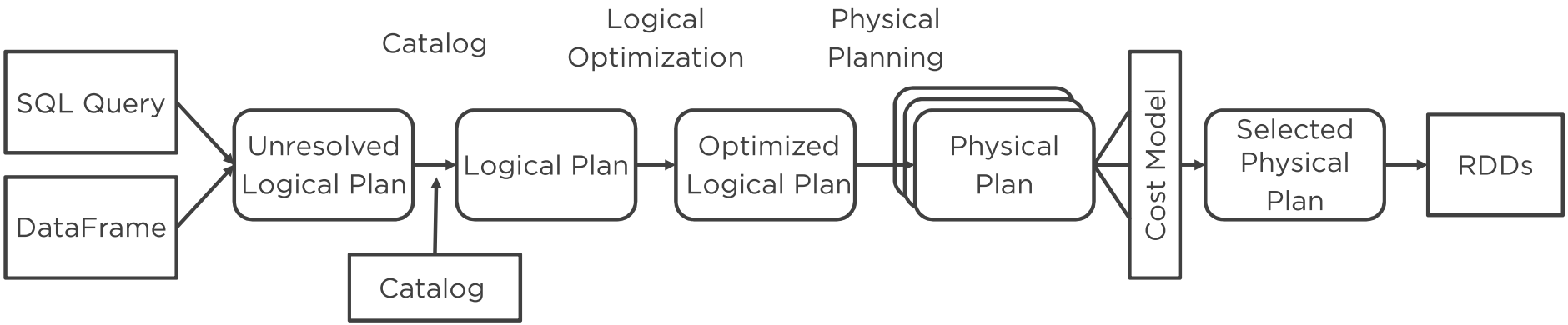
DataFrame

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...

```
spark=SparkSession.builder \
      .master('yarn') \
      .appName('StackOverflowTest') \
      .config('spark.executor.memory', '2g') \
      .getOrCreate()
```

SparkSession

Entry point to Spark SQL

Merges SQLContext, HiveContext

SparkContext

```
spark=SparkSession.builder \
      .master('yarn') \
      .appName('StackOverflowTest') \
      .config('spark.executor.memory', '2g') \
      .getOrCreate()
```

Getting a SparkSession

Created automatically for you in pyspark2 (REPL)

But you need to create it for self contained applications

```
-4003-8d81-c0ba8678fc1d/_tmp_space.db
18/01/18 18:08:25 INFO session.SessionState: No Tez session required at this point. hive.executio
n.engine=mr.
18/01/18 18:08:25 INFO client.HiveClientImpl: Warehouse location for Hive client (version 1.1.0)
is /user/hive/warehouses
18/01/18 18:08:25 INFO state.StateStoreCoordinatorRef: Registered StateStoreCoordinator endpoint
*** Testing simple_with_session.py ***
*** Application name: Simple with Session / Version: 2.2.0.cloudera1 ***
18/01/18 18:08:25 INFO server.AbstractConnector: Stopped Spark@10babd05{HTTP/1.1,[http/1.1]}{0.0.
0.0:4040}
18/01/18 18:08:25 INFO ui.SparkUI: Stopped Spark web UI at http://10.0.2.104:4040
18/01/18 18:08:25 INFO cluster.YarnClientSchedulerBackend: Interrupting monitor thread
18/01/18 18:08:25 INFO cluster.YarnClientSchedulerBackend: Shutting down all executors
18/01/18 18:08:25 INFO cluster.YarnSchedulerBackend$YarnDriverEndpoint: Asking each executor to s
hut down
18/01/18 18:08:25 INFO cluster.SchedulerExtensionServices: Stopping SchedulerExtensionServices
(serviceOption=None,
 services=List(),
 started=false)
18/01/18 18:08:25 INFO cluster.YarnClientSchedulerBackend: Stopped
18/01/18 18:08:25 INFO spark.MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopp
ed!
18/01/18 18:08:25 INFO memory.MemoryStore: MemoryStore cleared
18/01/18 18:08:25 INFO storage.BlockManager: BlockManager stopped
18/01/18 18:08:25 INFO storage.BlockManagerMaster: BlockManagerMaster stopped
18/01/18 18:08:25 INFO scheduler.OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputC
ommitCoordinator stopped!
18/01/18 18:08:25 INFO spark.SparkContext: Successfully stopped SparkContext
18/01/18 18:08:25 INFO util.ShutdownHookManager: Shutdown hook called
```

```
[hdfs@dn04 m7 - Increasing Proficiency with Spark - DataFrames and Spark SQL]$ \
> spark2-submit simple_no_session.py
Testing simple_no_session.py
Traceback (most recent call last):
  File "/spark-demos/m7 - Increasing Proficiency with Spark - DataFrames and Spark SQL/simple_no_
session.py", line 7, in <module>
    print "Application name: " + spark.sparkContext.appName + " / Version: " + spark.version
NameError: name 'spark' is not defined
[hdfs@dn04 m7 - Increasing Proficiency with Spark - DataFrames and Spark SQL]$ spark2-submit simp
le_no_session.py
```

```
spark
spark_two=spark.newSession()
spark_two
```

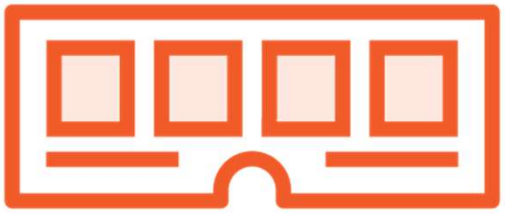
Multiple SparkSession Objects

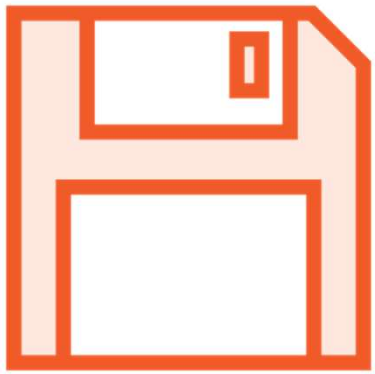
Possible to have multiple SparkSession objects

Independent SQLConf, UDFs and registered temporary views

- Shared SparkContext and table cache

Creating & Loading DataFrames





```
qa_listDF = spark.createDataFrame([(1,'Xavier'),(2,'Irene',(3,'Xavier')])
type(qa_listDF)
```

Creating a DataFrame Manually

Remember parallelize()? Similar idea

Use createDataFrame() and pass a list or pandas dataframe

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_dictDF=spark.createDataFrame({(1,'Xavier'),(2,'Irene'),(3,'Xavier')})
qa_dictDF.collect()
```

Create DataFrame with a Dictionary

Other objects can be used

Example is a dictionary

```
from pyspark.sql import Row

qa_from_row = spark.createDataFrame([Row(1,'Xavier'), Row(2,
'Irene'),(3,'Xavier')])

qa_from_row.collect()
```

Row Objects with CreateDataFrame

List of Row() objects

A Row represents a row of data in a DataFrame

Each Row object is a container, with additional attributes

```
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

```
dir(qa_listDF)
qa_listDF.show()
qa_listDF.limit(1).show()
qa_listDF.head()
qa_listDF.take(1)
qa_listDF.first()
qa_listDF.sample(False, .6, 42).collect()
```

More Options for Returning Data to the Driver

Let's check with dir()

Test some of the available functionality

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

Something is Still Bugging Me

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

```
qa_listDF.show()
qaDF=qa_listDF.toDF('Id','QAReviewer')
qaDF.show()
```

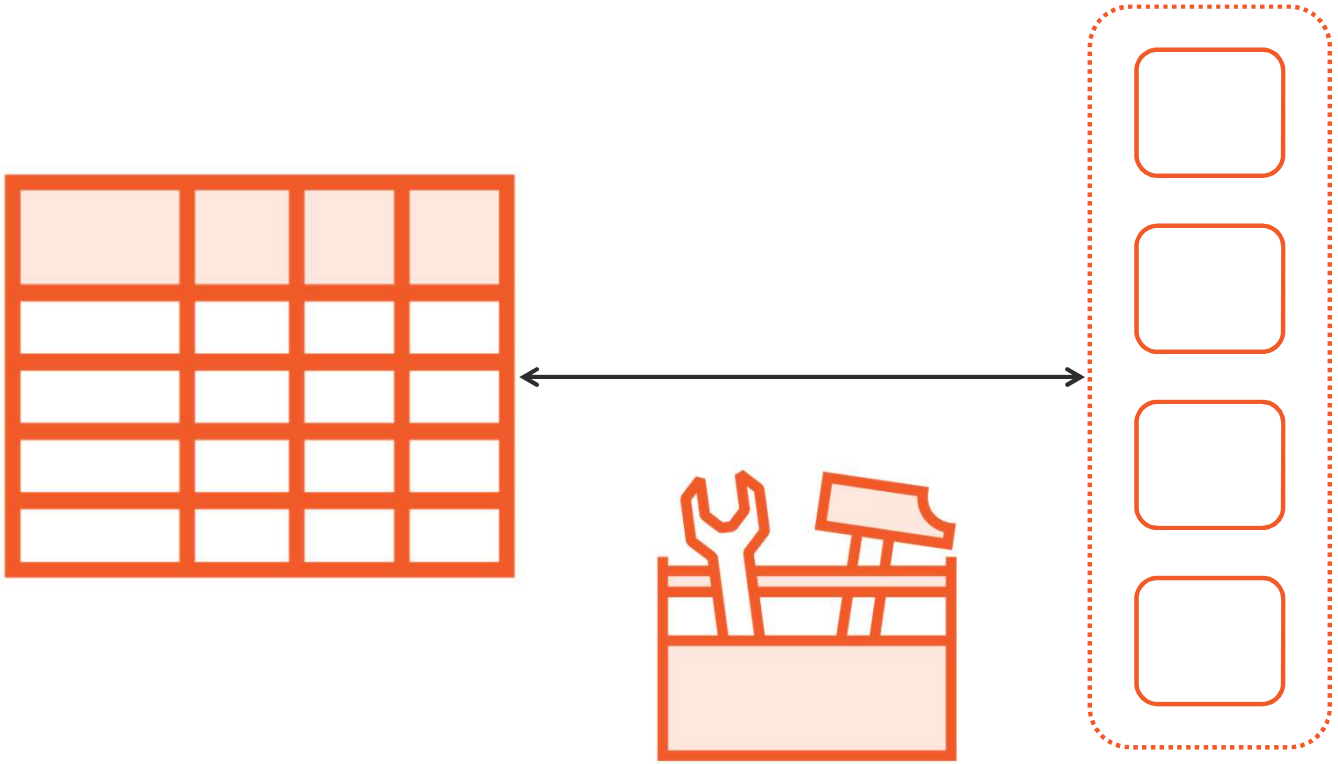
Nicer Column Names

Create a new DataFrame

But with nicer column names!

Use toDF()

DataFrames to RDDs & Viceversa



```
qa_rdd = sc.parallelize([ (1,'Xavier'), (2, 'Irene'),(3, 'Xavier')])
qa_rdd
qa_with_ToDF = qa_rdd.toDF()
qa_with_create = spark.createDataFrame(qa_rdd)
qa_rdd.collect()
qa_with_ToDF.show()
qa_with_create.show()
```

DataFrames to RDDs & Viceversa

Create an RDD

Use toDF() on RDD to get a DataFrame

Use rdd on DataFrame to get an RDD

Prerequisite

```
spark2-submit --packages
              com.databricks:spark-xml_2.11:0.4.1,
              com.databricks:spark-csv_2.11:1.5.0
              prepare_data_badges_csv.py
```

Badges Data *(loaded in a previous module)*

Data preparation step

```
badges_columns_rdd.take(3)
badges_from_rddDF = badges_columns_rdd.toDF()
badges_from_rddDF.show()
badges_from_rddDF.printSchema()
```

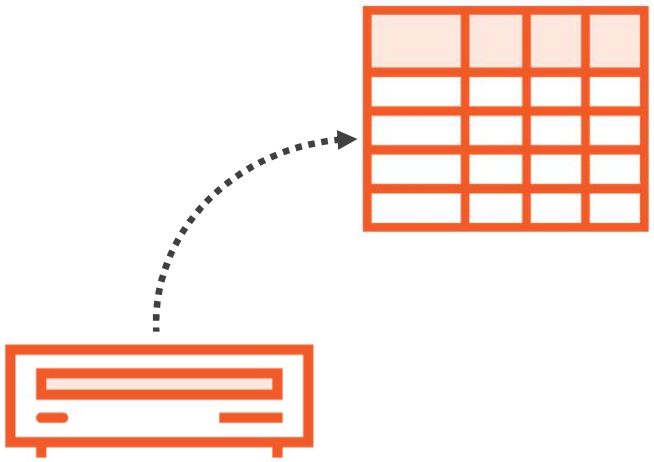
DataFrames

Use our StackOverflow / StackExchange RDDs

Badges data

Schema territory

Loading DataFrames

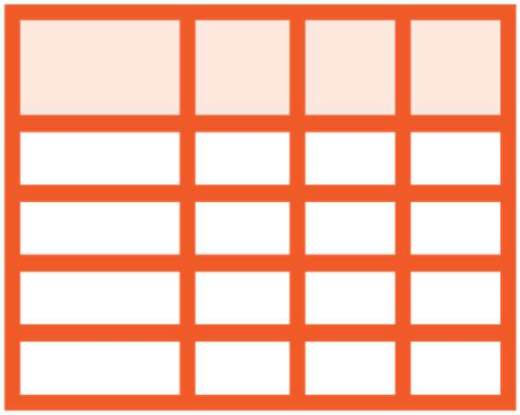


Data stored somewhere

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

```
posts_no_schemaTxtDF=spark.read
.format('text').load('/user/cloudera/stackexchange/posts_all_csv')
posts_no_schemaTxtDF.printSchema()
posts_no_schemaTxtDF.show()
posts_no_schemaTxtDF.show(truncate=False)
```

Loading DataFrames

Use DataFrameReader, with SparkSession.read()

You specify format() and load()

Let's start with text

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

posts_no_schemaTxtDF.show()

posts_no_schemaTxtDF.show(truncate=False)
```

Specifying Format

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

- Quicker and more intuitive way

Use text()

```
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

csv(*path*, *schema=None*, *sep=None*, *encoding=None*, *quote=None*, *escape=None*, *comment=None*, *header=None*, *inferSchema=None*, *ignoreLeadingWhiteSpace=None*, *ignoreTrailingWhiteSpace=None*, *nullValue=None*, *nanValue=None*, *positiveInf=None*, *negativeInf=None*, *dateFormat=None*, *timestampFormat=None*, *maxColumns=None*, *maxCharsPerColumn=None*, *maxMalformedLogPerPartition=None*, *mode=None*, *columnNameOfCorruptRecord=None*, *multiLine=None*)

Loads a CSV file 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`.

Parameters:

- **path** – string, or list of strings, for input path(s).
- **schema** – an optional **pyspark.sql.types.StructType** for the input schema.
- **sep** – sets the single character as a separator for each field and value. If None is set, it uses the default value, `,`.
- **encoding** – decodes the CSV files by the given encoding type. If None is set, it uses the default value, `UTF-8`.
- **quote** – sets the single character used for escaping quoted values where the separator can be part of the value. If None is set, it uses the default value, `"`. If you would like to turn off quotations, you need to set an empty string.
- **escape** – sets the single character used for escaping quotes inside an already quoted value. If None is set, it uses the default value, `\`.
- **comment** – sets the single character used for skipping lines beginning with this character. By default (None), it is disabled.
- **header** – uses the first line as names of columns. If None is set, it uses the default value, `false`.
- **inferSchema** – infers the input schema automatically from data. It requires one extra pass over the data. If None is set, it uses the default value, `false`.
- **ignoreLeadingWhiteSpace** – A flag indicating whether or not leading whitespaces from values being read should be skipped. If None is set, it uses the default value, `false`.
- **ignoreTrailingWhiteSpace** – A flag indicating whether or not trailing whitespaces from values being read should be skipped. If None is set, it uses the default value, `false`.
- **nullValue** – sets the string representation of a null value. If None is set, it uses the default value, empty string. Since 2.0.1, this `nullValue` param applies to all supported types including the string type.
- **nanValue** – sets the string representation of a non-number value. If None is set, it uses the default value, `NaN`.
- **positiveInf** – sets the string representation of a positive infinity value. If None is set, it uses the default value, `Inf`.
- **negativeInf** – sets the string representation of a negative infinity value. If None is set, it uses the default value, `Inf`.
- **dateFormat** – sets the string that indicates a date format. Custom date formats follow the formats at `java.text.SimpleDateFormat`. This applies to date type. If None is set, it uses the default value, `yyyy-MM-dd`.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at `java.text.SimpleDateFormat`. This applies to timestamp type. If None is set, it uses the default value, `yyyy-MM-dd'T'HH:mm:ss.SSSXXX`.
- **maxColumns** – defines a hard limit of how many columns a record can have. If None is set, it uses the default value, `20480`.
- **maxCharsPerColumn** – defines the maximum number of characters allowed for any given value being read. If None is set, it uses the default value, `-1` meaning unlimited length.
- **maxMalformedLogPerPartition** – this parameter is no longer used since Spark 2.2.0. If specified, it is ignored.
- **mode** –

  allows a mode for dealing with corrupt records during parsing. If None is
  set, it uses the default value, `PERMISSIVE`.

  - `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** – allows renaming the new field having malformed string created by `PERMISSIVE` mode. This overrides `spark.sql.columnNameOfCorruptRecord`. If None is set, it uses the value specified in `spark.sql.columnNameOfCorruptRecord`.
- **multiLine** – parse records, which may span multiple lines. If None is set, it uses the default value, `false`.

```
>>> df = spark.read.csv('python/test_support/sql/ages.csv')
>>> df.dtypes
[('_c0', 'string'), ('_c1', 'string')]
```

```
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!

```
posts_inferredDF=spark.read.csv('/user/cloudera/stackexchange/
posts_all_csv',inferSchema=True)

posts_inferredDF.printSchema()
```

Infer Schema

Spark can infer the schema

Specify inferSchema as parameter

```
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

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

Lazy Reading of Data(?)

RDD is lazy

DataFrames read ahead

```
posts_optionDF = spark.read\
.option("inferSchema", "true")\
.csv('/user/cloudera/stackexchange/posts_all_csv')
```

Option

Use option() to pass parameters

Chain multiple option() or use options()

Many more possibilities

Prerequisite

```
spark2-submit --packages
              com.databricks:spark-xml_2.11:0.4.1,
              com.databricks:spark-csv_2.11:1.5.0
              prepare_data_posts_all_csv_with_header.py
```

Prepare CSV with Headers

Data preparation step

```
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?

```
from pyspark.sql.types import *
postsSchema = \
StructType([
StructField("Id", IntegerType()),
```

Explicit Schemas

Import types

StructType holds the schema

StructField is each field

Specify Schema

```
from pyspark.sql.types import *

postsSchema = \
StructType([
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())])
```

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

postsDF.printSchema()

postsDF.schema

postsDF.dtypes

postsDF.columns

len(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

```
spark2-submit --packages
               com.databricks:spark-xml_2.11:0.4.1,
               com.databricks:spark-csv_2.11:1.5.0
               prepare_data_comments_parquet.py
```

Prepare Comments as Parquet

Data preparation step

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

Loading Parquet

Use parquet()

Schema preserved, no need to specify inferSchema

Better than text

```
spark2-submit --packages com.databricks:spark-xml_2.11:0.4.1,
                         com.databricks:spark-csv_2.11:1.5.0
                         prepare_data_tags_json.py
```

Convert Tags.xml to JSON

Data preparation step

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

tags_jsonDF.printSchema()
```

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

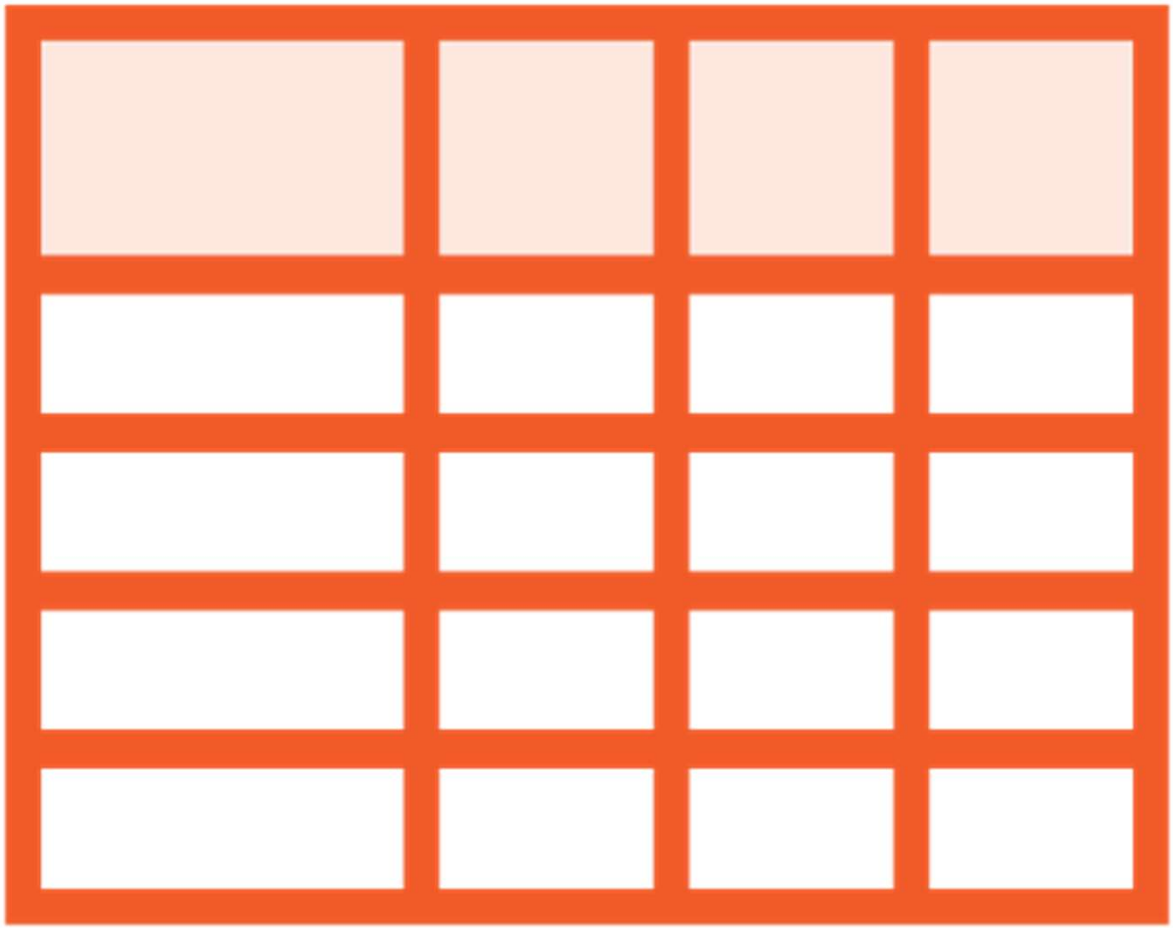
Page 1 to 6 of 6

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

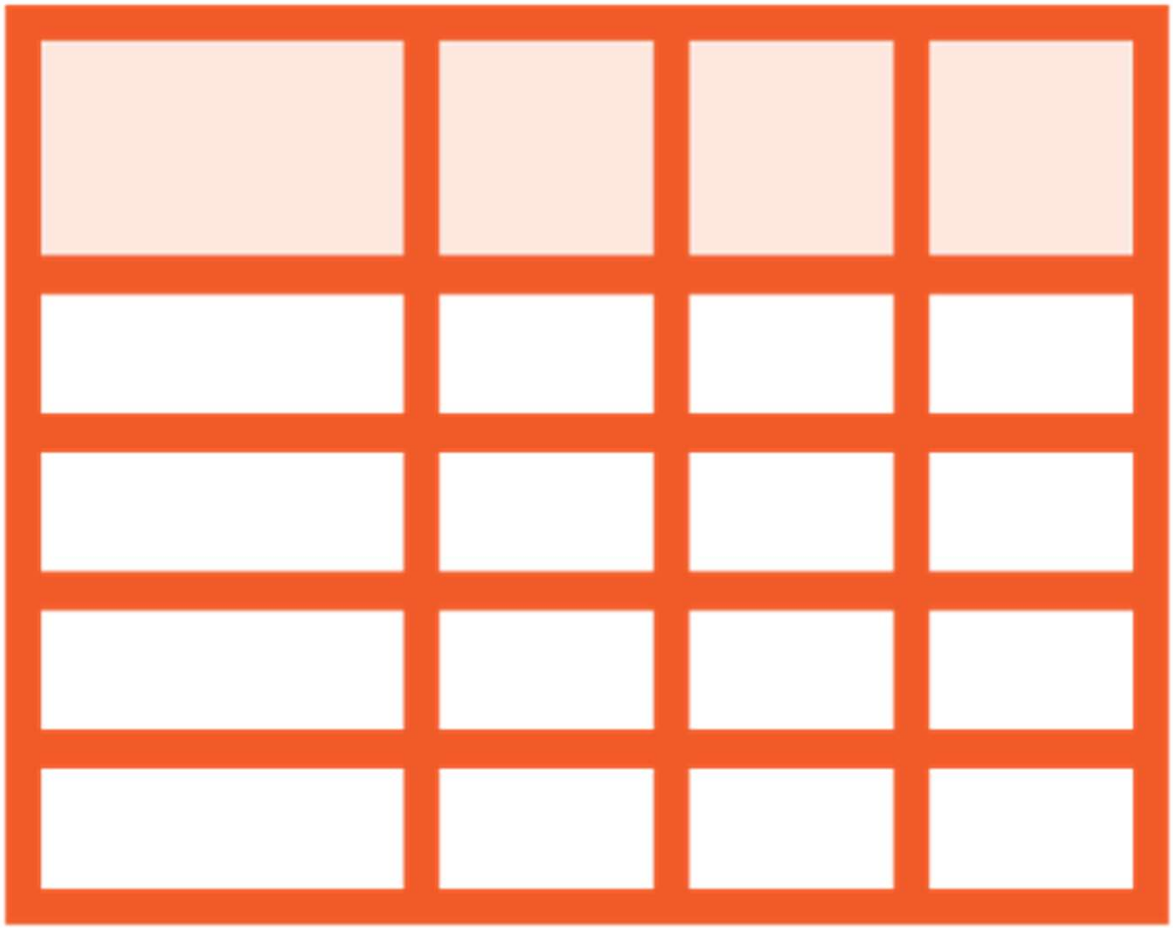

```
{"Id":"1","TagName":"line-numbers","Count":"33","ExcerptPostId":"4450","WikiPostId":"4449"}
{"Id":"2","TagName":"indentation","Count":"166","ExcerptPostId":"3239","WikiPostId":"3238"}
{"Id":"6","TagName":"macro","Count":"68","ExcerptPostId":"856","WikiPostId":"855"}
{"Id":"7","TagName":"text-generation","Count":"23","ExcerptPostId":"6549","WikiPostId":"6548"}
{"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

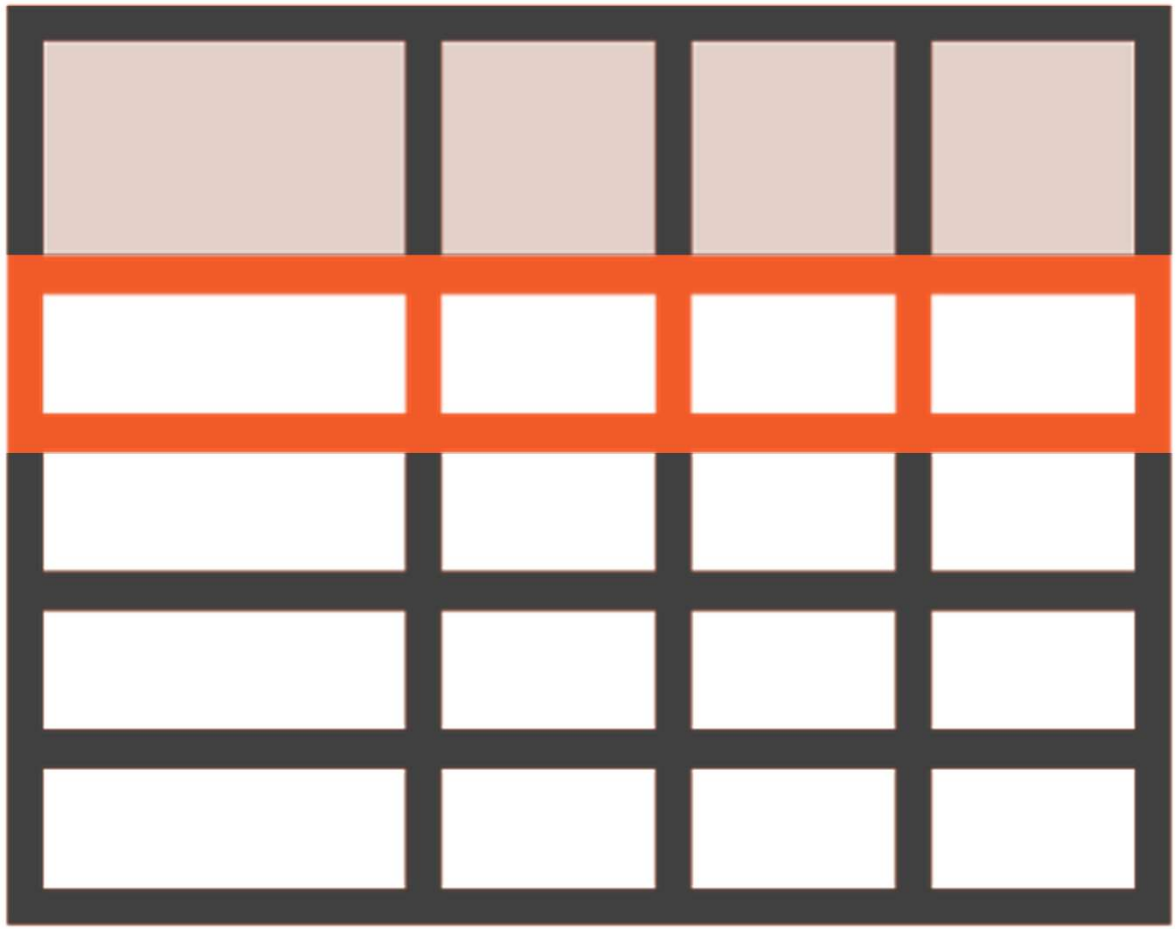
Rows, Columns, Expressions & Operators



Rows



Rows

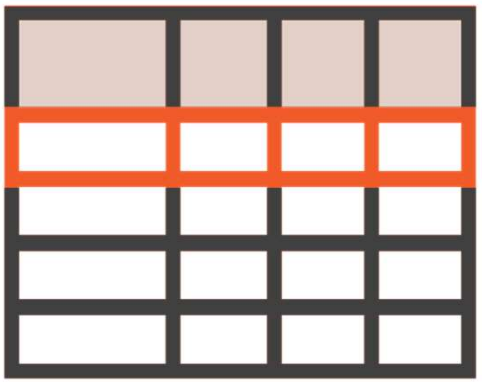


```
postsDF.take(1)
postsDF.show(1)
tags_jsonDF.show(5)
postsDF.columns
```

Row

Inspect one row

Column names with their values



Columns, Column Expressions & Column Operators

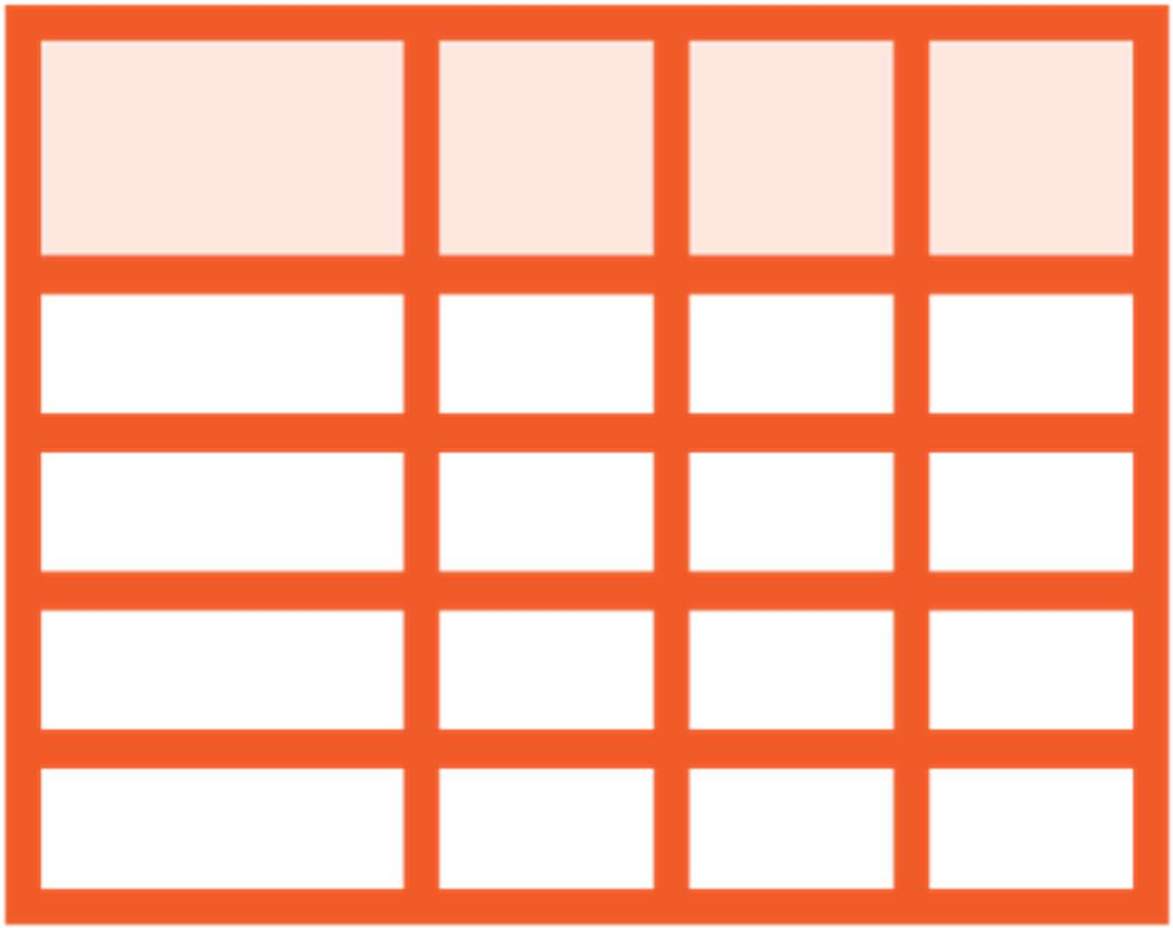
DataFrame

- Named columns

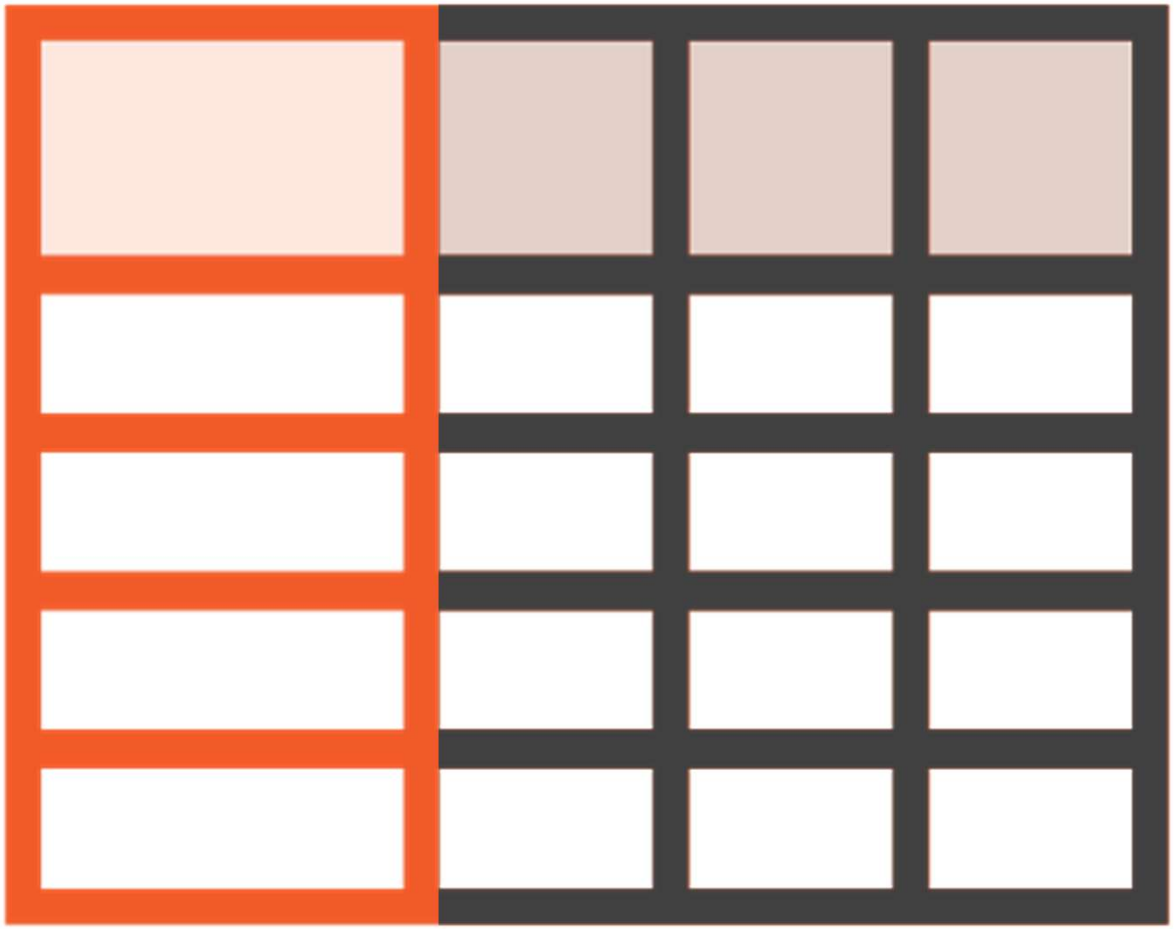
Does not simply store values

- What then?

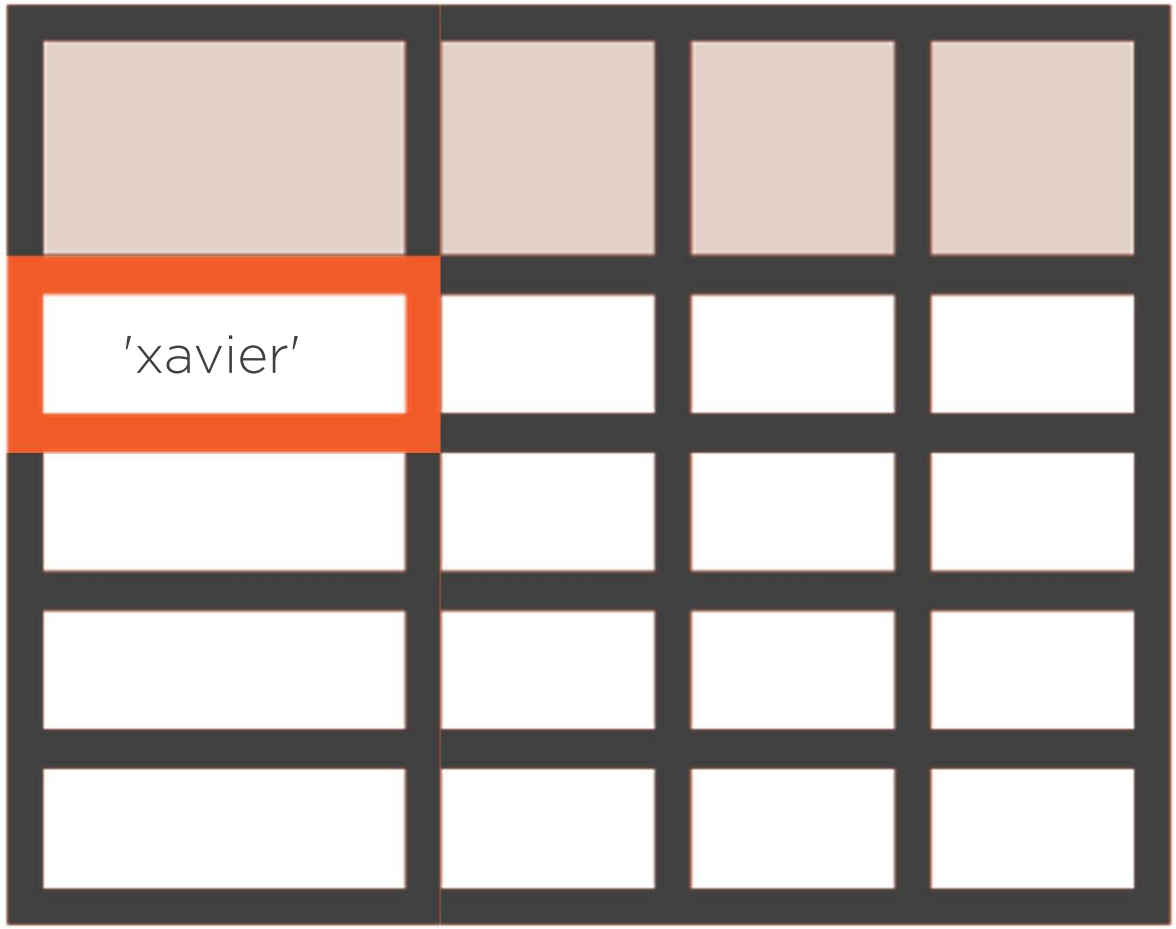
DataFrame



Columns



Columns



Expression

Catalyst expression

Produces a value per row

Based on a value, function or operation

Referring to Columns

Canonical

```
postsDF['Title']
```

Just like a dictionary

Using brackets

Not case sensitive

Dot notation

```
postsDF.Title
```

Specify name of column

Case sensitive

```
col('Title')
```

```
postsDF.select(postsDF.Title).show(1)
postsDF.select(postsDF['Title']).show(1)
postsDF.select('Title').show(1)
postsDF.select('Title', postsDF['Id'], postsDF.CreationDate).show(1)
postsDF.select(postsDF['Title'] ,postsDF['Score']*1000).show(1)
postsDF.select(concat('Title: ', 'Title')).show(5, truncate=False)
postsDF.select(concat(lit('Title: '), 'Title')).show(5, truncate=False)
```

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 pyspark.sql.functions for full list

spark.apache.org/docs/2.2.0/api/python/_modules/pyspark/sql/functions.html

```
    _.__doc__ = 'Window function: ' + doc
    return _


_functions = {
    'lit': 'Creates a :class:`Column` of literal value.',
    'col': 'Returns a :class:`Column` based on the given column name.',
    'column': 'Returns a :class:`Column` based on the given column name.',
    'asc': 'Returns a sort expression based on the ascending order of the given column name.',
    'desc': 'Returns a sort expression based on the descending order of the given column name.',

    'upper': 'Converts a string expression to upper case.',
    'lower': 'Converts a string expression to upper case.',
    'sqrt': 'Computes the square root of the specified float value.',
    'abs': 'Computes the absolute value.',

    'max': 'Aggregate function: returns the maximum value of the expression in a group.',
    'min': 'Aggregate function: returns the minimum value of the expression in a group.',
    'count': 'Aggregate function: returns the number of items in a group.',
    'sum': 'Aggregate function: returns the sum of all values in the expression.',
    'avg': 'Aggregate function: returns the average of the values in a group.',
    'mean': 'Aggregate function: returns the average of the values in a group.',
    'sumDistinct': 'Aggregate function: returns the sum of distinct values in the expression.',
}

_functions_1_4 = {
    # unary math functions
    'acos': 'Computes the cosine inverse of the given value; the returned angle is in the range' +
            '0.0 through pi.',
    'asin': 'Computes the sine inverse of the given value; the returned angle is in the range' +
            '-pi/2 through pi/2.',
    'atan': 'Computes the tangent inverse of the given value.',
    'cbrt': 'Computes the cube-root of the given value.',
    'ceil': 'Computes the ceiling of the given value.',
    'cos': 'Computes the cosine of the given value.',
    'cosh': 'Computes the hyperbolic cosine of the given value.',
    'exp': 'Computes the exponential of the given value.',
    'expm1': 'Computes the exponential of the given value minus one.',
    'floor': 'Computes the floor of the given value.',
```

```
postsDF.select(postsDF.Title).show()
postsDF.select(postsDF['Title']).show()
postsDF.select('Title').show()
postsDF.select(postsDF['Score']*1000).show()
```

Working with Columns

Specify which columns you want to return

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

Check pyspark.sql.functions for full list

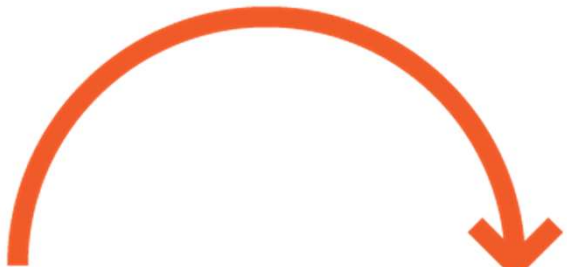
Did you notice the select?

DataFrame DSL feels like SQL

Stay tuned... More on this soon!

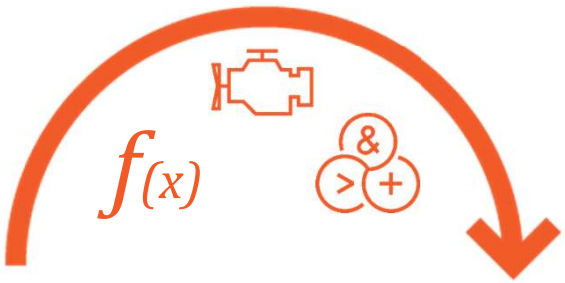
```
postsSchema = StructType([
  StructField("Id", IntegerType()),
  StructField("PostTypeId", IntegerType()),
  StructField("AcceptedAnswerId", IntegerType()),
  StructField("CreationDate", TimestampType()),
  StructField("Score", IntegerType()),
  StructField("ViewCount", StringType()),
  StructField("OwnerUserId", 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
[(x,y) for x, y in postsDF.dtypes if x == 'ViewCount']
postsDF.schema['ViewCount']
postsDF.select('ViewCount').printSchema()

posts_viewDF = postsDF.withColumn('ViewCount', postsDF.ViewCount.cast('integer'))
posts_viewDF.printSchema()
posts_viewDF.select('ViewCount').show()
```

Casting Columns

Change type of a column

Use withColumn() and cast()

Besides casting, you can apply functions

```
postsVCDF = postsDF.withColumnRenamed('ViewCount', 'ViewCountStr')

postsVCDF.printSchema()

posts_twoDF = postsVCDF.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?

```
posts_ticksDF = postsDF.withColumnRenamed('ViewCount', 'ViewCount.Str')
posts_ticksDF.select('ViewCount.Str').show()
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

```
posts_wcDF = postsDF.withColumn('TitleClone1', postsDF.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
'TitleClone1' in posts_wcDF.columns
posts_no_cloneDF = posts_wcDF.drop('TitleClone1')
'TitleClone1' in posts_no_cloneDF.columns
posts_no_cloneDF.printSchema()
```

Dropping Columns

Remove columns from DataFrame

Use drop()

pyspark.sql.functions.**concat**(**cols*) [source]

Concatenates multiple input string columns together into a single string column.

```
>>> df = spark.createDataFrame([('abcd','123')], ['s', 'd'])
>>> df.select(concat(df.s, df.d).alias('s')).collect()
[Row(s=u'abcd123')]
```

New in version 1.5.

pyspark.sql.functions.**lower**(*col*)

Converts a string column to lower case.

New in version 1.5.

```
questionsDF = postsDF.filter(col('PostTypeId') == 1)
questionsDF.select('Tags').show(20, truncate=False)
```

More Than pyspark.sql.functions

There are many functions available

Sometimes you may need more

course.functions **give_me_list**(*string_list*) [source]

A function I just created in this course to manipulate some text, taking the representation of an array as string, and creating a Python list

```
def give_me_list(string_list):
    if string_list is None:
        return []
    elements = string_list[1:len(string_list) - 1]
    return list(elements.split(','))
```

Created only for this course

```
from pyspark.sql.functions import udf

udf_give_me_list=udf(give_me_list, ArrayType(StringType()))

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(5)
```

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

```
from pyspark.sql.functions import 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