What we will be discussing...

- Apache Spark SQL
- DataFrame
- Catalyst Optimizer
- Examples in DSL and SQL
- Example of adding a new rule on Catalyst Optimizer
### Nowadays Challenges and Solutions

<table>
<thead>
<tr>
<th>Challenges</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perform ETL to and from various (semi or unstructured) data sources</td>
<td>A <em>DataFrame</em> API that can perform relational operations on both external data sources and Spark’s built-in RDDs</td>
</tr>
<tr>
<td>Perform advanced analytics (e.g. machine learning, graph processing) that are hard to express in relational systems</td>
<td>A highly extensible optimizer, <em>Catalyst</em>, that uses features of Scala to add composable rule, control code gen., and define extensions.</td>
</tr>
</tbody>
</table>
Why Apache Spark?

Fast and general cluster computing system, interoperable with Hadoop

Improves efficiency through:
- In-memory computing primitives
- General computation graphs

Improves usability through:
- Rich APIs in Scala, Java, Python
- Interactive shell

Note: More about Hadoop versus Spark [here](#).
Spark SQL

Is a Spark module which Integrates relational processing with Spark’s functional programming API

Module Characteristics:

- Supports querying data either via SQL or via Hive Query Language
- Extends the traditional relational data processing

Part of the core distribution since Spark 1.0 (April 2014):
How to use Spark SQL?

You issue SQL queries through a SQLContext or HiveContext, using the sql() method.

- The sql() method returns a DataFrame
- You can mix DataFrame methods and SQL queries in the same code

To use SQL you must either:

- Query a persisted Hive table
- Make a table alias for a DataFrame, using the registerTempTable() method

Note: a complete guide on how to use, can be found [here](#)
DataFrame API

*Provides a higher level abstraction (built on RDD API), allowing us to use a query language to manipulate data*

Formal Definition:

- A **DataFrame (DF)** is a size-mutable, potentially heterogeneous tabular data structure with labeled axes (i.e., rows and columns)

Characteristics:

- Supports all the RDD operations → but may return back an RDD not a DF
- Ability to scale from kB of data in a single laptop to petabytes on a large cluster
- Support for a wide array of data formats and storage systems
- State-of-the-art optimization and code generation through the Spark SQL **Catalyst optimizer**
- ...

Spark SQL Interfaces Interaction with SPARK

- Seamless integration with all big data tooling and infrastructure via Spark.
- APIs for Python, Java and R
Why DataFrame?

What are the advantages over Resilient Distributed Datasets?

1. **Compact binary representation**
   - Columnar, compressed cache; rows for processing
2. **Optimization across operations** (join, reordering, predicate pushdown, etc)
3. Runtime **code generation**

What are the advantages over Relational Query Languages?

- Holistic **optimization across functions** composed in different languages
- Control structures (e.g., if, for)
- Logical plan **analyzed eagerly** → identify code errors associated with data schema issues on the fly
Why DataFrame?

A DF can be **significantly faster** than RDDs and they perform the **same regardless the language**:

![Bar chart showing performance comparison between DataFrame SQL, DataFrame R, DataFrame Python, DataFrame Scala, RDD Python, and RDD Scala.]

But, we have **lost type safety** → `Array[org.apache.spark.sql.Row]`, because `Row` extends `Serializable`. **Mapping** it back to something **useful** e.g. `row(0).asInstanceOf[String]`, its ugly and error-prone.
Querying Native Datasets

Infer **column names** and **types** directly from data objects:

```scala
case class User(name: String, age: Int)
```

- **Native objects** accessed in-place to avoid expensive data format transformation

**Benefits:**

- Run **relational operations** on existing Spark Programs
- Combine **RDDs with external structured data**

```
RDD[String] → (User Defined Function) → RDD[User] → (toDF method) → DataFrame
```
User-Defined Functions (UDFs)

Easy extension of limited operations supported
Allows inline registration of UDFs

- Compare with Pig, which requires the UDF to be written in java package that’s loaded into the Pig script

Can be defined on simple data types or entire tables
UDFs available to other interfaces after registration

```scala
val model: LogisticRegressionModel = ...

ctx.udf.register("predict",
  (x: Float, y: Float) => model.predict(Vector(x, y)))

ctx.sql("SELECT predict(age, weight) FROM users")
```
Transformations contribute to the query plan, but they don't execute anything. Actions cause the execution of the query.

**DataFrames are lazy!**

<table>
<thead>
<tr>
<th>Transformation examples</th>
<th>Action examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>count</td>
</tr>
<tr>
<td>select</td>
<td>collect</td>
</tr>
<tr>
<td>drop</td>
<td>show</td>
</tr>
<tr>
<td>intersect</td>
<td>head</td>
</tr>
<tr>
<td>join</td>
<td>take</td>
</tr>
</tbody>
</table>

What exactly does “execution of the query” means?

- Spark initiates a distributed read of the data source
- The data flows through the transformations (the RDDs resulting from the catalyst query plan)
- The result of the action is pulled back into the driver JVM
## DataFrame API: Actions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>collect()</code></td>
<td>Returns an array that contains all of Rows in this DataFrame.</td>
</tr>
<tr>
<td><code>collectAsList()</code></td>
<td>Returns a Java list that contains all of Rows in this DataFrame.</td>
</tr>
<tr>
<td><code>count()</code></td>
<td>Returns the number of rows in the DataFrame.</td>
</tr>
<tr>
<td><code>describe(-cols: String*)</code></td>
<td>Computes statistics for numeric columns, including count, mean, stddev, min, and max.</td>
</tr>
<tr>
<td><code>first()</code></td>
<td>Returns the first row.</td>
</tr>
<tr>
<td><code>head()</code></td>
<td>Returns the first row.</td>
</tr>
<tr>
<td><code>head(n: Int)</code></td>
<td>Returns the first n rows.</td>
</tr>
<tr>
<td><code>show()</code></td>
<td>Displays the top 20 rows of DataFrame in a tabular form.</td>
</tr>
<tr>
<td><code>show(numRows: Int)</code></td>
<td>Displays the DataFrame in a tabular form.</td>
</tr>
<tr>
<td><code>take(n: Int)</code></td>
<td>Returns the first n rows in the DataFrame.</td>
</tr>
</tbody>
</table>
### Basic DataFrame functions

- **cache()**
  - Returns all column names as an array.

- **columns: Array[String]**
  - Returns all column names as an array.

- **dtypes: Array[(String, String)]**
  - Returns all column names and their data types as an array.

- **explain()**: Unit
  - Only prints the physical plan to the console for debugging purposes.

- **explain(extended: Boolean): Unit**
  - Prints the plans (logical and physical) to the console for debugging purposes.

- **isLocal**: Boolean
  - Returns true if the collect and take methods can be run locally (without any Spark executors).

- **persist(newLevel: StorageLevel): DataFrame.this.type**

- **persist()**: DataFrame.this.type

- **printSchema()**: Unit
  - Prints the schema to the console in a nice tree format.

- **registerTempTable(tableName: String): Unit**
  - Registers this DataFrame as a temporary table using the given name.
### Basic DataFrame functions

- **def schema: StructType**
  - Returns the schema of this DataFrame.

- **def toDF(colNames: String*): DataFrame**
  - Returns a new DataFrame with columns renamed.

- **def toDF(): DataFrame**
  - Returns the object itself.

- **def unpersist(): DataFrame.this.type**

- **def unpersist(blocking: Boolean): DataFrame.this.type**
**DataFrame API: Language Integrated Queries**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg(expr: Column, exprs: Column*)</code>: DataFrame</td>
<td>Aggregates on the entire DataFrame without groups.</td>
</tr>
<tr>
<td><code>agg(exprs: Map[String, String])</code>: DataFrame</td>
<td>(Java-specific) Aggregates on the entire DataFrame without groups.</td>
</tr>
<tr>
<td><code>agg(exprs: Map[String, String])</code>: DataFrame</td>
<td>(Scala-specific) Aggregates on the entire DataFrame without groups.</td>
</tr>
<tr>
<td><code>agg(aggExpr: (String, String), aggExprs: (String, String)*): DataFrame</code></td>
<td>(Scala-specific) Aggregates on the entire DataFrame without groups.</td>
</tr>
<tr>
<td><code>apply(colName: String)</code>: Column</td>
<td>Selects column based on the column name and return it as a Column.</td>
</tr>
<tr>
<td><code>as(alias: Symbol): DataFrame</code></td>
<td>(Scala-specific) Returns a new DataFrame with an alias set.</td>
</tr>
</tbody>
</table>

**Note:** More details about these functions [here](#).
DataFrame API: Relational Operations

Relational operations, `select`, `where`, `join`, `groupBy` via a domain-specific language:

- Operators take *expression* objects
- Operators build up an *Abstract Syntax Tree (AST)*, which is then *optimized* by *Catalyst*

```scala
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

Alternatively, register as temp SQL table and perform traditional *SQL query* strings:

```sql
users.where(users("age") < 21)
  .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```
DataFrame API: Output Operations

```
def write: DataFrameWriter
    Interface for saving the content of the DataFrame out into external storage.
```
DataFrame API: RDD Operations

```scala
// RDD Operations

def coalesce(numPartitions: Int): DataFrame
Returns a new DataFrame that has exactly numPartitions partitions.

def flatMap[R](f: Row => TraversableOnce[R])(implicit arg0: ClassTag[R]): RDD[R]
Returns a new RDD by first applying a function to all rows of this DataFrame, and then flattening the results.

def foreach(f: Row => Unit): Unit
Applies a function f to all rows.

def foreachPartition(f: Iterator[Row] => Unit): Unit
Applies a function f to each partition of this DataFrame.

def javaRDD: JavaRDD[Row]
Returns the content of the DataFrame as a JavaRDD of Rows.

def map[R](f: Row => R)(implicit arg0: ClassTag[R]): RDD[R]
Returns a new RDD by applying a function to all rows of this DataFrame.

def mapPartitions[R](f: Iterator[Row] => Iterator[R])(implicit arg0: ClassTag[R]): RDD[R]
Returns a new RDD by applying a function to each partition of this DataFrame.

lazy val rdd: RDD[Row]
Represents the content of the DataFrame as an RDD of Rows.

def repartition(numPartitions: Int): DataFrame
Returns a new DataFrame that has exactly numPartitions partitions.

def toJSON: RDD[String]
Returns the content of the DataFrame as a RDD of JSON strings.

def toJavaRDD: JavaRDD[Row]
Returns the content of the DataFrame as a JavaRDD of Rows.
```
Data Sources

Uniform way to access structured data:

- Apps can migrate across Hive, Cassandra, JSON, Parquet, etc..
- Rich semantics allows query pushdown into data sources
Apache Spark Catalyst Internals

Deep Dive into Spark SQL’s Catalyst Optimizer
April 13, 2015 | by Michael Armbrust, Yin Huai, Cheng Liang, Reynold Xin and Matei Zaharia

Spark SQL is one of the newest and most technically involved components of Spark. It powers both SQL queries and the new DataFrame API. At the core of Spark SQL is the Catalyst optimizer, which leverages advanced programming language features (e.g. Scala’s pattern matching and quasiquotes) in a novel way to build an extensible query optimizer.

We recently published a paper on Spark SQL that will appear in SIGMOD 2015 (co-authored with Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, and Ali Ghodsi). In this blog post we are republishing a section in the paper that explains the internals of the Catalyst optimizer for broader consumption.
From the above diagram, you can already predict the amount of work that is being done by Spark Catalyst to execute your Spark SQL queries 😳

- The SQL queries of Spark application will be converted to Dataframe APIs
- Logical Plan is converted to an **Optimized Logic plan** and then to one or more **Physical Plans**

**Note:** Find more about what happening under the hood of Spark SQL [here](#) and [here](#).
Spark Catalyst’s analyzer is responsible for resolving types and names of attributes in SQL queries

- The analyzer looks at the table statistics to know the types of the referred column. For example:

  ```sql
  SELECT (col1 + 1) FROM mytable;
  ```

- Now, Spark needs to know:
  1. If `col1` is actually a valid column in `mytable`
  2. If the type of the referred column needs to be known so that `(col1 + 1)` can be validated and necessary type casts can be added
How analyzer resolve attributes?

To resolve attributes:

- **Look up relations** by name from the catalog
- **Map named attributes** to the input provided given operator’s children
- **UID** for references to the same value
- **Propagate** the coerce types **through expressions** (e.g. 1 + col1)
Spark Catalyst’s optimizer is responsible for generating an optimized logical plan from the analyzed logical plan.

- Optimization is done by applying rules in batches. Each operation is represented as a `TreeNode` in Spark SQL.
- When an analyzed plan goes through the optimizer, the tree is transformed to a new tree repeatedly by applying a set of optimization rules.

For instance, a simple Rule:

```
Replace the addition of Literal values with new Literal
```

Then, expressions of the form \((1+5)\) will be replaced by \(6\). Spark will be repeatedly apply such rules to the expression tree until the tree becomes constant.
The optimizer applies standard rule-based optimization rules:

- Constant folding
- Predicate-pushdown
- Projection
- Null propagation
- Boolean expression simplification
- ...

**Note:** Find more optimization rules [here](#)
Optimizer: Example

- An inefficient query where **filter** is used before **join** operation → Costly shuffle operation (Find more about this example [here](#))

```java
users.join(events, users("id") === events("uid")).filter(events("date") > "2015-01-01")
```
Physical plans are the ones that can actually be executed on a cluster. They actually translate optimized logical plans into RDD operations to be executed on the data source.

- A generated **Optimized Logical Plan** is passed through a series of Spark strategies that produce one or more **Physical plans** (More about these strategies [here](#))
- Spark uses cost based optimization (CBO) to select the best physical plan based on the data source (i.e. table sizes)
```python
def add_demographics(events):
    u = sqlCtx.table("users")
    events \n        .join(u, events.user_id == u.user_id) \n        .withColumn("city", zipToCity(df.zip)) \n    # Load partitioned Hive table
    # Join on user_id
    # Run udf to add city column
    events = add_demographics(sqlCtx.load("/data/events", "parquet"))
    training_data = events.where(events.city == "Melbourne").select(events.timestamp).collect()
```
A comparison of the performance evaluating the expression “x + x + x”, where x is an integer, 1 billion times:

Catalyst transforms a SQL tree into an abstract syntax tree (AST) for scala code to evaluate expressions and generate code.
import org.apache.spark.sql._

// Create a Spark Session
val spark = SparkSession.builder().appName("test").master("local").getOrCreate()

// read some text source file
val srcDF = spark.read.format("csv").option("header", "true").option("inferSchema", "true").load("/home/jovyan/sales_src.csv")

// self explanatory i guess ? multiply Units Sold column by 2
val unitsBy2 = srcDF.withColumn("Units Sold", ".Units Sold * 2") // transformation

// Filter rows by order id
val filterOrderId = unitsBy2.filter("Order Id" > 100) // transformation

// select only
val select = filterOrderId.select("Region") // transformation

select.take(10) // action

select.explain(extended=true) // spark, please tell me what you did under the hood

Save it as spark_sql_example.scala (Find the source code here)
Run your first `.scala` script, in three simple steps:

1. Open a command line → `win + R` and type `CMD`
2. Run the spark shell using user-defined memory → `spark-shell --driver-memory 5g`
3. Load the script → `:load <path to>\spark_sql_example.scala`
Suppose you have a text file that looks like this:

<table>
<thead>
<tr>
<th>First name</th>
<th>Last name</th>
<th>Gender</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erin</td>
<td>Shannon</td>
<td>F</td>
<td>42</td>
</tr>
<tr>
<td>Norman</td>
<td>Lockwood</td>
<td>M</td>
<td>81</td>
</tr>
<tr>
<td>Miguel</td>
<td>Ruiz</td>
<td>M</td>
<td>64</td>
</tr>
<tr>
<td>Rosalita</td>
<td>Ramirez</td>
<td>F</td>
<td>14</td>
</tr>
<tr>
<td>Ally</td>
<td>Garcia</td>
<td>F</td>
<td>39</td>
</tr>
<tr>
<td>Claire</td>
<td>McBride</td>
<td>F</td>
<td>23</td>
</tr>
<tr>
<td>Abigail</td>
<td>Cottrell</td>
<td>F</td>
<td>75</td>
</tr>
<tr>
<td>José</td>
<td>Rivera</td>
<td>M</td>
<td>59</td>
</tr>
<tr>
<td>Ravi</td>
<td>Dasgupta</td>
<td>M</td>
<td>25</td>
</tr>
</tbody>
</table>

The file has no schema, but looks like:

- **First name**: string
- **Last name**: string
- **Gender**: string
- **Age**: integer

```scala
case class Person(firstName: String, lastName: String, gender: String, age: Int)
val rdd = sc.textFile("people.csv")
val peopleRDD = rdd.map { line =>
  val cols = line.split(",")
  Person(cols(0), cols(1), cols(2), cols(3).toInt)
}
val df = peopleRDD.toDF
// df: DataFrame = [firstName: string, lastName: string, gender: string, age: int]
```
How to see the Content of a DataFrame?

You can have Spark tell you what it thinks the data schema is, by calling the `printSchema()` method (This is mostly useful in the shell)

```
scala> df.printSchema()
root
|-- firstName: string (nullable = true)
|-- lastName: string (nullable = true)
|-- gender: string (nullable = true)
|-- age: integer (nullable = false)
```

You can look at the first $n$ elements in a DataFrame with the `show()` method

If not specified, $n$ defaults to 20

```
scala> df.show()
+-----------+-----------+-----+-----+
<table>
<thead>
<tr>
<th>firstName</th>
<th>lastName</th>
<th>gender</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erin</td>
<td>Shannon</td>
<td>F</td>
<td>42</td>
</tr>
<tr>
<td>Claire</td>
<td>McBride</td>
<td>F</td>
<td>23</td>
</tr>
<tr>
<td>Norman</td>
<td>Lockwood</td>
<td>M</td>
<td>81</td>
</tr>
<tr>
<td>Miguel</td>
<td>Ruiz</td>
<td>M</td>
<td>64</td>
</tr>
<tr>
<td>Rosalita</td>
<td>Ramirez</td>
<td>F</td>
<td>14</td>
</tr>
<tr>
<td>Ally</td>
<td>Garcia</td>
<td>F</td>
<td>39</td>
</tr>
<tr>
<td>Abigail</td>
<td>Cottrell</td>
<td>F</td>
<td>75</td>
</tr>
<tr>
<td>José</td>
<td>Rivera</td>
<td>M</td>
<td>59</td>
</tr>
</tbody>
</table>
+-----------+-----------+-----+-----+
```
How to Persist a DataFrame in Memory?

Spark can **cache a DataFrame**, using an in-memory columnar format, by calling:

```scala
scala> df.cache()
```

Which just calls `df.persist(MEMORY_ONLY)`

- Spark will scan only those columns used by the DataFrame and will automatically tune compression to minimize memory usage and GC pressure.

You can **remove the cached data** from memory, by calling:

```scala
scala> df.unpersist()
```
How to Select Cols from a DataFrame?

The **select()** is like a SQL SELECT, allowing you to limit the results to specific columns.

- The **DSL** also allows you create on-the-fly derived columns.
- The **SQL** version is also available.

```
scala> df.select("firstName", "age").show(5)
+------------+-------+
| firstName  | age   |
| Erin       | 42    |
| Claire     | 23    |
| Norman     | 81    |
| Miguel     | 64    |
| Rosalita   | 14    |
```

```
scala> df.registerTempTable("names")
In[2]: sqlContext.sql("SELECT first_name, age, age > 49 FROM names").show(5)
+------------+-------+----------------+
| first_name | age   | _c2             |
| Erin       | 42    | false           |
| Claire     | 23    | false           |
| Norman     | 81    | true            |
| Miguel     | 64    | true            |
| Rosalita   | 14    | false           |
```

```
scala> df.select("firstName", "age", "age" > 49, "age" + 10).show(5)
+------------+-------+-------+----------------+
| firstName  | age   | age   | (age + 10)     |
| Erin       | 42    | false | 52             |
| Claire     | 23    | false | 33             |
| Norman     | 81    | true  | 91             |
| Miguel     | 64    | true  | 74             |
| Rosalita   | 14    | false | 24             |
```
How to Filter the Rows of a DataFrame?

The `filter()` method allows you to filter rows out of your results.

- The **DSL** as well as **SQL** version are available.

```scala
scala> df.filter($"age" > 49).select($"firstName", $"age").show()
+---------+-----+
| firstName| age |
+---------+-----+
|  Norman |  81 |
|  Miguel |  64 |
| Abigail |  75 |
+---------+-----+

In[1]: SQLContext.sql("SELECT first_name, age FROM names " + 
                  "WHERE age > 49").show()
+---------+-----+
| firstName| age |
+---------+-----+
|  Norman |  81 |
|  Miguel |  64 |
| Abigail |  75 |
+---------+-----+
How to Sort the Rows of a DataFrame?

The `orderBy()` method allows you to sort the results

- The **DSL** as well as **SQL** version are available
- It’s easy to **reverse** the sort order

```scala
scala> df.filter(df("age") > 49).
   select(df("firstName"), df("age").
   orderBy(df("age"), df("firstName"))
   show()

+-----+-----+
| age |
|-----+-----+
| 56  |
| 75  |
| 81  |
+-----+-----+
scala> df.filter($"age" > 49).
   select($"firstName", $"age").
   orderBy($"age".desc, $"firstName").
   show()

+-----+-----+
| firstName | age |
|-----------+-----+
| Miguel    | 64  |
| Abigail   | 75  |
| Norman    | 81  |
+-----+-----+
scala> sqlContext.SQL("SELECT first_name, age FROM names " +
   | "WHERE age > 49 ORDER BY age DESC, first_name").show()

+-----+-----+
| first_name | age |
|-----------+-----+
| Norman    | 81  |
| Abigail   | 75  |
| Miguel    | 64  |
+-----+-----+
```
The `as()` or `alias()` allows you to rename a column. It’s especially useful with generated columns.

- The **DSL** as well as **SQL** version are available.

```scala
scala> df.select("firstName", "age", ("age" < 30).as("young")).show()
+----------+---+-------+
| first_name| age| young |
|-----------+---+-------+
| Erin      | 42| false |
| Claire    | 23| true  |
| Norman    | 81| false |
| Miguel    | 64| false |
| Rosalita  | 14| true  |
+------------+---+-------+
```

```scala
scala> sqlContext.sql("SELECT firstName, age, age < 30 AS young " + 
  | FROM names")
+----------+---+-------+
| first_name| age| young |
|-----------+---+-------+
| Erin      | 42| false |
| Claire    | 23| true  |
| Norman    | 81| false |
| Miguel    | 64| false |
| Rosalita  | 14| true  |
+------------+---+-------+
```
Implement the Collapse sorts optimizer rule

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql._
import org.apache.spark.sql.catalyst.rules.Rule
import org.apache.spark.sql.catalyst.plans.logical._
import org.apache.spark.sql.catalyst.analysis._
import org.apache.spark.sql.catalyst.catalog._
import org.apache.spark.sql.catalyst.expressions.{Expression, InputFileBlockSize, InputFileName, RowOrdering}
```

**Query:**
- `val data = Seq(('a', 1), ('b', 2), ('c', 3)).toDF('a', 'b')`
- `val query = data.select(a, b).orderBy(b.asc).filter('b == 2').orderBy(a.asc)`

**The Optimized logical Plan without our new Rule**

**The Optimized logical Plan with our new Rule**

**Note:** Find more information of this example [here](#)
Which Spark Components do People Use?

- Spark SQL: 69%
- DataFrames: 62%
- Spark Streaming: 58%
- MLlib + GraphX: 58%

75% of users use 2 or more components

(Survey 2015)
Which Languages are Used?

### 2014 Languages Used
- Scala: 84%
- Java: 38%
- Python: 38%

### 2015 Languages Used
- Scala: 71%
- Java: 31%
- Python: 58%
- R: 18%
## Special Thanks!

<table>
<thead>
<tr>
<th>Title</th>
<th>Year</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intro to DataFrames and Spark SQL</td>
<td>2015</td>
<td>Databricks</td>
</tr>
<tr>
<td>RDDs, DataFrames and Datasets in Apache Spark</td>
<td>2016</td>
<td>Akmal B. Chaudhri</td>
</tr>
<tr>
<td>Spark SQL: Relational Data Processing in Spark</td>
<td>2015</td>
<td>Databricks, MIT and Amplab</td>
</tr>
</tbody>
</table>