ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Spark SQL

Big Data Analysis with Scala and Spark

Heather Miller

Relational Databases

SQL is the lingua franca for doing analytics.

Relational Databases

SQL is the lingua franca for doing analytics.

But it's a pain in the neck to connect big data processing pipelines like Spark or Hadoop to an SQL database.

Wouldn't it be nice...

- if it were possible to seamlessly intermix SQL queries with Scala?
- to get all of the optimizations we're used to in the databases community on Spark jobs?

Relational Databases

SQL is the lingua franca for doing analytics.

But it's a pain in the neck to connect big data processing pipelines like Spark or Hadoop to an SQL database.

Wouldn't it be nice...

- if it were possible to seamlessly intermix SQL queries with Scala?
- to get all of the optimizations we're used to in the databases community on Spark jobs?

Spark SQL delivers both!

Spark SQL: Goals

Three main goals:

1. Support **relational processing** both within Spark programs (on RDDs) and on external data sources with a friendly API.

Sometimes it's more desirable to express a computation in SQL syntax than with functional APIs and vice a versa.

Spark SQL: Goals

Three main goals:

1. Support **relational processing** both within Spark programs (on RDDs) and on external data sources with a friendly API.
2. High performance, achieved by using techniques from research in databases.
3. Easily support new data sources such as semi-structured data and external databases.

Spark SQL

Spark SQL is a component of the Spark stack.

- It is a Spark module for structured data processing.
- It is implemented as a library on top of Spark.

Spark SQL

Spark SQL is a component of the Spark stack.

- It is a Spark module for structured data processing.
- It is implemented as a library on top of Spark.

Three main APIs:

- **SQL literal syntax**
- `DataFrames`
- `Datasets`

Spark SQL

Spark SQL is a component of the Spark stack.

- It is a Spark module for structured data processing.
- It is implemented as a library on top of Spark.

Three main APIs:

- **SQL literal syntax**
- `DataFrames`
- `Datasets`

Two specialized backend components:

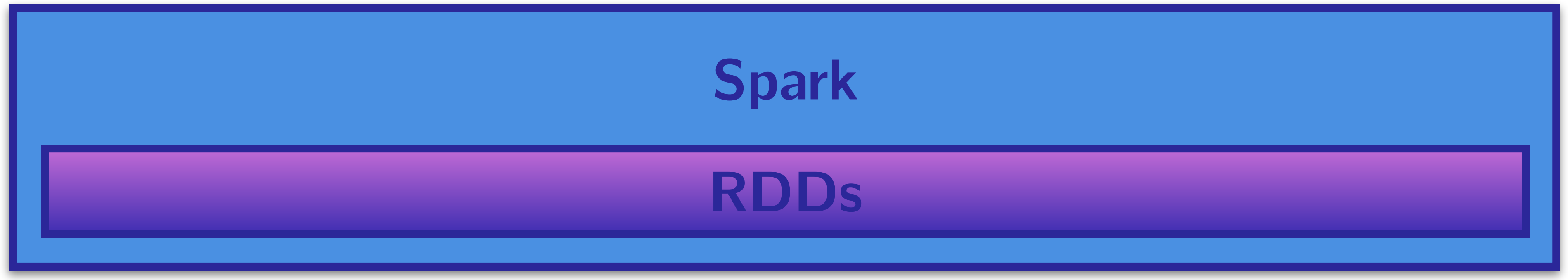
- **Catalyst**, query optimizer.
- **Tungsten**, off-heap serializer.

Spark SQL

Spark SQL is a component of the Spark stack.

- It is a Spark module for structured data processing.
- It is implemented as a library on top of Spark.

Visually, Spark SQL relates to the rest of Spark like this:

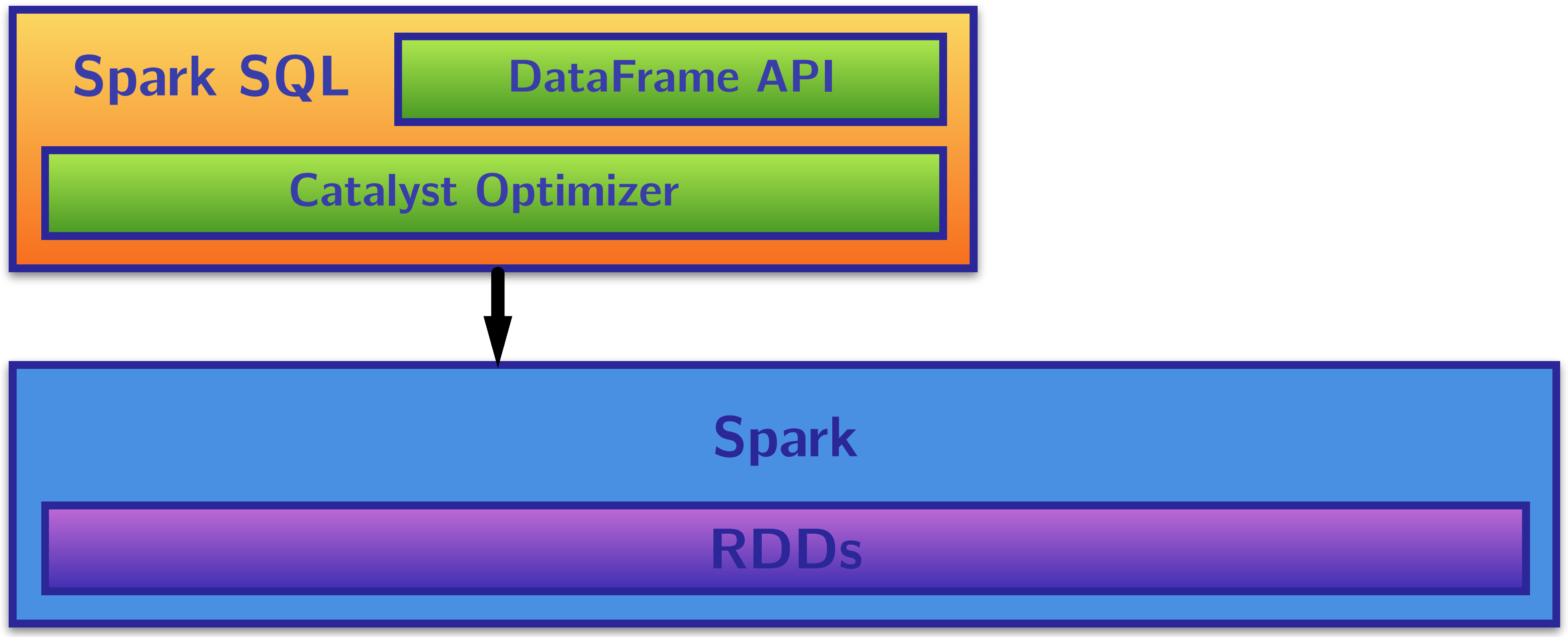


Spark SQL

Spark SQL is a component of the Spark stack.

- It is a Spark module for structured data processing.
- It is implemented as a library on top of Spark.

Visually, Spark SQL relates to the rest of Spark like this:

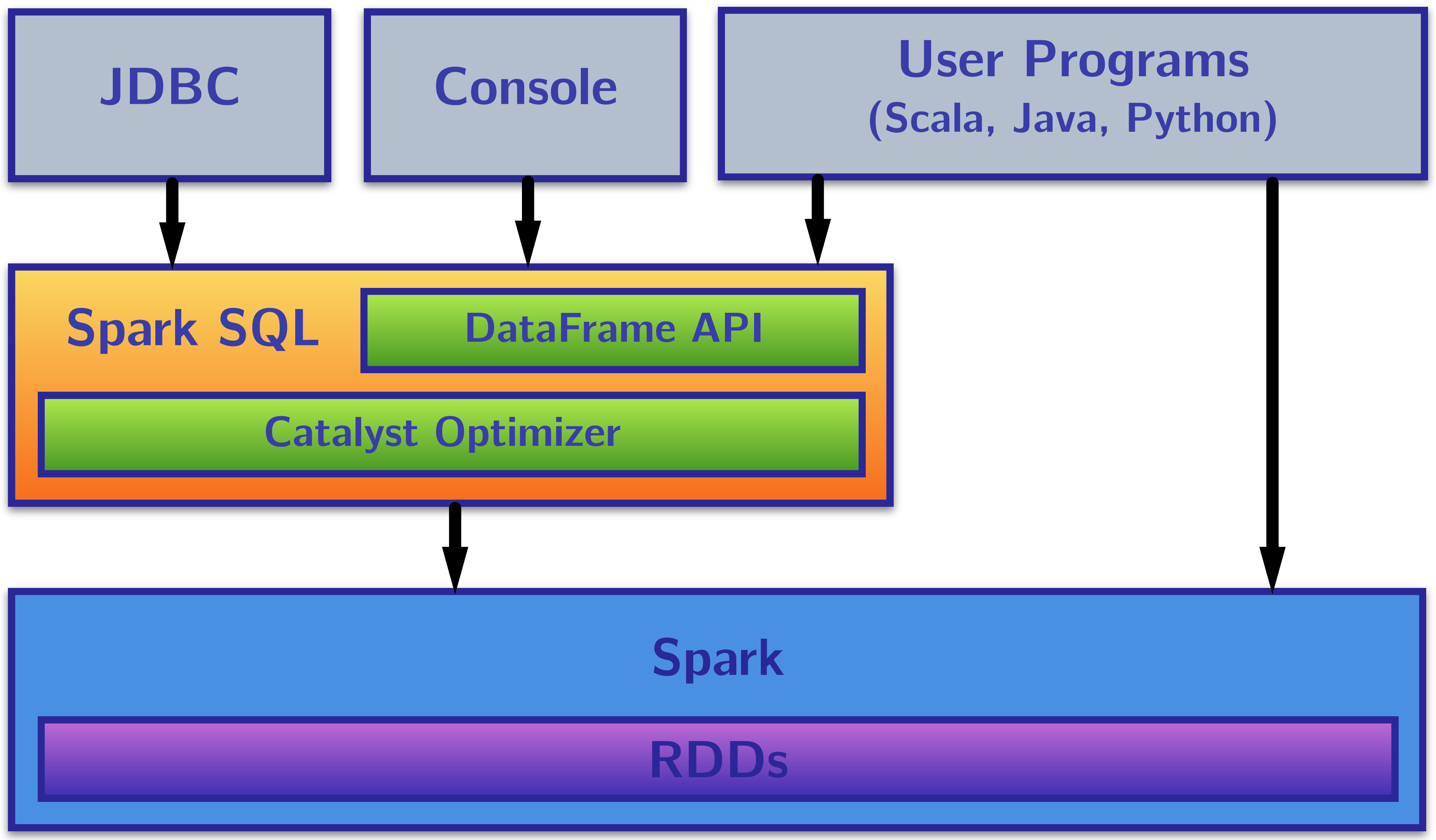


Spark SQL

Spark SQL is a component of the Spark stack.

- It is a Spark module for structured data processing.
- It is implemented as a library on top of Spark.

Visually, Spark SQL relates to the rest of Spark like this:



Relational Queries (SQL)

Everything about SQL is structured.

In fact, SQL stands for *structural query language*.

- There are a set of fixed data types. `Int`, `Long`, `String`, etc.
- There are fixed set of operations. `SELECT`, `WHERE`, `GROUP BY`, etc.

Research and industry surrounding relational databases has focused on exploiting this rigidness to get all kinds of performance speedups.

Relational Queries (SQL)

Everything about SQL is structured.

In fact, SQL stands for *structural query language*.

- There are a set of fixed data types. `Int`, `Long`, `String`, etc.
- There are fixed set of operations. `SELECT`, `WHERE`, `GROUP BY`, etc.

Research and industry surrounding relational databases has focused on exploiting this rigidness to get all kinds of performance speedups.

Let's quickly establish a common set of vocabulary and a baseline understanding of SQL.

Relational Queries (SQL)

Data organized into one or more **tables**

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries (SQL)

Data organized into one or more **tables**

- Tables contain ***columns*** and ***rows***.

columns

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries (SQL)

Data organized into one or more **tables**

- Tables contain ***columns*** and ***rows***.

rows

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries (SQL)

Data organized into one or more **tables**

- Tables contain ***columns*** and ***rows***.
- Tables typically represent a collection of objects of a certain type, such as **`customers`** or **`products`**

SBB customers dataset

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries (SQL)

Data organized into one or more **tables**

- Tables contain ***columns*** and ***rows***.
- Tables typically represent a collection of objects of a certain type, such as **`customers`** or **`products`**

A *relation* is just a table.

Attributes are columns.

attribute

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries (SQL)

Data organized into one or more **tables**

- Tables contain ***columns*** and ***rows***.
- Tables typically represent a collection of objects of a certain type, such as **`customers`** or **`products`**

A *relation* is just a table.

Attributes are columns.

Rows are *records* or *tuples*

record /tuple

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

`DataFrame` is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

`DataFrame`, *is a table, sort of.*

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

`DataFrame` is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records with a known schema

distributed collection of rows/records.

`DataFrame`, *is a table, sort of.*

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

`DataFrame` is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records **with a known schema**

Unlike RDDs though, DataFrames **require** some kind of schema info!

`DataFrame`*, is a table, sort of.*

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

`DataFrame` is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records with a known schema

!! !!

DataFrames are **untyped**!

That is, the Scala compiler doesn't check the types in its schema!

*DataFrames contain **`Rows`** which can contain any schema.*

`DataFrame`, *is a table, sort of.*

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

RDD[T]

DataFrame

Spark SQL

`DataFrame` is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records with a known schema

DataFrames are **untyped**!
That is, the Scala compiler doesn't check the types in its schema!

Transformations on DataFrames are also known as **untyped transformations**

`DataFrame`, *is a table, sort of.*

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Creating DataFrames

`DataFrames` can be created in two ways:

1. From an existing RDD.
   Either with schema inference, or with an explicit schema.
2. Reading in a specific **data source** from file.
   Common structured or semi-structured formats such as JSON.

(1a) Create `DataFrame` from RDD, schema reflectively inferred

Given pair RDD, `RDD[(T1, T2, ... TN)]`, a `DataFrame` can be created with its schema automatically inferred by simply using the `toDF` method.

```
val tupleRDD = ... // Assume RDD[(Int, String String, String)]
val tupleDF = tupleRDD.toDF("id", "name", "city", "country") // column names
```

Note: if you use `toDF` *without arguments, Spark will assign numbers as attributes (column names) to your* `DataFrame`.

_1 _2 _3

Creating DataFrames

(1a) Create `DataFrame` from RDD, schema reflectively inferred

Given pair RDD, `RDD[(T1, T2, ... TN)]`, a `DataFrame` can be created with its schema automatically inferred by simply using the `toDF` method.

```
val tupleRDD = ... // Assume RDD[(Int, String String, String)]
val tupleDF = tupleRDD.toDF("id", "name", "city", "country") // column names
```

Note: if you use `toDF` without arguments, Spark will assign numbers as attributes (column names) to your `DataFrame`.

If you already have an RDD containing some kind of case class instance, then Spark can infer the attributes from the case class's fields.

```
case class Person(id: Int, name: String, city: String)
val peopleRDD = ... // Assume RDD[Person]
val peopleDF = peopleRDD.toDF
```

Creating DataFrames

(1b) Create `DataFrame` from existing RDD, schema explicitly specified

Sometimes it's not possible to create a `DataFrame` with a pre-determined case class as its schema. For these cases, it's possible to explicitly specify a schema.

It takes three steps:

- Create an RDD of `Rows` from the original RDD.
- Create the schema represented by a `StructType` matching the structure of `Rows` in the RDD created in Step 1.
- Apply the schema to the RDD of `Rows` via `createDataFrame` method provided by `SparkSession`.

Given:

```
case class Person(name: String, age: Int)
val peopleRdd = sc.textFile(...) // Assume RDD[Person]
```

(1b) Create `DataFrame` from existing RDD, schema explicitly specified

```
// The schema is encoded in a string
val schemaString = "name age"

// Generate the schema based on the string of schema
val fields = schemaString.split(" ")
  .map(fieldName => StructField(fieldName, StringType, nullable = true))
val schema = StructType(fields)

// Convert records of the RDD (people) to Rows
val rowRDD = peopleRDD
  .map(_.split(","))
  .map(attributes => Row(attributes(0), attributes(1).trim))

// Apply the schema to the RDD
val peopleDF = spark.createDataFrame(rowRDD, schema)
```

Creating DataFrames

(2) Create `DataFrame` by reading in a data source from file.

Using the `SparkSession` object, you can read in semi-structured/structured data by using the `read` method. For example, to read in data and infer a schema from a JSON file:

```
// ‘spark‘ is the SparkSession object we created a few slides back
val df = spark.read.json(”examples/src/main/resources/people.json”)
```

Creating DataFrames

(2) Create `DataFrame` by reading in a data source from file.

Using the `SparkSession` object, you can read in semi-structured/structured data by using the `read` method. For example, to read in data and infer a schema from a JSON file:

```
// ‘spark‘ is the SparkSession object we created a few slides back
val df = spark.read.json(”examples/src/main/resources/people.json”)
```

Semi-structured/Structured data sources Spark SQL can directly create DataFrames from:

- JSON
- CSV
- Parquet
- JDBC

To see a list of all available methods for directly reading in semi-structured/structured data, see the latest API docs for `DataFrameReader`:
`http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameReader`

SQL Literals

Once you have a `DataFrame` to operate on, you can now freely write familiar SQL syntax to operate on your dataset!

SQL Literals

Once you have a DataFrame to operate on, you can now freely write familiar SQL syntax to operate on your dataset!

Given:

A DataFrame called peopleDF, we just have to register our DataFrame as a temporary SQL view first:

```
// Register the DataFrame as a SQL temporary view
peopleDF.createOrReplaceTempView("people")
// This essentially gives a name to our DataFrame in SQL
// so we can refer to it in an SQL FROM statement
```

SQL Literals

Once you have a DataFrame to operate on, you can now freely write familiar SQL syntax to operate on your dataset!

Given:

A DataFrame called peopleDF, we just have to register our DataFrame as a temporary SQL view first:

```
// Register the DataFrame as a SQL temporary view
peopleDF.createOrReplaceTempView("people")
// This essentially gives a name to our DataFrame in SQL
// so we can refer to it in an SQL FROM statement

// SQL literals can be passed to Spark SQL's sql method
val adultsDF
  = spark.sql("SELECT * FROM people WHERE age > 17")
```

SQL Literals

The SQL statements available to you are largely what's available in HiveQL. This includes standard SQL statements such as:

SQL Literals

The SQL statements available to you are largely what's available in HiveQL. This includes standard SQL statements such as:

- `SELECT`
- `FROM`
- `WHERE`
- `COUNT`
- `HAVING`
- `GROUP BY`
- `ORDER BY`
- `SORT BY`
- `DISTINCT`
- `JOIN`
- `(LEFT|RIGHT|FULL) OUTER JOIN`
- Subqueries: `SELECT col FROM ( SELECT a + b AS col from t1) t2`

Supported Spark SQL syntax:
https://docs.datastax.com/en/datastax_enterprise/4.6/datastax_enterprise/spark/sparkSqlSupportedSyntax.html

For a HiveQL cheatsheet:
https://hortonworks.com/blog/hive-cheat-sheet-for-sql-users/

For an updated list of supported Hive features in Spark SQL, the official Spark SQL docs enumerate:
https://spark.apache.org/docs/latest/sql-programming-guide.html#supported-hive-features

A More Interesting SQL Query

Let's assume we have a `DataFrame` representing a data set of employees:

```scala
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)

// DataFrame with schema defined in Employee case class
val employeeDF = sc.parallelize(...).toDF
```

A More Interesting SQL Query

Let's assume we have a DataFrame representing a data set of employees:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)

// DataFrame with schema defined in Employee case class
val employeeDF = sc.parallelize(...).toDF
```

Let's query this data set to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort our result in order of increasing employee ID.

What would this SQL query look like?

A More Interesting SQL Query

Let's assume we have a DataFrame representing a data set of employees:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)

// DataFrame with schema defined in Employee case class
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF
  = spark.sql("""SELECT id, lname
                 FROM employees
                WHERE city = "Sydney"
             ORDER BY id""")
```

registered "employees"

A More Interesting SQL Query

Let's assume we have a `DataFrame` representing a data set of employees:

```scala
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)

// DataFrame with schema defined in Employee case class
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF
  = spark.sql("""SELECT id, lname
                  FROM employees
                 WHERE city = "Sydney"
              ORDER BY id""")
```

Pretty simple.

A More Interesting SQL Query

Let's visualize the result on an example dataset.

Given:

```
val employeeDF = sc.parallelize(...).toDF

// employeeDF:           "employees"
// +---+-----+-------+---+--------+
// | id|fname|  lname|age|    city|
// +---+-----+-------+---+--------+
// | 12|  Joe|  Smith| 38|New York|
// |563|Sally|  Owens| 48|New York|
// |645|Slate|Markham| 28|  Sydney|
// |221|David| Walker| 21|  Sydney|
// +---+-----+-------+---+--------+
```

A More Interesting SQL Query

Let's visualize the result on an example dataset.

Given:

```
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF
  = spark.sql("""SELECT id, lname
                  FROM employees
                 WHERE city = "Sydney"
              ORDER BY id""")
```

Result

```
// employeeDF:
// +---+-----+-------+---+--------+
// | id|fname|  lname|age|    city|
// +---+-----+-------+---+--------+
// | 12|  Joe|  Smith| 38|New York|
// |563|Sally|  Owens| 48|New York|
// |645|Slate|Markham| 28|  Sydney|
// |221|David| Walker| 21|  Sydney|
// +---+-----+-------+---+--------+
```

```
sydneyEmployeesDF:
+---+-------+
| id|  lname|
+---+-------+
|221| Walker|
|645|Markham|
+---+-------+
```

Note: it's best to use Spark 2.1+ with Scala 2.11+ for doing SQL queries with Spark SQL.