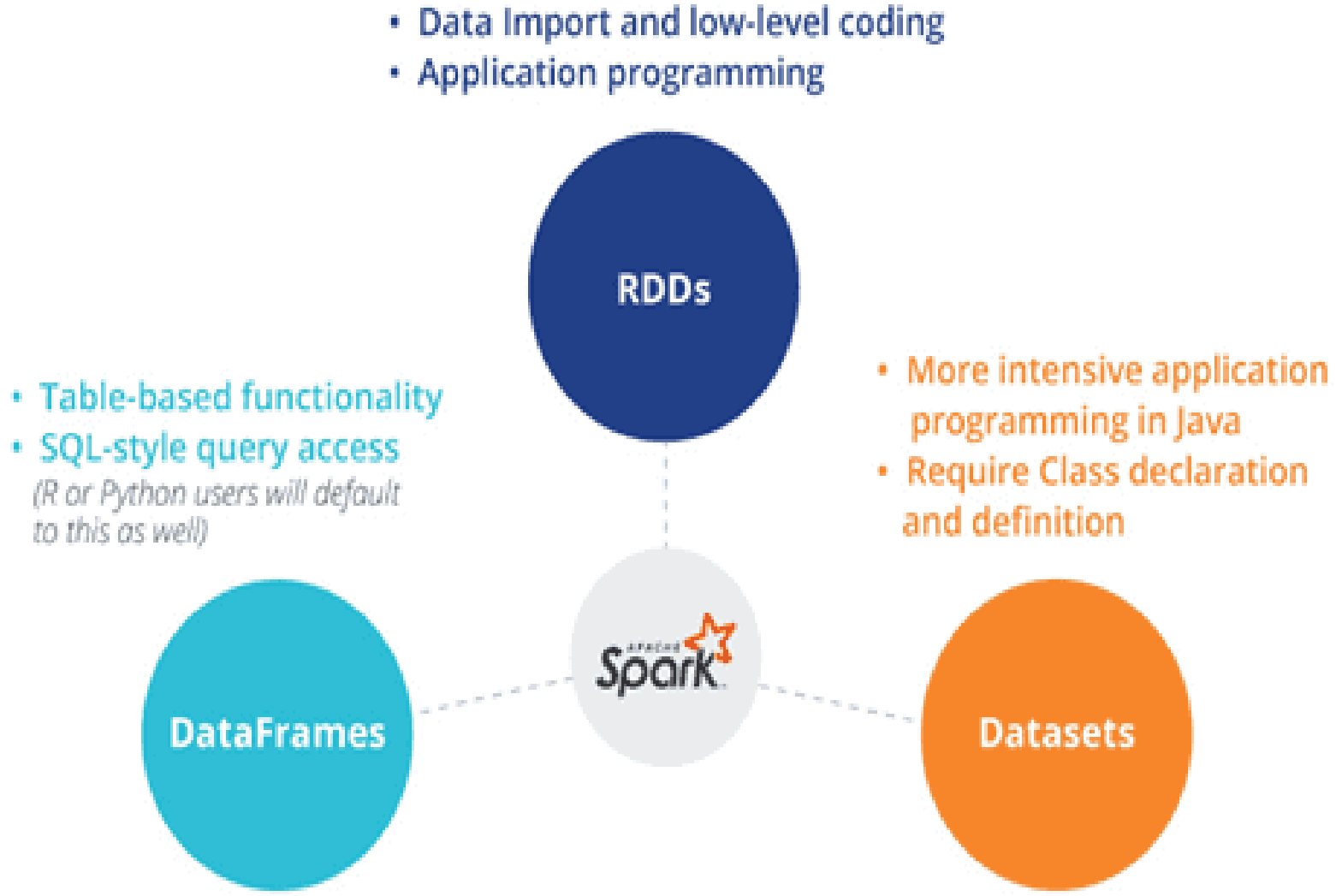
Spark SQL. DataFrame



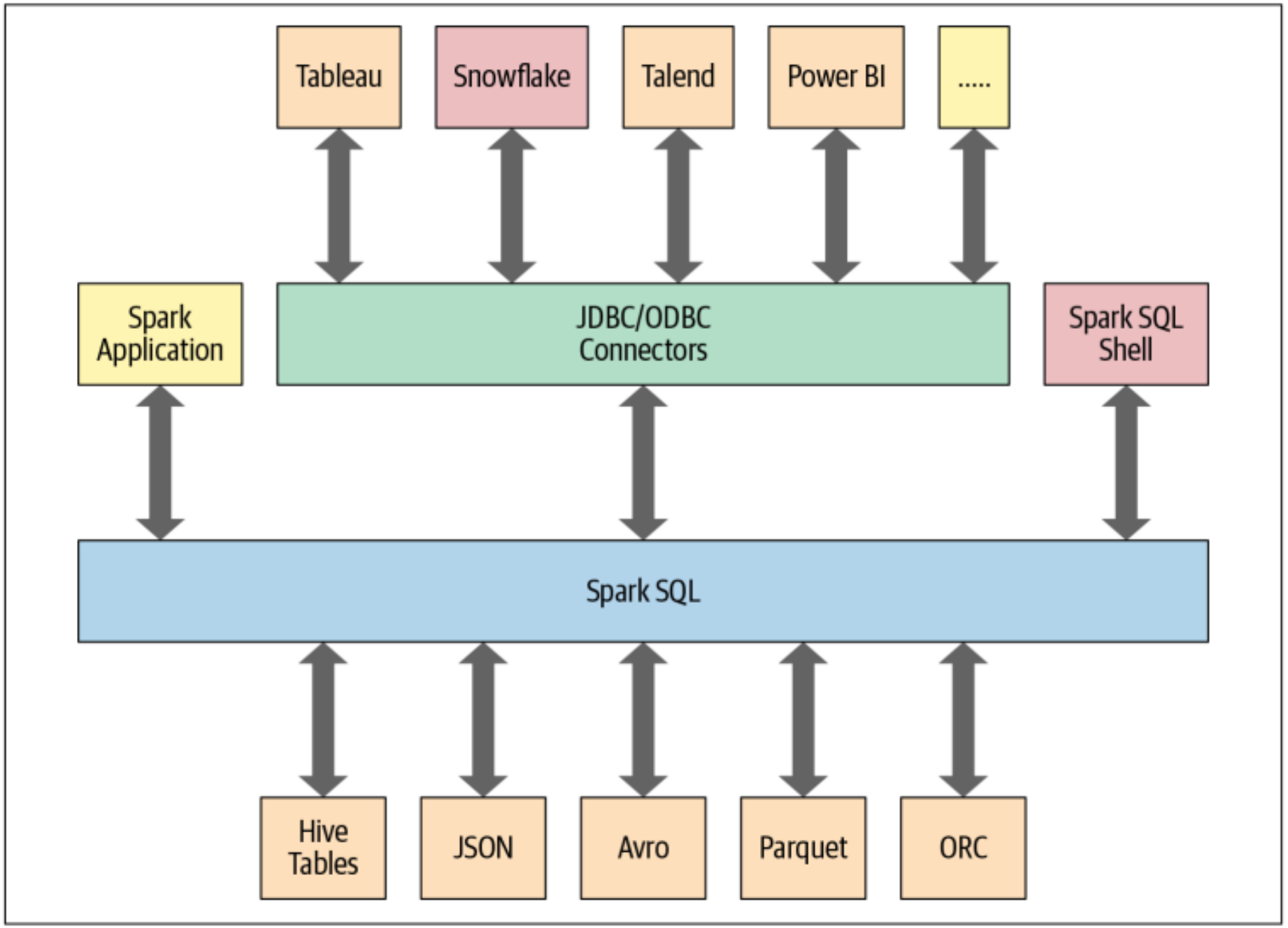
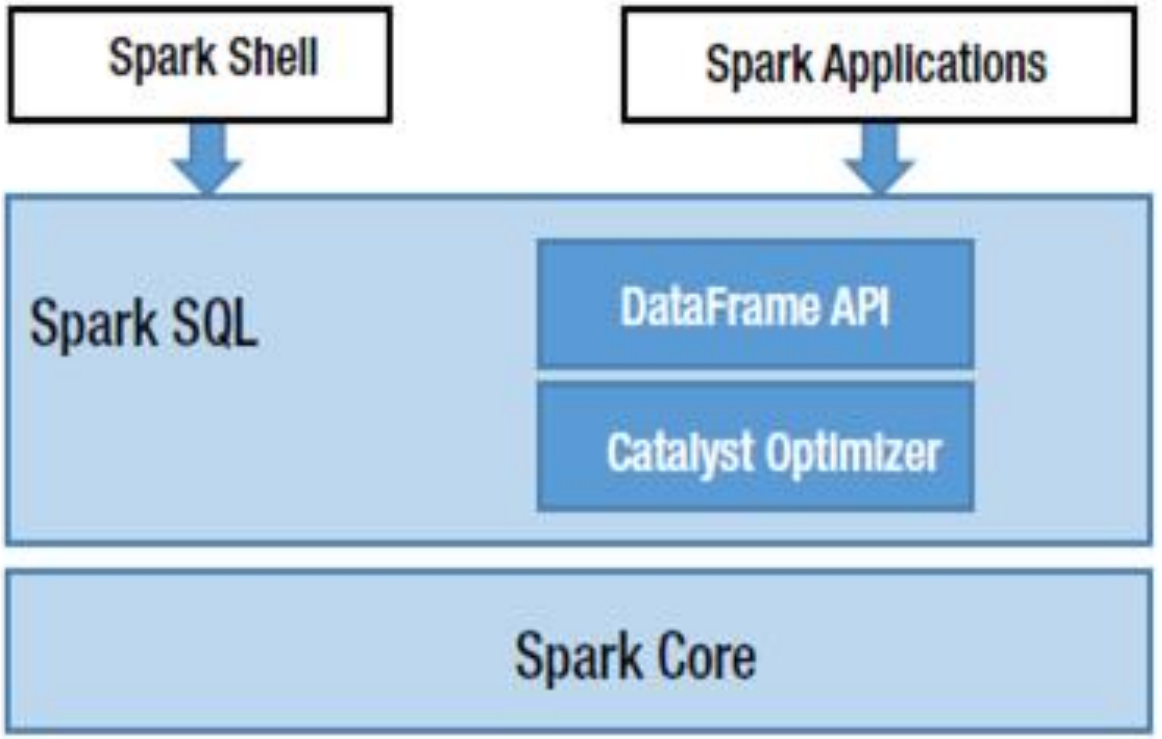


Tableau
Snowflake
Talend
Power BI
.....
Spark Application
JDBC/ODBC Connectors
Spark SQL Shell
Spark SQL
Hive Tables
JSON
Avro
Parquet
ORC

Spark SQL. DataFrame



Создание DataFrame из набора RDD

```
import scala.util.Random
val rdd = spark.sparkContext.parallelize(1 to 10).map(x => (x,
Random.nextInt(100)* x))
val kvDF = rdd.toDF("key","value")
```

Печать Schema и отображение данных DataFrame

```
kvDF.printSchema
|-- key: integer (nullable = false)
|-- value: integer (nullable = false)

kvDF.show
+---+-----+
|key|value|
+---+-----+
|  1|   58|
|  2|   18|
|  3|  237|
|  4|   32|
|  5|   80|
|  6|  210|
|  7|  567|
|  8|  360|
|  9|  288|
| 10|  260|
+---+-----+
```

Вызов функции show для отображения пяти строк в табличном формате

```
kvDF.show(5)
+---+-----+
|key|value|
+---+-----+
|  1|   59|
|  2|   60|
|  3|   66|
|  4|  280|
|  5|   40|
+---+-----+
```

Вызов функции show на дисплей 5 строк через табулятор

```
kvDF.show(5)
+---+-----+
|key|value|
+---+-----+
|  1|   59|
|  2|   60|
|  3|   66|
|  4|  280|
|  5|   40|
+---+-----+
```

Создание DataFrame из RDD с использованием Schema Created

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

val peopleRDD = spark.sparkContext.parallelize(Array(Row(1L, "John
Doe",  30L),
                                                    Row(2L, "Mary Jane", 25L)))

val schema = StructType(Array(
        StructField("id", LongType, true),
        StructField("name", StringType, true),
        StructField("age", LongType, true)
))

val peopleDF = spark.createDataFrame(peopleRDD, schema)
```

Displaying the Schema of peopleDF and Its Data

```
peopleDF.printSchema
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)

peopleDF.show
+--+-----------+---+
|id|       name|age|
+--+-----------+---+
| 1|   John Doe| 30|
| 2|  Mary Jane| 25|
+--+-----------+---+
```

Data Type	Scala Type
BooleanType	Boolean
ByteType	Byte
ShortType	Short
IntegerType	Int
LongType	Long
FloatType	Float
DoubleType	Double
DecimalType	java.math.BigDecial
StringType	String
BinaryType	Array[Byte]
TimestampType	java.sql.Timestamp
DateType	java.sql.Date
ArrayType	scala.collection.Seq
MapType	scala.collection.Map
StructType	org.apache.spark.sql.Row

Using the SparkSession.range Function to Create a DataFrame

```
val df1 = spark.range(5).toDF("num").show
+---+
|num|
+---+
|  0|
|  1|
|  2|
|  3|
|  4|
+---+
```

```
spark.range(5,10).toDF("num").show
+---+
|num|
+---+
|  5|
|  6|
|  7|
|  8|
|  9|
+---+
```

Converting a Collection Tuple to a DataFrame Using Spark's toDF Implicit

```
val movies = Seq(("Damon, Matt", "The Bourne Ultimatum", 2007L),
                 ("Damon, Matt", "Good Will Hunting", 1997L))

val moviesDF = movies.toDF("actor", "title", "year")

moviesDF.printSchema
|-- actor: string (nullable = true)
|-- title: string (nullable = true)
|-- year: long (nullable = false)

moviesDF.show
+----------+--------------------+----+
|      actor|               title|year|
+----------+--------------------+----+
|Damon, Matt|The Bourne Ultimatum|2007|
|Damon, Matt|   Good Will Hunting|1997|
+----------+--------------------+----+
```

Using read Variable from SparkSession

spark.read

Common Pattern for Interacting with DataFrameReader

spark.read.format(...).option("key", value").schema(...).load()

Три основных команды для DataFrameReader

Name	Optional	Comments
Format	No	Это может быть один из встроенных источников данных или пользовательский формат. Для встроенного формат, вы можете пользовать короткое имя (json, parquet, jdbc, orc, csv, text). Для настраиваемого источника данных необходимо указать полное имя.
option	Yes	DataFrameReader имеет набор параметров по умолчанию для каждого формата источника данных. Вы можете переопределить эти значения по умолчанию, указав значение для параметра функция.
Schema	Да	Некоторые источники данных имеют схему,

Specifying the Data Source Format

```
spark.read.json("<path>")
spark.read.format("json")
spark.read.parquet("<path>")
spark.read.format("parquet")
spark.read.jdbc
spark.read.format("jdbc")
spark.read.orc("<path>")
spark.read.format("orc")
spark.read.csv("<path>")
spark.read.format("csv")
spark.read.text("<path>")
spark.read.format("text")
// custom data source – fully qualifed package name
spark.read.format("org.example.mysource")
```

Spark's Built-in Data Sources

Name	**Data Format**	**Comments**
Text file	text	Не структурирован
CSV	text	Значения, разделенные запятыми. Это можно использовать для указания другого разделителя. На имя столбца можно ссылаться из заголовка.
JSON	text	Популярный полуструктурированный формат. Имя столбца и тип данных определяются автоматически.
Parquet	Binary	(Формат по умолчанию.) Популярный двоичный формат в сообществе Hadoop.

Reading the README.md File As a Text File from a Spark Shell

```
val textFile = spark.read.text("README.md")

textFile.printSchema
|-- value: string (nullable = true)

// show 5 lines and don't truncate
textFile.show(5, false)
```

Table 4-4. *CSV Common Options*

Key	Values	Default	Description
`sep`	Single character	`,`	This is a single-character value used as a delimiter for each column.
`header`	true, false	`false`	If the value is true, it means the first line in the file represents the column names.
`escape`	Any character	`\`	This is the character to use to escape the character in the column value that is the same as `sep`.
`inferSchema`	true, false	`false`	This specifies whether Spark should try to infer the column type based on column value.

```
val movies =
spark.read.option("header","true").csv("<path>/book/chapter4/

data/movies/movies.csv")

movies.printSchema

|-- actor: string (nullable = true)

|-- title: string (nullable = true)

|-- year: string (nullable = true)

// now try to infer the schema

val movies2 = spark.read.option("header","true").

option("inferSchema","true")

.csv("<path>/book/chapter4/data/movies/movies.csv")

movies2.printSchema

|-- actor: string (nullable = true)

|-- title: string (nullable = true)
```

```
// now try to manually provide a schema
import org.apache.spark.sql.types._
val movieSchema = StructType(Array(StructField("actor_name", StringType, true
StructField("movie_title", StringType, true),
StructField("produced_year", LongType, true)))
val movies3 = spark.read.option("header","true").schema(movieSchema)
.csv("<path>/book/chapter4/data/movies/
movies.csv")
movies3.printSchema
|-- actor_name: string (nullable = true)
|-- movie_title: string (nullable = true)
|-- produced_year: long (nullable = true)
movies3.show(5)
+----------------+----
```

Reading a TSV File with the CSV Format

```
val movies4 = spark.read.option("header","true").option("sep", "\t")
    .schema(movieSchema).csv("<path>/book/chapter4/data/movies/movies.tsv")

movies.printSchema
|-- actor_name: string (nullable = true)
|-- movie_title: string (nullable = true)
|-- produced_year: long (nullable = true)
```

Операции	Описание
select	выбирает один или несколько столбцов из существующего набора столбцов в DataFrame. В процессе преобразования столбцы можно преобразовывать и переставлять.
selectExpr	поддерживает мощные выражения SQL при преобразовании столбцов, пока выполнение проекции.
filter	Оба filter и where имеют одинаковую семантику. where более реляционный чем filter, и он аналогичен условию where в SQL. Они оба используется для фильтрации строк на основе заданных логических условий.
where	
Distinct dropDuplicates	удаляет повторяющиеся строки из DataFrame.
sortorderBy	сортирует DataFrame по предоставленным столбцам.
limit	возвращает новый DataFrame, беря первые n строк.
union	объединяет два DataFrame и возвращает их как новый DataFrame.
withColumn	Используется для добавления нового столбца или замены существующего столбца в DataFrame.
withColumnRenamed	переименовывает существующий столбец. Если заданное имя столбца не существует в Schema, то это не работает.
drop	удаляет один или несколько столбцов из DataFrame. Эта операция делает ничего, если указанное имя столбца не существует.
sample	случайным образом выбирает набор строк на основе заданного параметра фракции, необязательное начальное значение и необязательный вариант замены.
randomSplit	разбивает DataFrames на один или несколько DataFrames на основе заданного веса. Обычно он используется для разделения набора основных данных на обучение и наборы тестовых данных в процессе обучения модели машинного обучения.
join	объединяет два DataFrames. Spark поддерживает множество типов объединений.
groupBy	Группирует DataFrame по одному или нескольким столбцам. Распространенным шаблоном является выполнение некоторой агрегации после операции groupBy.
describe	вычисляет общую статистику о числовых и строковых столбцах в DataFrame. Доступная статистика: количество, среднее, стандартное отклонение, минимальное, максимальное и произвольные приблизительные процентили.

Different Ways of Referring to a Column

Way	Example	Description
""	"columName"	This refers to a column as a string type.
col	col("columnName")	The col function returns an instance of the Column class.
column	column("columnName")	Similar to col, this function returns an instance of the Column class.
$	$"columnName"	This is a syntactic sugar way of constructing a Column class in Scala.
' (tick mark)	'columnName	This is a syntactic sugar way of constructing a Column class in Scala by leveraging the Scala symbolic literals feature.

Различные способы ссылки на столбец

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

val kvDF = Seq((1,2),(2,3)).toDF("key","value")

// to display column names in a DataFrame, we can call the columns function
kvDF.columns
Array[String] = Array(key, value)

kvDF.select("key")
kvDF.select(col("key"))

kvDF.select(column("key"))
kvDF.select($"key")
kvDF.select('key)

// using the col function of DataFrame
kvDF.select(kvDF.col("key"))
```

```
kvDF.select('key, 'key > 1).show
+---+----------+
|key| (key > 1)|
+---+----------+
|  1|     false|
|  2|      true|
+---+----------+
```

Два варианта выбранной трансформации

```
movies.select("movie_title","produced_year").show(5)
+------------------+-------------+
|       movie_title| produced_year|
+------------------+-------------+
|      Coach Carter|         2005|
|       Superman II|         1980|
|         Apollo 13|         1995|
|          Superman|         1978|
| Back to the Future|         1985|
+------------------+-------------+
```

```
// using a column expression to transform year to decade
movies.select('movie_title,('produced_year - ('produced_year % 10)).
as("produced_decade")).show(5)

+------------------+---------------+
|       movie_title| produced_decade|
+------------------+---------------+
|      Coach Carter|           2000|
|       Superman II|           1980|
|         Apollo 13|           1990|
|          Superman|           1970|
| Back to the Future|           1980|
+------------------+---------------+
```

```
movies.selectExpr("*","(produced_year - (produced_year % 10)) as decade").
show(5)
+----------------+-------------------+-------------+-------+
|      actor_name|        movie_title| produced_year| decade|
+----------------+-------------------+-------------+-------+
|McClure, Marc (I)|       Coach Carter|          2005|   2000|
|McClure, Marc (I)|        Superman II|          1980|   1980|
|McClure, Marc (I)|          Apollo 13|          1995|   1990|
|McClure, Marc (I)|           Superman|          1978|   1970|
|McClure, Marc (I)| Back to the Future|          1985|   1980|
+----------------+-------------------+-------------+-------+
```

Добавление столбца десятилетия в DataFrame фильмов с использованием SQL выражение

```
movies.selectExpr("*","(produced_year - (produced_year % 10)) as decade").
show(5)
```

```
+----------------+-------------------+--------------+-------+
|      actor_name|        movie_title| produced_year| decade|
+----------------+-------------------+--------------+-------+
|McClure, Marc (I)|       Coach Carter|          2005|   2000|
|McClure, Marc (I)|        Superman II|          1980|   1980|
|McClure, Marc (I)|          Apollo 13|          1995|   1990|
|McClure, Marc (I)|           Superman|          1978|   1970|
|McClure, Marc (I)| Back to the Future|          1985|   1980|
+----------------+-------------------+--------------+-------+
```

```
movies.selectExpr("count(distinct(movie_title)) as
movies","count(distinct(actor_name)) as actors").show
+------+-------+
| movies| actors|
+------+-------+
```

Фильтрация строк с помощью функций логического сравнения в классе столбцов

```
movies.filter('produced_year < 2000)
movies.where('produced_year > 2000)

movies.filter('produced_year >= 2000)
movies.where('produced_year >= 2000)
```

```
// equality comparison require 3 equal signs
movies.filter('produced_year === 2000).show(5)
```

```
// inequality comparison uses an interesting looking operator =!=
movies.select("movie_title","produced_year").filter('produced_year =!=
2000).show(5)
```

```
// to combine one or more comparison expressions, we will use either the OR
and AND expression operator
movies.filter('produced_year >= 2000 && length('movie_title) < 5).show(5)
```

sort(columns), orderBy(columns)

```
movies.select("movie_title").distinct.selectExpr("count(movie_title) as
movies").show
movies.dropDuplicates("movie_title").selectExpr("count(movie_title) as
movies").show
```

```
val movieTitles = movies.dropDuplicates("movie_title")
                        .selectExpr("movie_title", "length(movie_title) as
                        title_length", , "produced_year")
```

```
movieTitles.sort('title_length).show(5)
```

```
// sorting in descending order
movieTitles.orderBy('title_length.desc).show(5)
```

```
// sorting by two columns in different orders
movieTitles.orderBy('title_length.desc, 'produced_year).show(5)
```

Commonly Used Structured Actions

Operation	Description

`show()`
`show(numRows)`
`show(truncate)`
`show(numRows, truncate)`
	Displays a number of rows in a tabular format. If `numRows` is not specified, it will show the top 20 rows. The `truncate` option controls whether to truncate the string column if it is longer than 20 characters.

`head()`
`first()`
`head(n)`
`take(n)`
	Returns the first row. If `n` is specified, then it will return the first `n` rows. `first` is an alias for `head`. `take(n)` is an alias for `first(n)`.

`takeAsList(n)`	Returns the first `n` rows as a Java list. Be careful not to take too many rows; otherwise, it may cause an out-of-memory error on the application's driver process.
`collect` `collectAsList`	Returns all the rows as an array or Java list. Apply the same caution as the one described in the `takeAsList` action.
`count`	Returns the number of rows in a DataFrame.

Dataset

```
// define Movie case class
case class Movie(actor_name:String, movie_title:String, produced_year:Long)

// convert DataFrame to strongly typed Dataset
val moviesDS = movies.as[Movie]

// create a Dataset using SparkSession.createDataset() and the toDS
implicit function
val localMovies = Seq(Movie("John Doe", "Awesome Movie", 2018L),
                      Movie("Mary Jane", "Awesome Movie", 2018L))

val localMoviesDS1 = spark.createDataset(localMovies)
val localMoviesDS2 = localMovies.toDS()
localMoviesDS1.show
```

Результат

```
+----------+--------------+--------------+
|actor_name|   movie_title| produced_year|
+----------+--------------+--------------+
|  John Doe| Awesome Movie|          2018|
| Mary Jane| Awesome Movie|          2018|
+----------+--------------+--------------+
```

Manipulating a Dataset in a Type-Safe Manner

```
// filter movies that were produced in 2010 using
            ter(movie => movie.produced_year == 2010).show(5)
+------------------+--------------------+--------------+
|        actor_name|         movie_title| produced_year|
+------------------+--------------------+--------------+
|  Cooper, Chris (I)|            The Town|          2010|
|    Jolie, Angelina|                Salt|          2010|
|    Jolie, Angelina|         The Tourist|          2010|
|     Danner, Blythe|       Little Fockers|          2010|
| Byrne, Michael (I)| Harry Potter and ...|          2010|
+------------------+--------------------+--------------+
```

```
// displaying the title of the first movie in the moviesDS

moviesDS.first.movie_title

String = Coach Carter

// try with misspelling the movie_title and get compilation error

moviesDS.first.movie_tile

error: value movie_tile is not a member of Movie

// perform projection using map transformation

val titleYearDS = moviesDS.map(m => ( m.movie_title,
m.produced_year))

titleYearDS.printSchema

|-- _1: string (nullable = true)

|-- _2: long (nullable = false)
```

```
// demonstrating a type-safe transformation that fails at compile time,

performing subtraction on a column with string type

// a problem is not detected for DataFrame until runtime

movies.select('movie_title - 'movie_title)

// a problem is detected at compile time

moviesDS.map(m => m.movie_title - m.movie_title)

error: value - is not a member of String

// take action returns rows as Movie objects to the driver

moviesDS.take(5)

Array[Movie] = Array(Movie(McClure, Marc (I),Coach Carter,2005),

Movie(McClure, Marc (I),Superman II,1980), Movie(McClure, Marc (I),Apollo

13,1995))
```