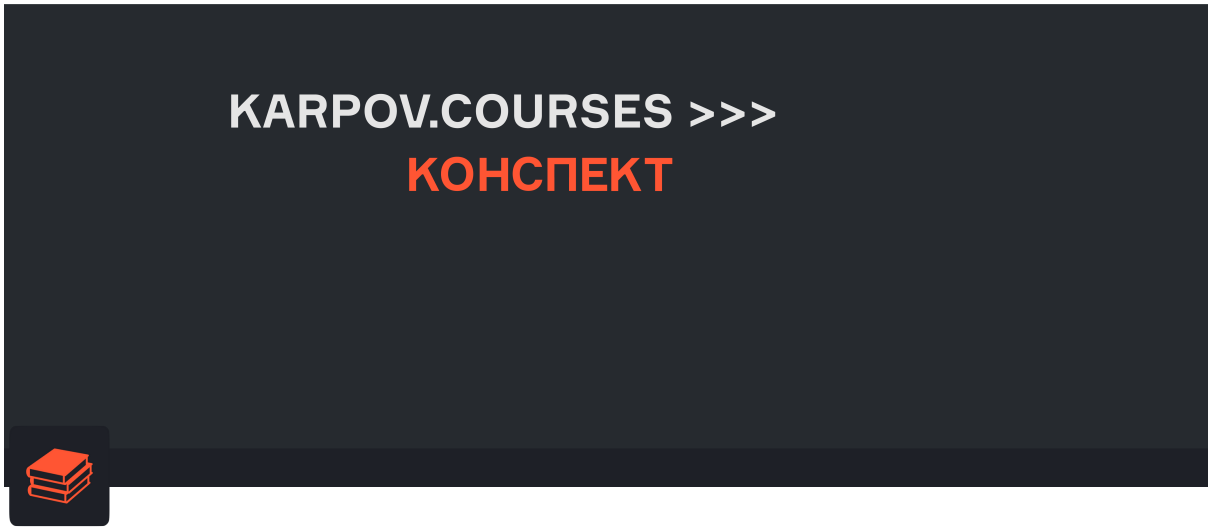


> Конспект > 6 урок > Spark SQL

> Оглавление

> Spark SQL

Spark SQL – это компонент фреймворка Apache Spark для структурированной обработки данных.

Spark SQL позволяет работать со структирированными и полуструкированными данными.

В Spark SQL основные два интерфейса:

- DataFrame
- Dataset. Доступен, если вы пишите на Scala/Java.

> DataFrame

DataFrame - это распределенная коллекция данных, организованных посредством именованных столбцов. Является абстракцией поверх RDD. Данная абстракция

предназначена для выборки, фильтрации, агрегации и визуализации структурированных данных.

Что входит в DataFrame:

- Схема данных, которая состоит из:
  - имен полей
  - типизации
- Оптимизатор (Catalyst). Может работать в двух режимах (**rule-based** и **cost-based**). Их работа не противоречит друг другу.
- Собственный DSL. С помощью которого можно расширять функционал DataFrame.
- Поддержка SQL-like способа обработки данных. Обращение к DataFrame как к обычной БД через SQL-запрос.
- Поддержка большего кол-ва форматов и БД. Реализовано через:
  - **DataFrame Reader**
  - **DataFrame Writer**

DataFrame schema

DataFrame schema - объект, который описывает массив структурированных полей данных хранящихся в DataFrame. Данный объект похож на case класс в Scala и на класс data в Pyhton.

Каждое из полей обладает 4 атрибутами:

1. name. Название поля.
2. dataType. Тип поля.
3. nullable: Boolean. Может ли быть поле NULL.
4. metadata. Доп. информация. На практике используется редко.

```
case class StructType(fields: Array[StructField])

case class StructField(
    name: String,
    dataType: DataType,
    nullable: Boolean = true,
    metadata: Metadata = Metadata.empty)
```

Рассмотрим пример данных записаных в json-файл. И выведем его структуры с помощью метода `.printSchema()`.

```
hadoop fs -cat test.json
{"name": "Anton", "surname": "Pilipenko", "phones": {"mobile": "+7903XXXXXXX", "work": "+74950000000"}}
{"name":  "Doug", "surname": "Cutting", "country":  "USA"}
```

```
spark.read.json('test.json').printSchema()

root
 |-- country: string (nullable = true)
 |-- name: string (nullable = true)
 |-- phones: struct (nullable = true)
 |    |-- mobile: string (nullable = true)
 |    |-- work: string (nullable = true)
 |-- surname: string (nullable = true)
```

Spark может автоматически распознавать структуру данных. Делает это он не на всех данных, а только на части. В первом примере был рассмотрен автоматический вывод структуры DataFrame.

Теперь явно укажем структуру данных. Этот способ является рекомендуемым, т.к. вы четко задаете структуру ваших данных и тем самым избежите ошибок.

```
from pyspark.sql.types import StringType, StructType, StructField

df_schema = StructType([
    StructField("name", StringType()),
    StructField("surname", StringType()),
    StructField("country", StringType()),
    StructField("phones", StructType([
        StructField("work", StringType()),
        StructField("mobile", StringType())
    ]))
])

spark.read.schema(df_schema).json('test.json').printSchema()

root
 |-- country: string (nullable = true)
 |-- name: string (nullable = true)
 |-- phones: struct (nullable = true)
 |    |-- mobile: string (nullable = true)
 |    |-- work: string (nullable = true)
 |-- surname: string (nullable = true)
```

Создание DataFrame

Варианты создания DataFrame:

- Из существующей RDD
- Из Spark Data Sources:
  - Файла. Например, csv, json, ORC, Parquet, любой формат через newAPIHadoopFile (требуется InputFormat).
  - Любые реляционной БД поддерживаемые JDBC драйвер.
  - Hive.

Базовые методы DataFrame

- `select`
- `filter` (where)
- `count`
- `distinct`
- `agg` + `groupBy`
- `join`
- `limit`
- `orderBy`
- `withColumn` / `drop`
- `withColumnRenamed`

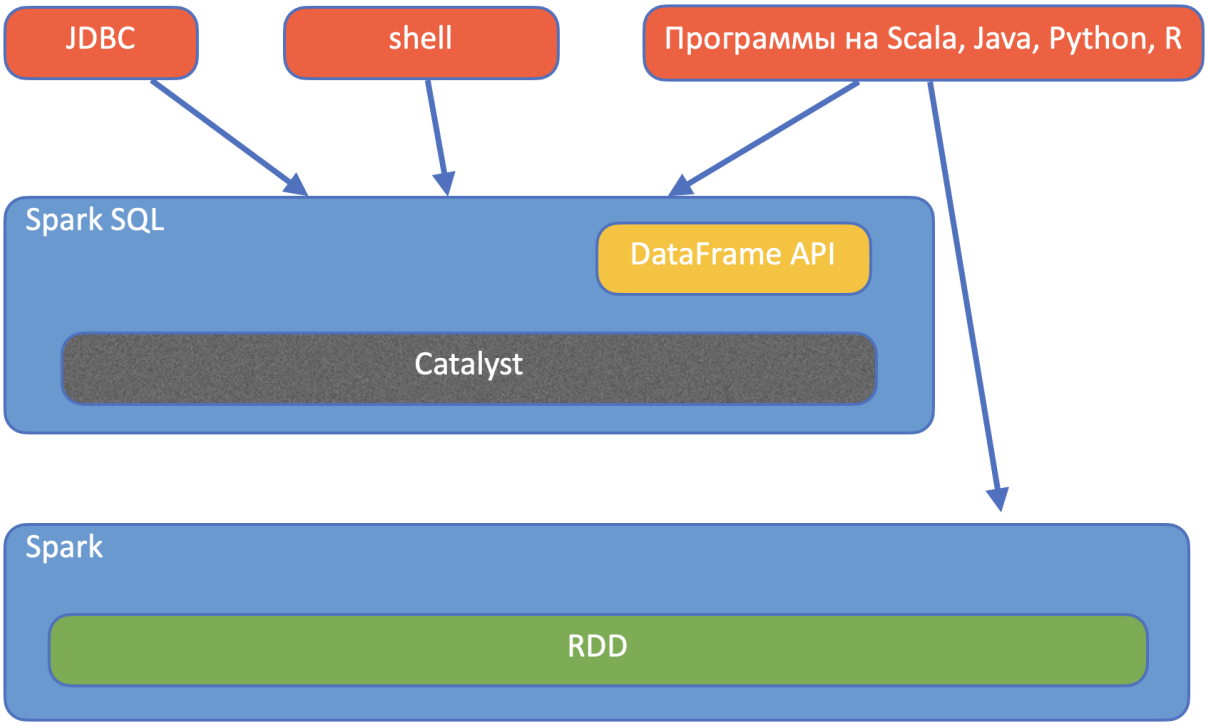
> Пример работы с DataFrame

```
from pyspark.sql import functions as F

data.where(
    (F.col('name') == 'Anton') &
    (~F.isnull(F.col('phones.work')))
).select(F.col('name'), F.col('surname'), F.col('country')).show()

+-----+---------+-------+
| name|  surname|country|
+-----+---------+-------+
|Anton|Pilipenko|   null|
+-----+---------+-------+
```

Тот же самый запрос, только через поддержку SQL.

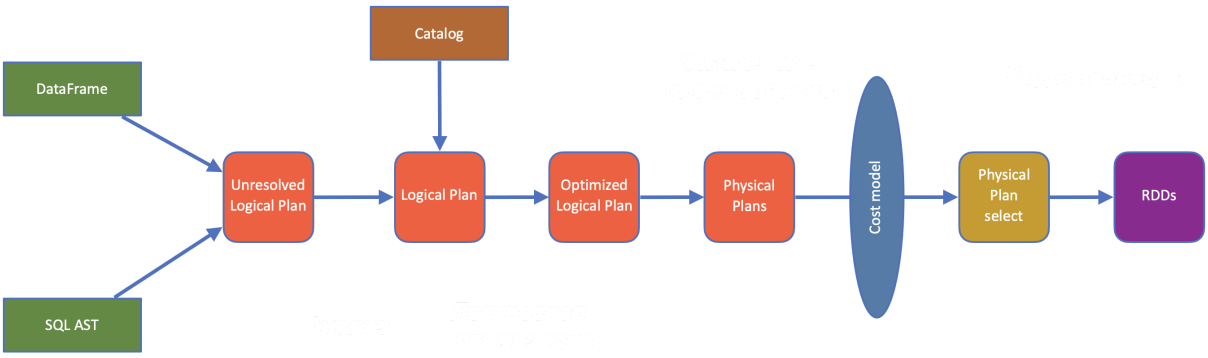
```
data.registerTempTable('test_table')


spark.sql("""select name,
                    surname,
                    country
               from test_table
              where name = 'Anton'
                and phones.work is not null""").show()

+-----+---------+-------+
| name|  surname|country|
+-----+---------+-------+
|Anton|Pilipenko|   null|
+-----+---------+-------+
```

> Схема интерфейса Spark SQL



> Оптимизатор Catalyst

Оптимизатор - это сущность, которая берет запрос к данным, разбивает его на шаги, и анализируя их выполняет этот запрос оптимально и эффективно.



Сначала формируется Unresolved Logical Plan, который раскладывает наш запрос в последовательность действий. Далее происходит анализ. На этом этапе оптимизатор пытается понять насколько наш план валиден и что каждая из сущностей в нем обозначает. Для поиска информации о сущностях используется Catalog. Каталог хранит информацию о сущностях, их структурах и взаимосвязях. По результатам анализа появляется Logical Plan, который после логической оптимизации становится Optimized Logical Plan. Далее происходит физическое планирование, где мы получаем какое-то кол-во Physical Plans, как правило это не один план. Потом Cost model оценивает каждый Physical Plan и решает какой план будет выбран. Выбранный план называется Selected Physical Plan. На его основе происходит кодогенерация, где выбранный план переводится на язык Scala и далее передается на уровень RDD для получения результата.

Catalyst анализ

Что происходит на стадии анализа? Catalyst выводит тип атрибута, т.к. например, могут быть ситуации когда у нас есть одинаковые поля в разных DataFrame, которые имеют разные типы данных (int и float). Catalyst делает допущение, в котором он считает каждый тип атрибута unresolved. После catalyst начинает идти в каждый источник атрибута и выводить его тип, если в источнике типа нет, catalyst проверяет не указан ли был тип заранее в запросе. В случае если тип не найден в источнике и не был указан заранее - тип остается unresolved.

Логическая оптимизация

Логическая оптимизация - это когда типовые правила применяются к логическому плану.

Типы логических оптимизаций:

- constant folding (свертка).

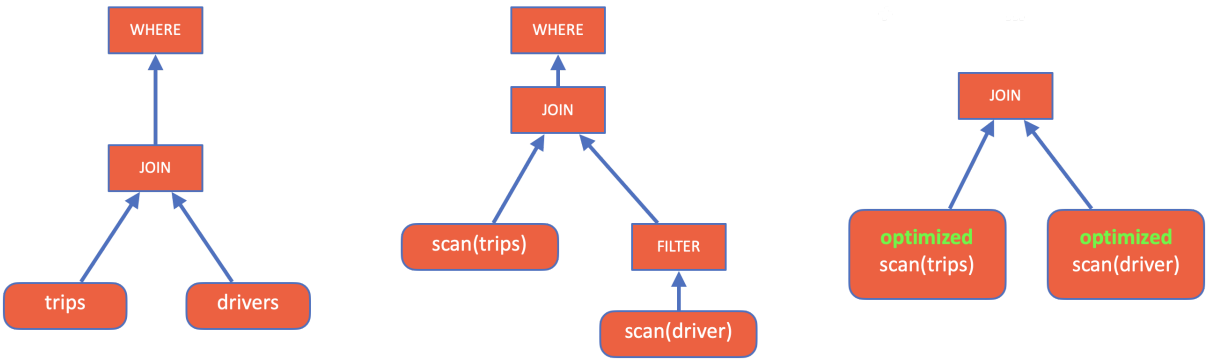
```
spark.range(1).select(lit(3) > 2).explain(true)
...
TRACE SparkOptimizer:
=== Applying Rule org.apache.spark.sql.catalyst.optimizer.ConstantFolding ===
!Project [(3 > 2) AS (3 > 2)#3]            Project [true AS (3 > 2)#3]
 +- Range (0, 1, step=1, splits=Some(8))   +- Range (0, 1, step=1, splits=Some(8))
```

- predicate pushdown (предикатное сжатие). Например добавление where условия для JDBC источника.
- empty relation propagation. Например, если у нас есть пустой DataFrame и с ним происходит join - происходит замена на пустой DataFrame.
- boolean expression simplification.
- и т.д.

Физическая оптимизация

Когда один или несколько физических планов формируются из логического с использованием физического оператора, соответствующего движку Apache Spark. Итоговый план выбирается на основе стоимостной модели (CBO, Cost-based optimization). CBO позволяет оценить стоимость рекурсивно для всего дерева с помощью правил. Физическая оптимизация на основе правил (RBO, Rule-based Optimization), такая как конвейерные проекции или map-фильтры в Spark также выполняется физическим планировщиком. Помимо этого, он может передавать операции из логического плана в источники данных, которые поддерживают сжатие предикатов или проекций.

```
select sum(t.total_amount) as total_income,
       d.name
 from trips as t join drivers as d
where t.driver_id = d.id
  and d.name = 'Smith'
```


WHERE
JOIN
trips
drivers
WHERE
JOIN
scan(trips)
FILTER
scan(driver)
JOIN
optimized
scan(trips)
optimized
scan(driver)