# Урок 2. DataFrame API

# Cluster manager

## Распределяет ресурсы между spark приложениями

**standalone cluster manager**

FIFO исполнение приложений

**Apache Hadoop YARN**

стандартное решение

распределяет, освобождает ресурсы между различными приложениями

Часть Hadoop

**Apache Mesos**

YARN-like, but большая изолированность процессов

поддерживает не-hadoop приложения

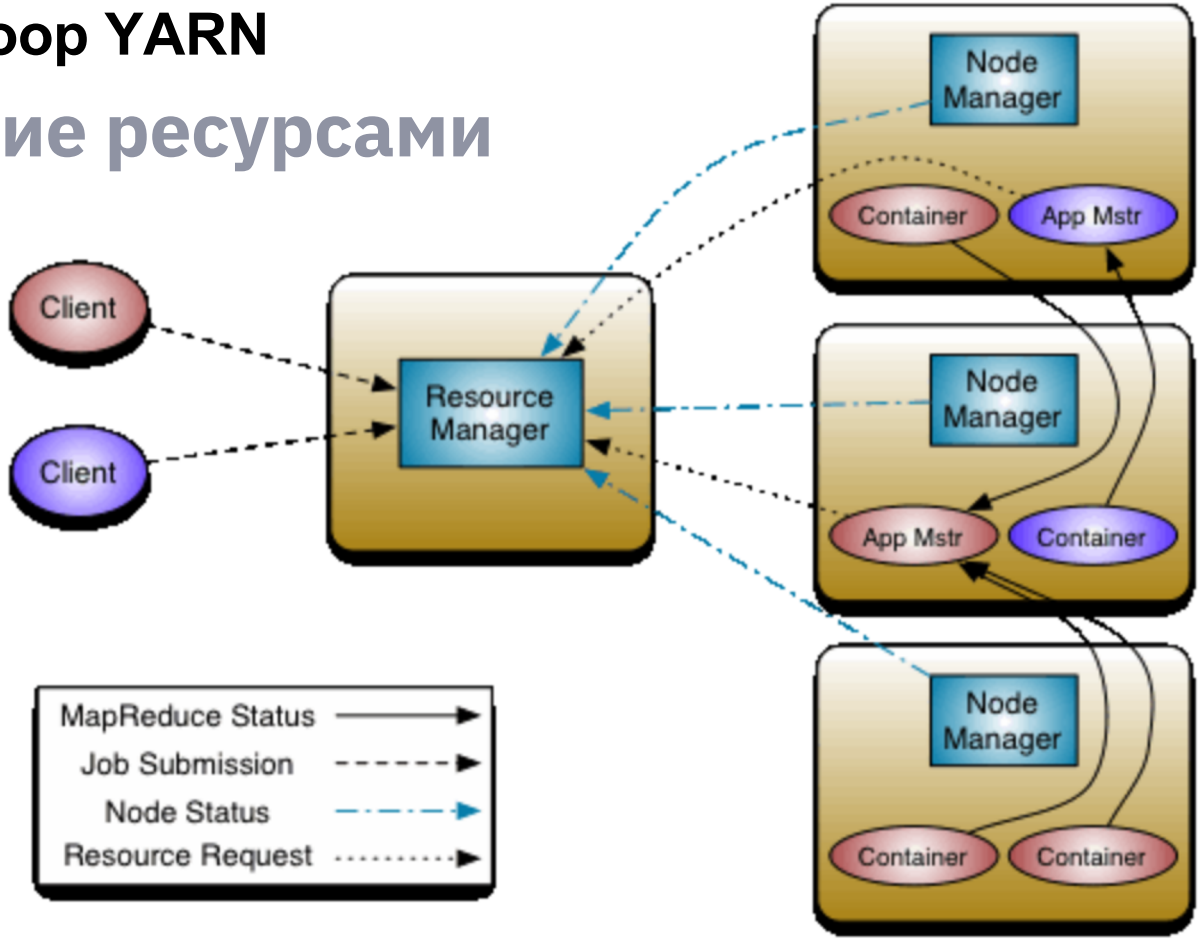
**Kubernetes**

запуск в контейнерах = абсолютная изолированность

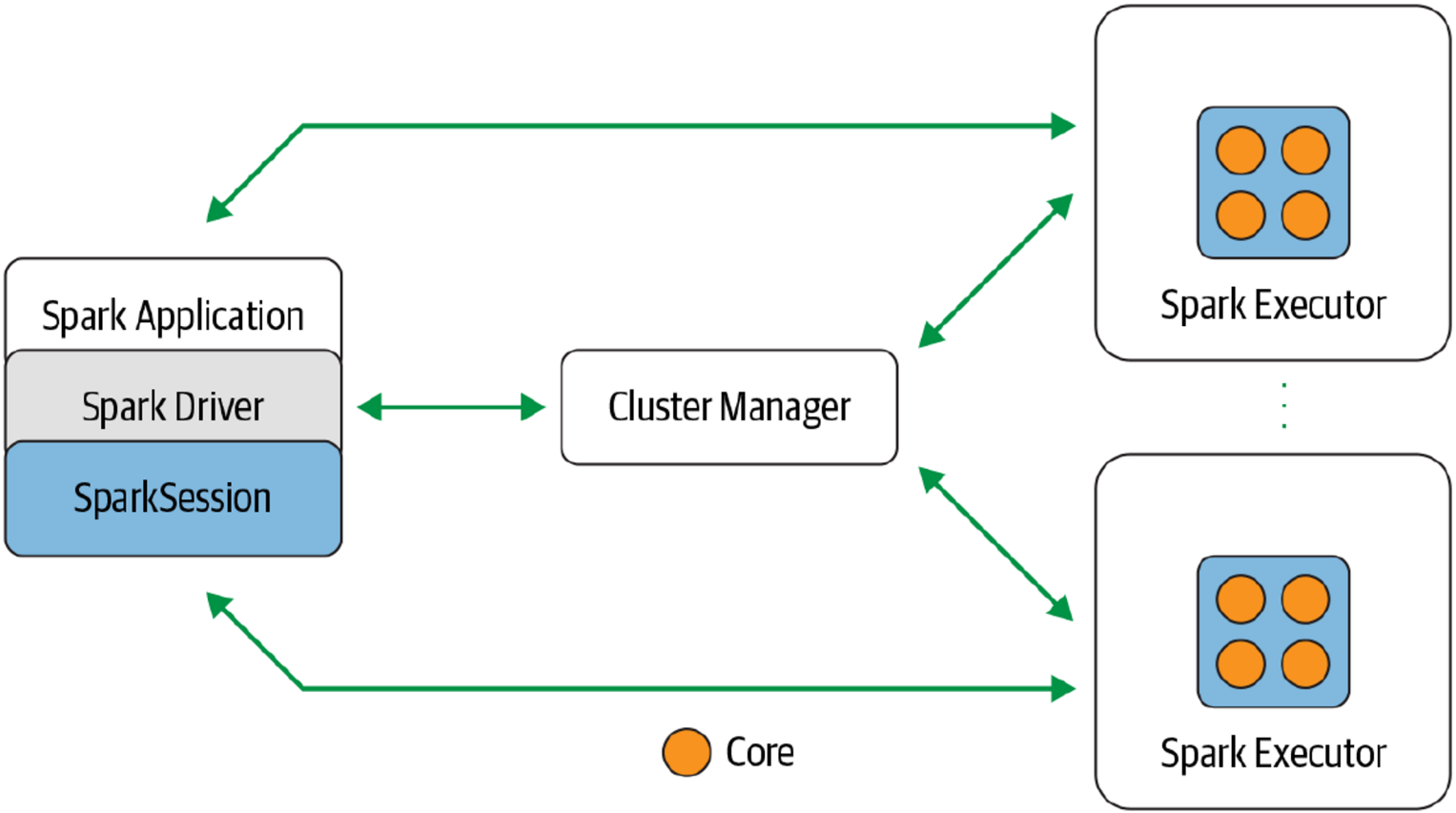
существенно сложнее в поддержке

# Apache Hadoop YARN

## Управление ресурсами

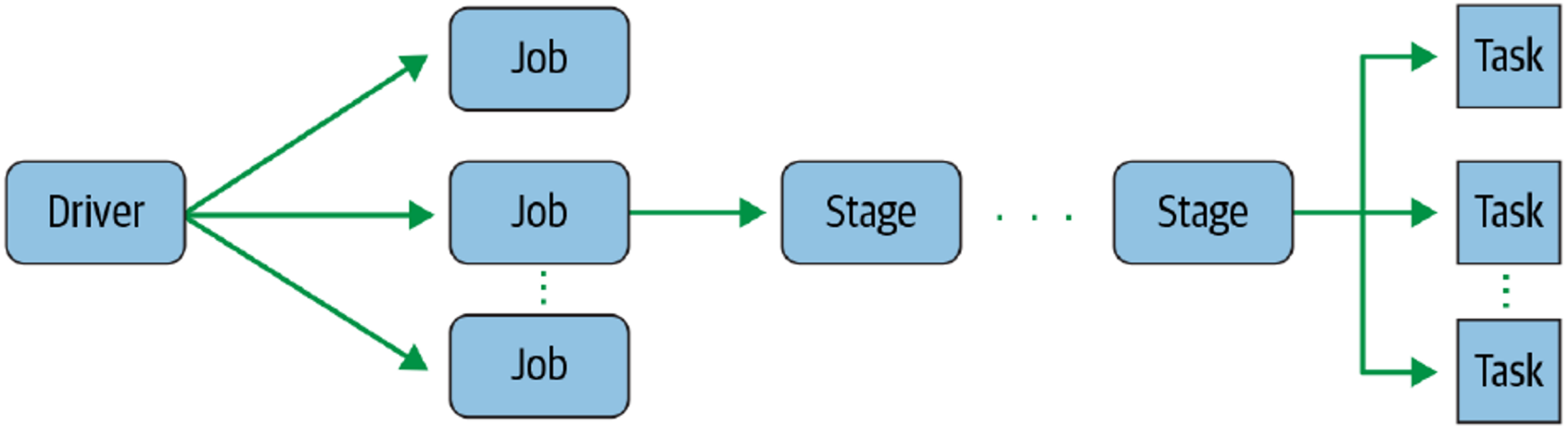


# Архитектура приложения Spark



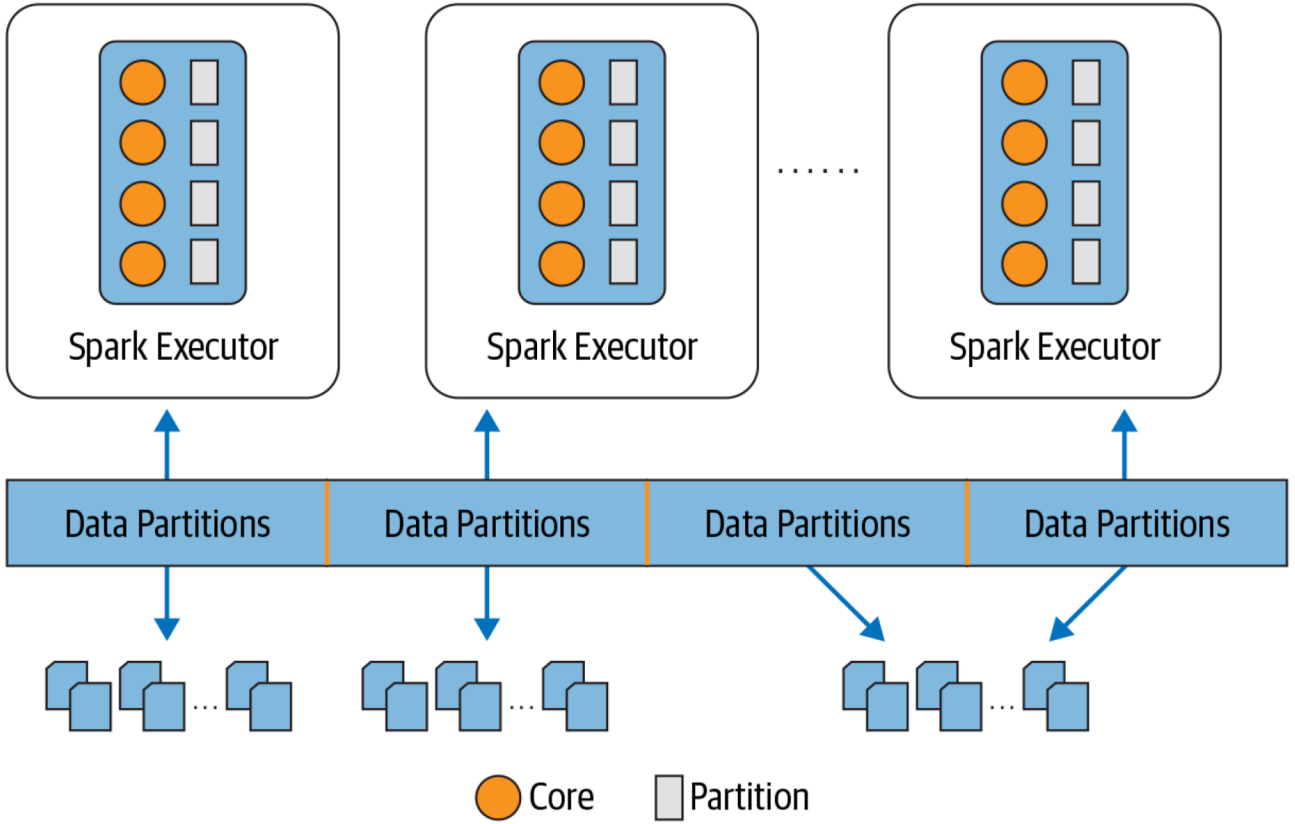
# Spark jobs, stage, task

## shuffle occurs between every two stages



# Spark executors, partitions

## Relationship of Spark tasks, cores, partitions



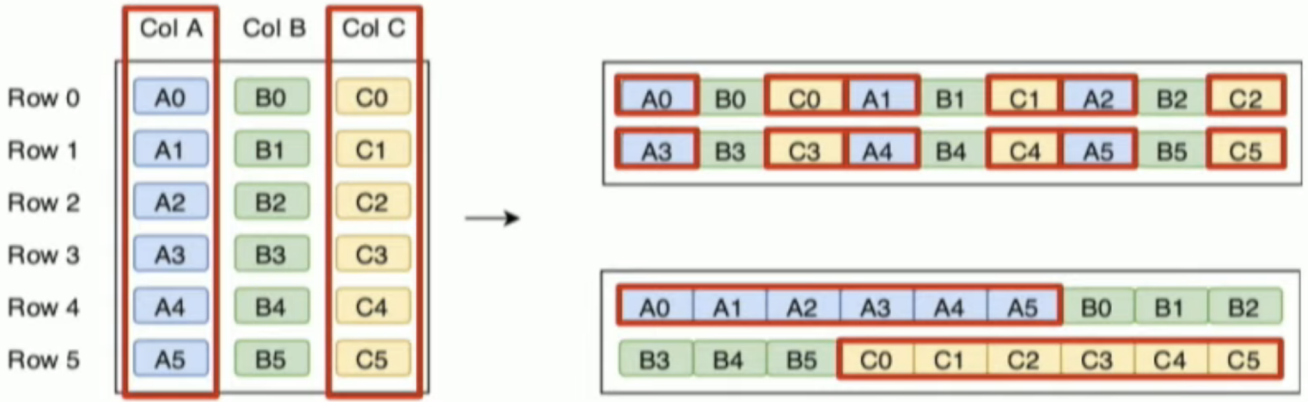
# Parquet vs CSV

## Меньше размер, быстрее чтение

The following table compares the savings as well as the speedup obtained by converting data into Parquet from CSV.

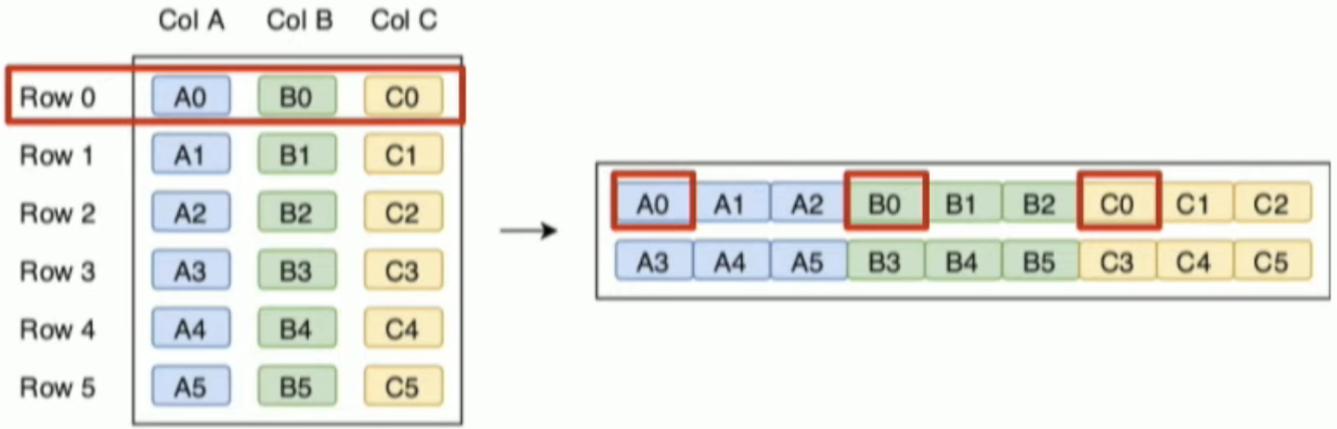
| Dataset | Size on Amazon S3 | Query Run Time | Data Scanned | Cost |
|---|---|---|---|---|
| Data stored as CSV files | 1 TB | 236 seconds | 1.15 TB | $5.75 |
| Data stored in Apache Parquet Format | 130 GB | 6.78 seconds | 2.51 GB | $0.01 |
| Savings | 87% less when using Parquet | 34x faster | 99% less data scanned | 99.7% savings |

# Row-wise vs Columnar




https://www.youtube.com/watch?v=1j8SdS7s_NY

# Hybrid




- Horizontal & vertical partitioning
- Used by Parquet & ORC
- Best of both worlds

# Типичная batch задача на Spark

1. Чтение (через объект SparkSession)
2. Преобразование данных (RDD, DataFrame, Dataset)
3. Сохранение

Источниками для чтения/записи данных могут быть:

a. файл csv, parquet, xlsx на hdfs
b. внешняя реляционная база, через jdbc драйвер
c. hive metastore -- чтение таблицы по имени schema.tablename

# RDD

## низкоуровневый API для работы с распределенными данными

```
# In Python
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average
agesRDD = (dataRDD
.map(lambda x: (x[0], (x[1], 1)))
.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
.map(lambda x: (x[0], x[1][0]/x[1][1])))
```

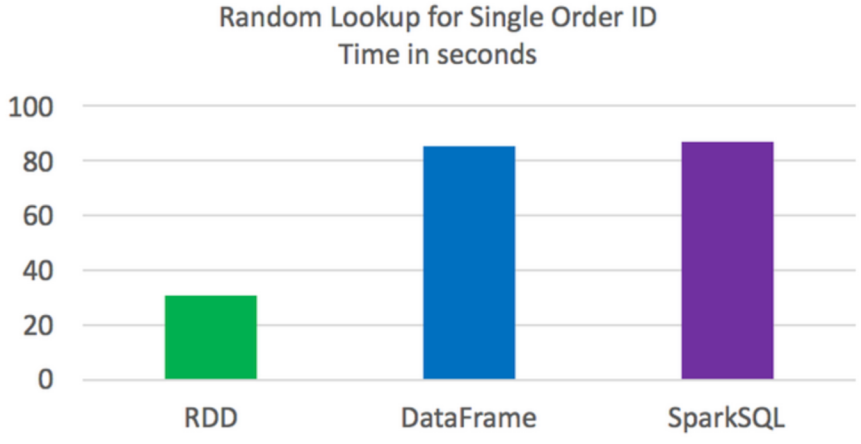
# DataFrame

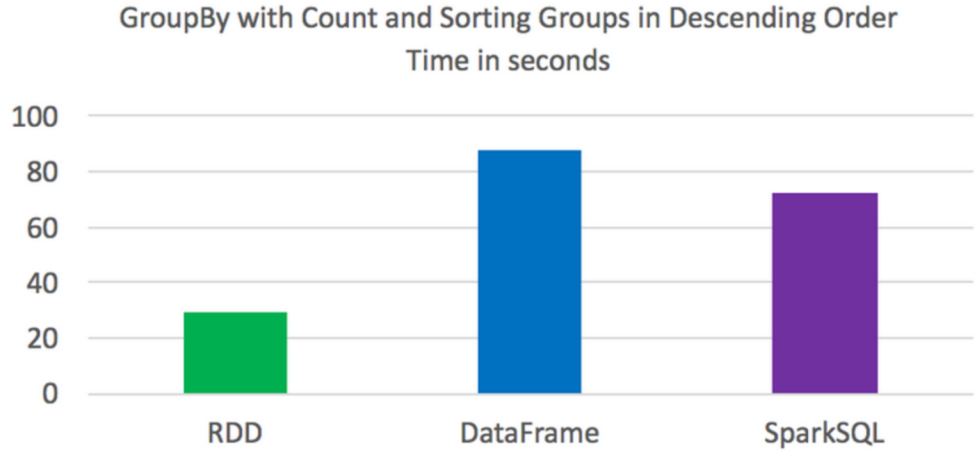
## Начиная со Spark 1.6 обобщенный API для работы с RDD

```
+------+--------+
|  name|avg(age)|
+------+--------+
|Brooke|    22.5|
| Jules|    30.0|
|    TD|    35.0|
| Denny|    31.0|
+------+--------+
```

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg
# Create a DataFrame using SparkSession
spark = (SparkSession
.builder
.appName("AuthorsAges")
.getOrCreate())
# Create a DataFrame
data_df = spark.createDataFrame([("Brooke", 20), ("Denny", 31), ("Jules", 30),("TD", 35), ("Brooke", 25)], ["name", "age"])
# Group the same names together, aggregate their ages, and compute an average
avg_df = data_df.groupBy("name").agg(avg("age"))
# Show the results of the final execution
avg_df.show()
```

# Сравнение скорости выполнения SQL-запросов на разных структурах данных




[Spark RDDs vs DataFrames vs SparkSQL]

# Spark DataFrame

## таблица с типизированными колонками

| Id (Int) | First (String) | Last (String) | Url (String) | Published (Date) | Hits (Int) | Campaigns (List[Strings]) |
|---|---|---|---|---|---|---|
| 1 | Jules | Damji | https://tinyurl.1 | 1/4/2016 | 4535 | [twitter, LinkedIn] |
| 2 | Brooke | Wenig | https://tinyurl.2 | 5/5/2018 | 8908 | [twitter, LinkedIn] |
| 3 | Denny | Lee | https://tinyurl.3 | 6/7/2019 | 7659 | [web, twitter, FB, LinkedIn] |
| 4 | Tathagata | Das | https://tinyurl.4 | 5/12/2018 | 10568 | [twitter, FB] |

# Основные типы данных Python

## значения в колонках

| Data type | Value assigned in Python | API to instantiate |
|---|---|---|
| ByteType | int | DataTypes.ByteType |
| ShortType | int | DataTypes.ShortType |
| IntegerType | int | DataTypes.IntegerType |
| LongType | int | DataTypes.LongType |
| FloatType | float | DataTypes.FloatType |
| DoubleType | float | DataTypes.DoubleType |
| StringType | str | DataTypes.StringType |
| BooleanType | bool | DataTypes.BooleanType |
| DecimalType | decimal.Decimal | DecimalType |

# Python structured data types in Spark

| Data type | Value assigned in Python | API to instantiate |
|---|---|---|
| `BinaryType` | `bytearray` | `BinaryType()` |
| `TimestampType` | `datetime.datetime` | `TimestampType()` |
| `DateType` | `datetime.date` | `DateType()` |
| `ArrayType` | List, tuple, or array | `ArrayType(dataType, [nullable])` |
| `MapType` | `dict` | `MapType(keyType, valueType, [nullable])` |
| `StructType` | List or tuple | `StructType([fields])` |
| `StructField` | A value type corresponding to the type of this field | `StructField(name, dataType, [nullable])` |

# Задание schema
## и зачем это нужно

- Избегаем приведение типов
- Не нужно считывать файл только для определения схемы данных (существенно для больших файлов)
- Можно отловить несоответствие исходных данных

```
1 from pyspark.sql.types import *
2 schema = StructType([StructField("author", StringType(), False),
3 StructField("title", StringType(), False),
4 StructField("pages", IntegerType(), False)])
5
6 #the same schema using DDL
7 schema = "author STRING, title STRING, pages INT"
```

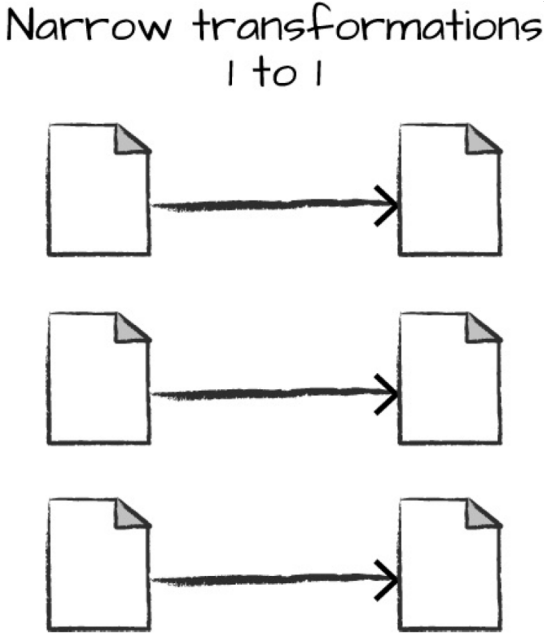
# Создаем статический DataFrame

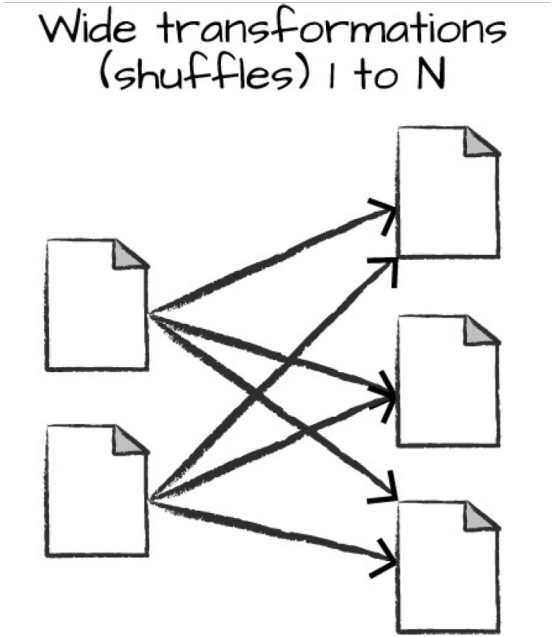
```
from pyspark.sql import SparkSession
# Define schema for our data using DDL
schema = "`Id` INT, `First` STRING, `Last` STRING, `Url` STRING,
`Published` STRING, `Hits` INT, `Campaigns` ARRAY<STRING>"
# Create our static data
data = [
[1, "Jules", "Damji", "https://tinyurl.1", "1/4/2016", 4535, ["twitter","LinkedIn"]],
[2, "Brooke","Wenig", "https://tinyurl.2", "5/5/2018", 8908, ["twitter","LinkedIn"]]
]

# Create a DataFrame using the schema defined above
blogs_df = spark.createDataFrame(data, schema)
# Show the DataFrame; it should reflect our table above
blogs_df.show()
# Print the schema used by Spark to process the DataFrame
print(blogs_df.printSchema())
```

# Transformation types

narrow -> in-memory filters

wide -> shuffle



# Transformations and actions as Spark operations

## Трансформации копятся до вызова action

| Transformations | Actions |
|---|---|
| orderBy() | show() |
| groupBy() | take() |
| filter() | count() |
| select() | collect() |
| join() | save() |

# Count, countDistinct

агрегирующие функции могут вызываться по всему набору данных

```
1 from pyspark.sql.functions import count
2 df.select(count("StockCode")).show()
3
4 from pyspark.sql.functions import countDistinct
5 df.select(countDistinct("StockCode")).show()
6
7 from pyspark.sql.functions import approx_count_distinct
8 df.select(approx_count_distinct("StockCode", 0.1)).show()
```

```
df.groupBy("key").agg(
    percentile_approx("value", 0.5, lit(1000000)).alias("median")
).printSchema()
```

# Column alias

## название колонки

```
from pyspark.sql.functions import sum, count, avg, expr
df.select(
  count("Quantity").alias("total_transactions"),
  sum("Quantity").alias("total_purchases"),
  avg("Quantity").alias("avg_purchases"),
  expr("mean(Quantity)").alias("mean_purchases"))\
  .selectExpr(
  "total_purchases/total_transactions",
  "avg_purchases",
  "mean_purchases")\
.show()
```

# Booleans

## Фильтрация по значению в колонке

```
# in Python
from pyspark.sql.functions import col
df.where(col("InvoiceNo") != 536365)\
.select("InvoiceNo", "Description")\
.show(5, False)
```

```
from pyspark.sql.functions import instr

priceFilter = col("UnitPrice") > 600
descripFilter = instr(df.Description, "POSTAGE") >= 1

df.where(df.StockCode.isin("DOT"))\
.where(priceFilter | descripFilter).show()
```

# Nulls

## первое не-null значение из списка столбцов

```
1 from pyspark.sql.functions import coalesce
2
3 df.select(coalesce(col("Description"), col("CustomerId")))\
4     .show()
```

## удаление строк, содержащих null

```
1 df.na.drop()
2 df.na.drop("any") # drop if ANY column is null
3
4 df.na.drop("all") # drop if ALL columns are null
5
6 df.na.drop("all", subset=["StockCode", "InvoiceNo"])
```

# Nulls

## заполнение пустых значений

```
1 df.na.fill("it was null value")
2
3 # specify values with dict
4 fill_cols_vals = {"StockCode": 5, "Description" : "No Value"}
5 df.na.fill(fill_cols_vals)
```

# SQL для создания колонок и логических выражений через expr

```
1 from pyspark.sql.functions import expr
2 df.withColumn("isExpensive", expr("NOT UnitPrice <= 250"))\
3 .where("isExpensive")\
4 .select("Description", "UnitPrice").show(5)
```

# Численные значения

```
from pyspark.sql.functions import expr, pow
fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5

df.select(expr("CustomerId"), \
          fabricatedQuantity.alias("realQuantity"))\
.show(2)

# либо через SQL
df.selectExpr("CustomerId",
"(POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity")\
.show(2)
```