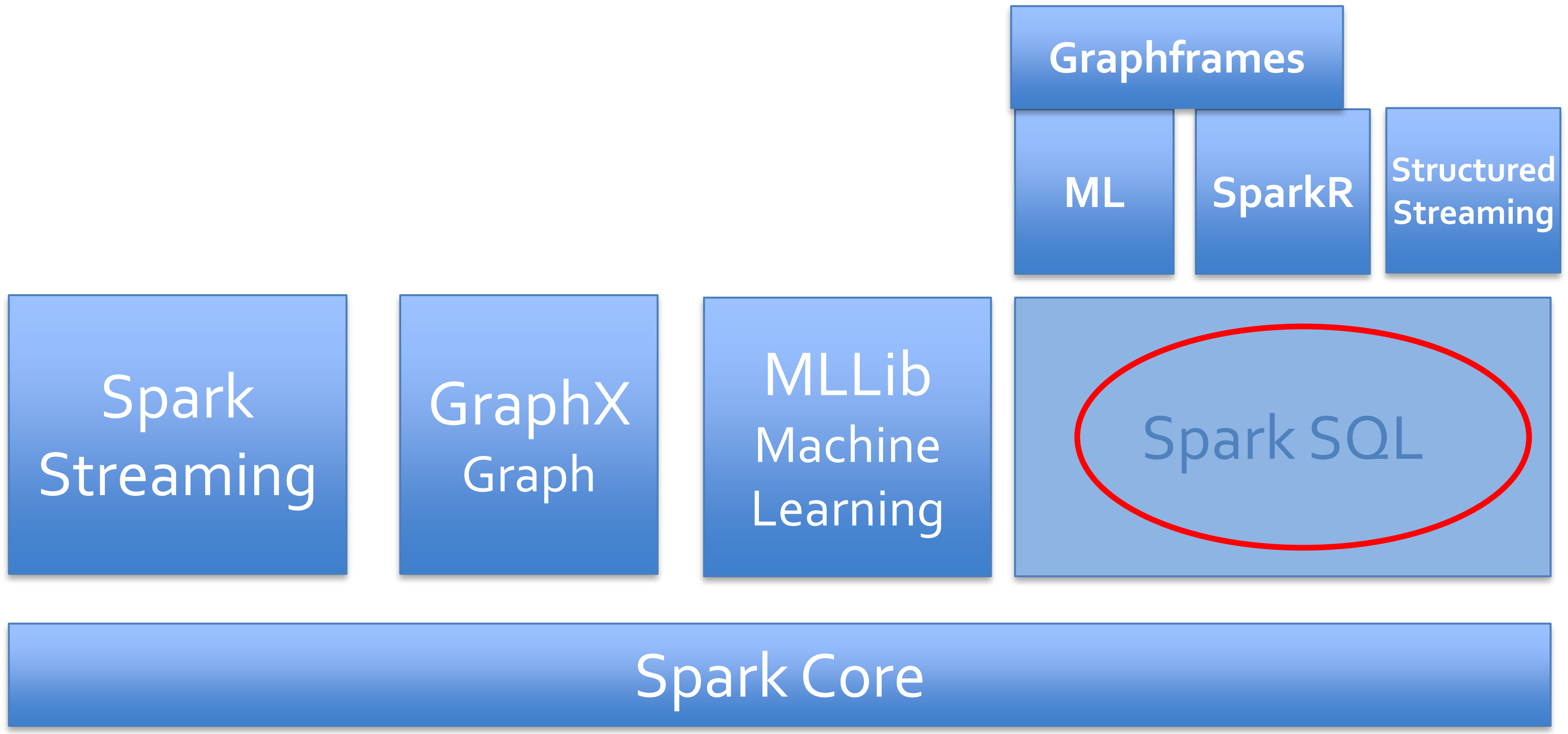
2. SPARK SQL

Apache Spark - November 2020

EDEM
Escuela de Empresarios

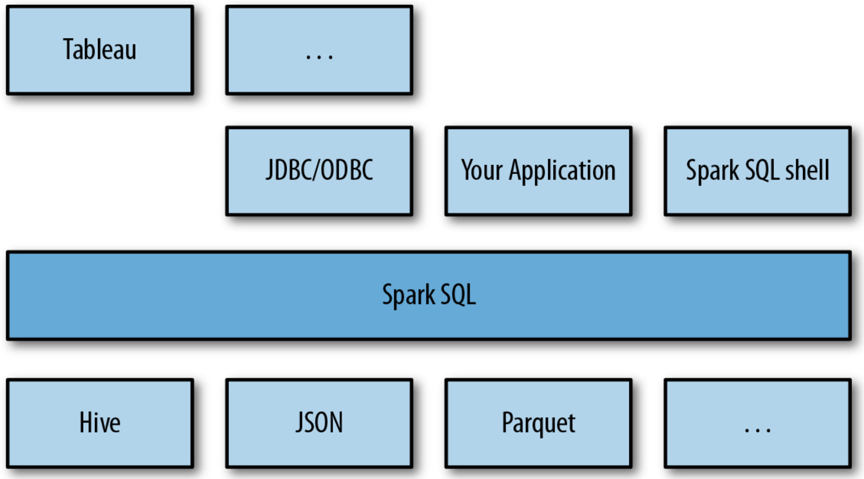
Introduction to Spark SQL

- **Spark SQL** was first released in Spark 1.0 (May, 2014)
- Initial committed by Michael Armbrust & Reynold Xin from **Databricks**



Introduction to Spark SQL

- Interface for working with structured (schema) and semi-structured data
- Spark SQL applies structured views to data stored in different formats
- Three main capabilities:
  - DataFrame abstraction for structured datasets.
    - Similar to tables in Relational database.
  - Read & write in **structured formats** (JSON, Hive, Parquet,…)
  - Query data using SQL inside Spark program & from external tools using JDBC/ODBC



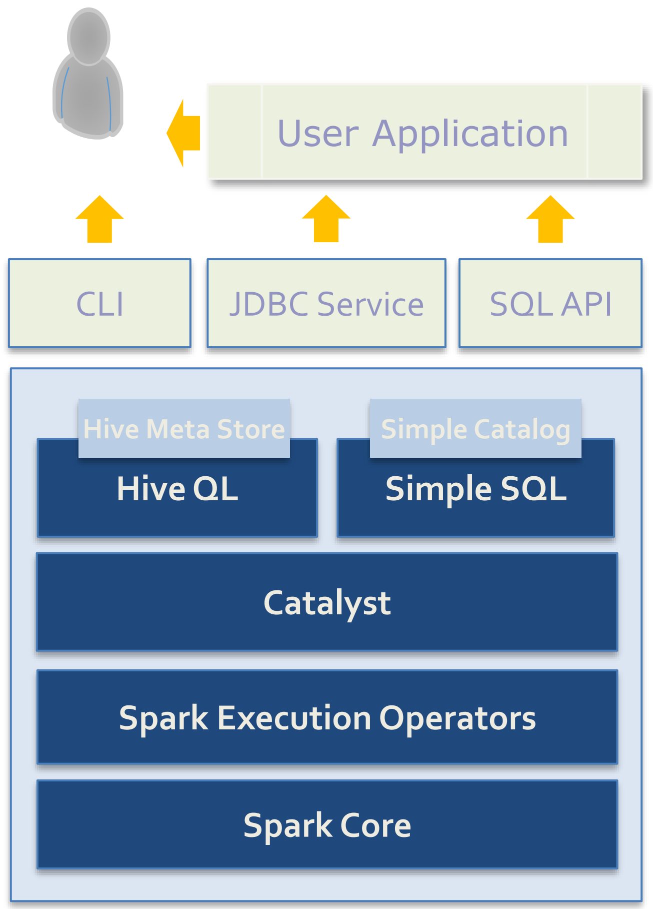
Introduction to Spark SQL

- Mix SQL queries with Spark programs
  - Process structured data (SQL tables, JSON files) as RDDs
- Load and query data from a variety of sources
  - Apache Hive tables
  - Parquet files
  - JSON files
  - Cassandra column families
- Run unmodified Hive queries
  - Reuses the Hive metastore, data, queries, SerDes and UDFs
- Connect through JDBC or ODBC
  - Spark SQL includes a server mode
  - Use BI tools

Component Stack

From a user perspective, Spark SQL:

- Hive-like interface(JDBC Service / CLI)
- SQL API support (LINQ-like)
- Both HiveQL & Simple SQL dialects are Supported
- DDL is 100% compatible with Hive Metastore
- HiveQL aims to 100% compatible with Hive DML
- Simple SQL dialect is now very weak in functionality, but easy to extend



Spark SQL – Hive Integration

- Spark SQL supports reading and writing data store in Hive
- Accessing to Hive UDFs (Hive installation not required)

```
SparkSession.builder().enableHiveSupport().getOrCreate()
```

- It´s necessary to add *spark-hive* dependency

```
libraryDependencies ++= Seq(
        “org.apache.spark” %% “spark-sql” % sparkVersion
        “org.apache.spark” %% “spark-hive” % sparkVersion)
```

- The default Hive Metastore version is 1.2.1.

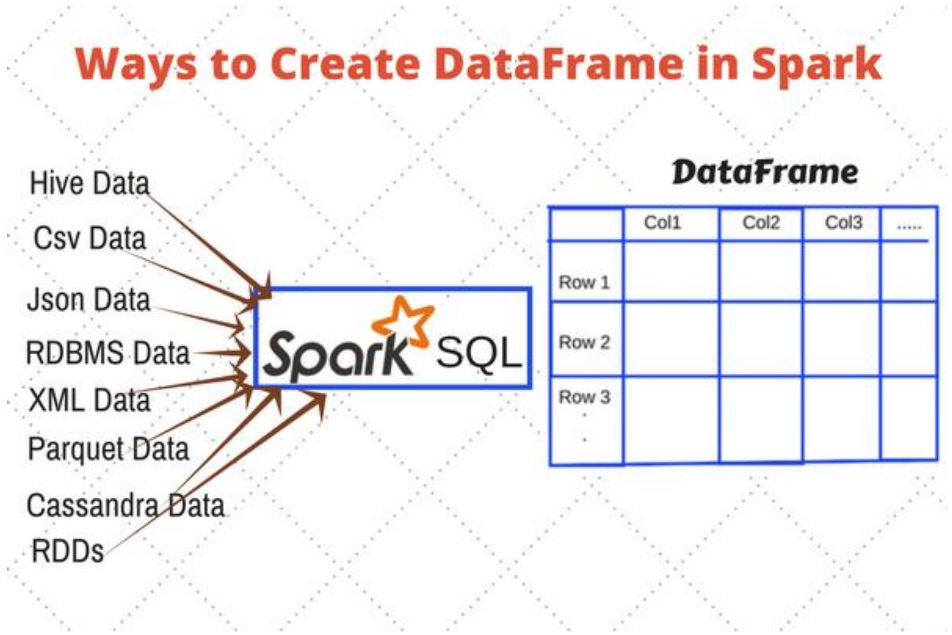
```
// Create a Hive managed Parquet table, with HQL syntax instead of the Spark SQL native syntax
// `USING hive`
sql("CREATE TABLE hive_records(key int, value string) STORED AS PARQUET")
```

Spark SQL – Hive Integration

- If you have a Hive installation, copy *hive-site.xml* to Spark conf directory
- If not, Spark SQL will:
  - Create its own Hive metastore in your program's working directory **metastore_db**
  - Tables will be placed in **/user/hive/warehouse** in local filesystem or HDFS if you have **hdfs-site.xml** on your classpath
    - hive.metastore.warehouse.dir

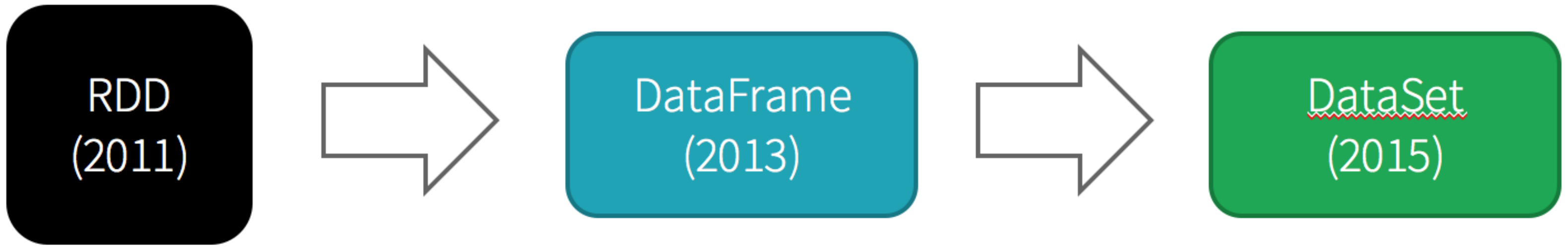
DataFrames & Datasets

- Represent distributed collections (like RDDs)
- **Adding schema** information not found in RDDs
- More **efficient** storage layer ([Tungsten])
- Provide **new operations** and can run SQL queries
- Creation from:
  - External data sources
  - Result of queries
  - Regular RDDs
  - …



DataFrames & Datasets

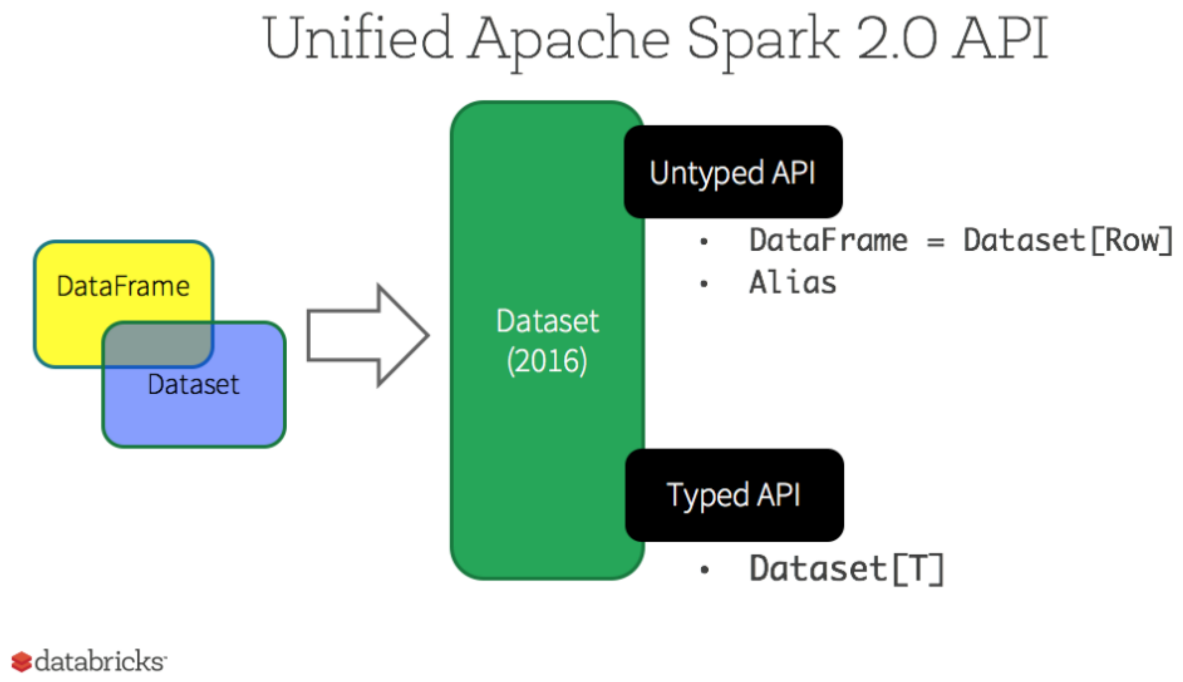
History of Spark APIs



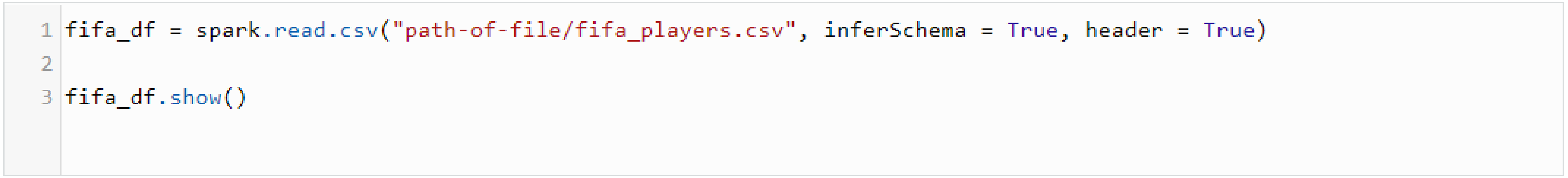
Distribute collection of JVM objects	Distribute collection of Row objects	Internally rows, externally JVM objects
Functional Operators (map, filter, etc.)	Expression-based operations and UDFs	Almost the "Best of both worlds": **type safe + fast**
	Logical plans and optimizer	But slower than DF Not as good for interactive analysis, especially Python
	Fast/efficient internal representations	

DataFrames & Datasets

- Starting in Spark 2.0, DataFrames and Datasets were unified

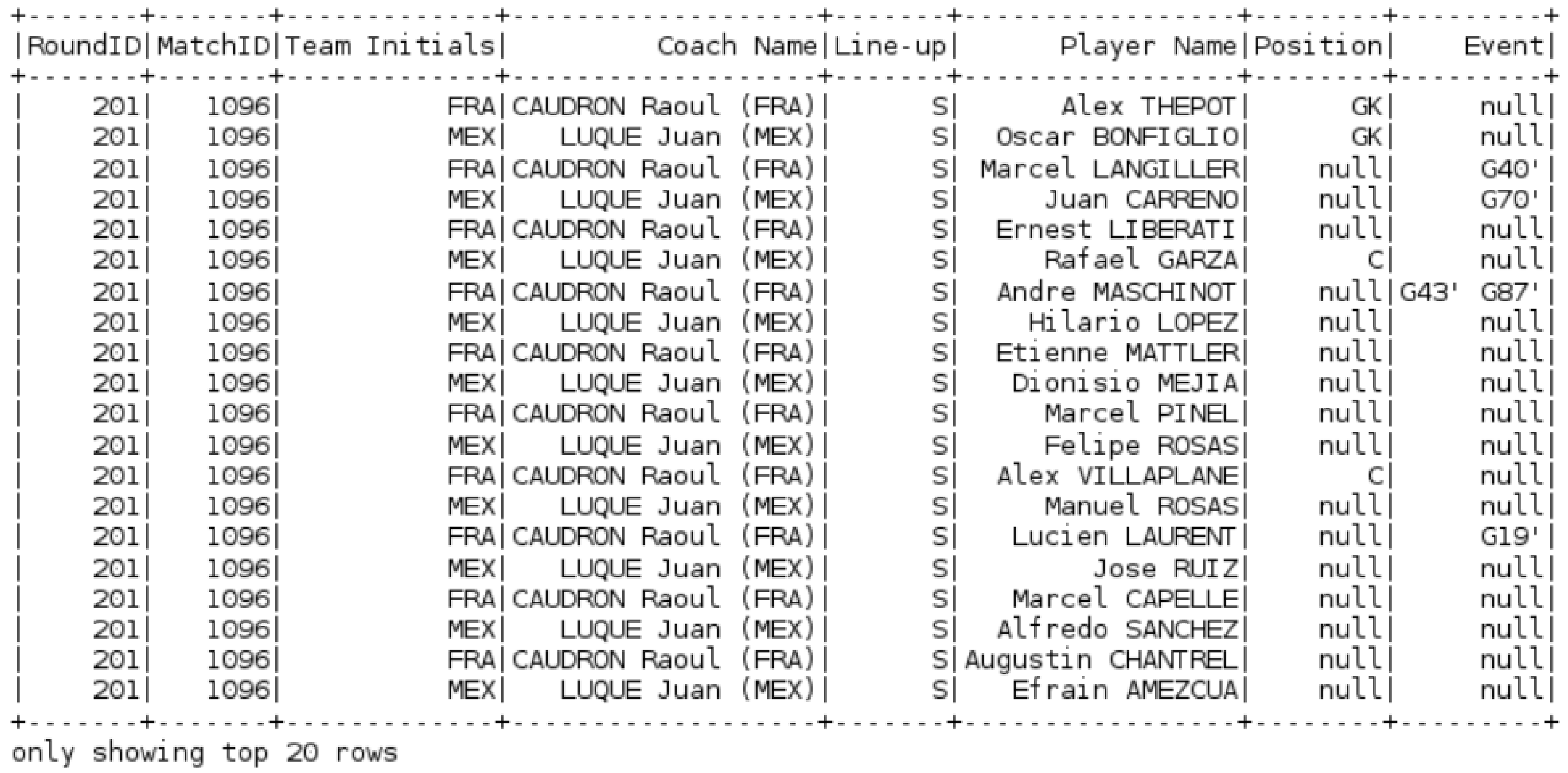


DataFrame in PySpark



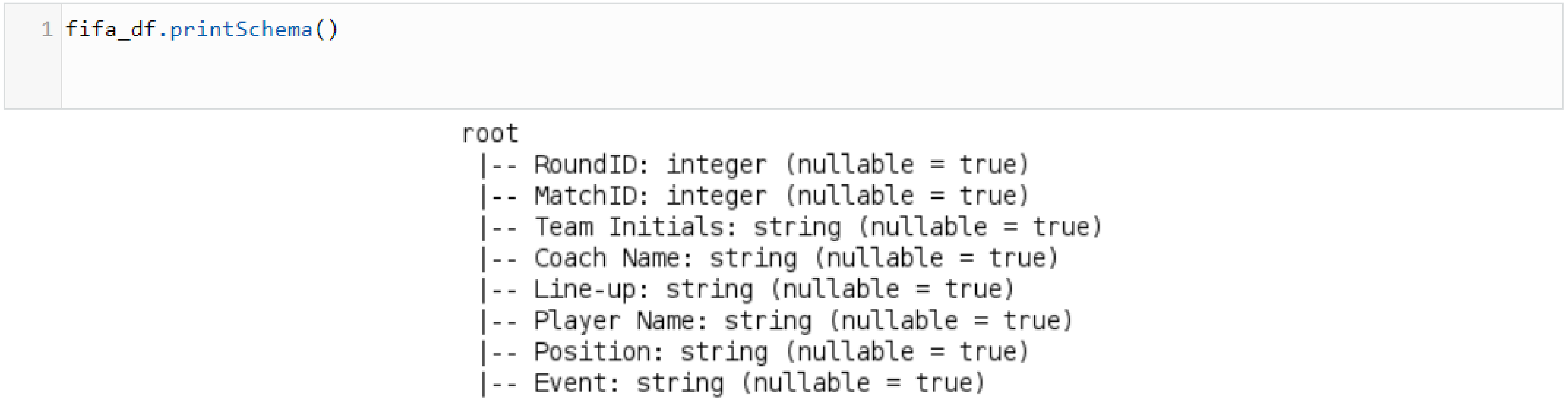
```python
fifa_df = spark.read.csv("path-of-file/fifa_players.csv", inferSchema = True, header = True)

fifa_df.show()
```



```
+-------+-------+-------------+-------------------+-------+-----------------+--------+---------+
|RoundID|MatchID|Team Initials|         Coach Name|Line-up|      Player Name|Position|    Event|
+-------+-------+-------------+-------------------+-------+-----------------+--------+---------+
|    201|   1096|          FRA|CAUDRON Raoul (FRA)|      S|      Alex THEPOT|      GK|     null|
|    201|   1096|          MEX|   LUQUE Juan (MEX)|      S|  Oscar BONFIGLIO|      GK|     null|
|    201|   1096|          FRA|CAUDRON Raoul (FRA)|      S| Marcel LANGILLER|    null|     G40'|
|    201|   1096|          MEX|   LUQUE Juan (MEX)|      S|     Juan CARRENO|    null|     G70'|
|    201|   1096|          FRA|CAUDRON Raoul (FRA)|      S|  Ernest LIBERATI|    null|     null|
|    201|   1096|          MEX|   LUQUE Juan (MEX)|      S|     Rafael GARZA|       C|     null|
|    201|   1096|          FRA|CAUDRON Raoul (FRA)|      S|  Andre MASCHINOT|    null|G43' G87'|
|    201|   1096|          MEX|   LUQUE Juan (MEX)|      S|    Hilario LOPEZ|    null|     null|
|    201|   1096|          FRA|CAUDRON Raoul (FRA)|      S|  Etienne MATTLER|    null|     null|
|    201|   1096|          MEX|   LUQUE Juan (MEX)|      S|   Dionisio MEJIA|    null|     null|
|    201|   1096|          FRA|CAUDRON Raoul (FRA)|      S|     Marcel PINEL|    null|     null|
|    201|   1096|          MEX|   LUQUE Juan (MEX)|      S|     Felipe ROSAS|    null|     null|
|    201|   1096|          FRA|CAUDRON Raoul (FRA)|      S|  Alex VILLAPLANE|       C|     null|
|    201|   1096|          MEX|   LUQUE Juan (MEX)|      S|     Manuel ROSAS|    null|     null|
|    201|   1096|          FRA|CAUDRON Raoul (FRA)|      S|   Lucien LAURENT|    null|     G19'|
|    201|   1096|          MEX|   LUQUE Juan (MEX)|      S|        Jose RUIZ|    null|     null|
|    201|   1096|          FRA|CAUDRON Raoul (FRA)|      S|   Marcel CAPELLE|    null|     null|
|    201|   1096|          MEX|   LUQUE Juan (MEX)|      S|  Alfredo SANCHEZ|    null|     null|
|    201|   1096|          FRA|CAUDRON Raoul (FRA)|      S|Augustin CHANTREL|    null|     null|
|    201|   1096|          MEX|   LUQUE Juan (MEX)|      S|   Efrain AMEZCUA|    null|     null|
+-------+-------+-------------+-------------------+-------+-----------------+--------+---------+
only showing top 20 rows
```

DataFrame in PySpark

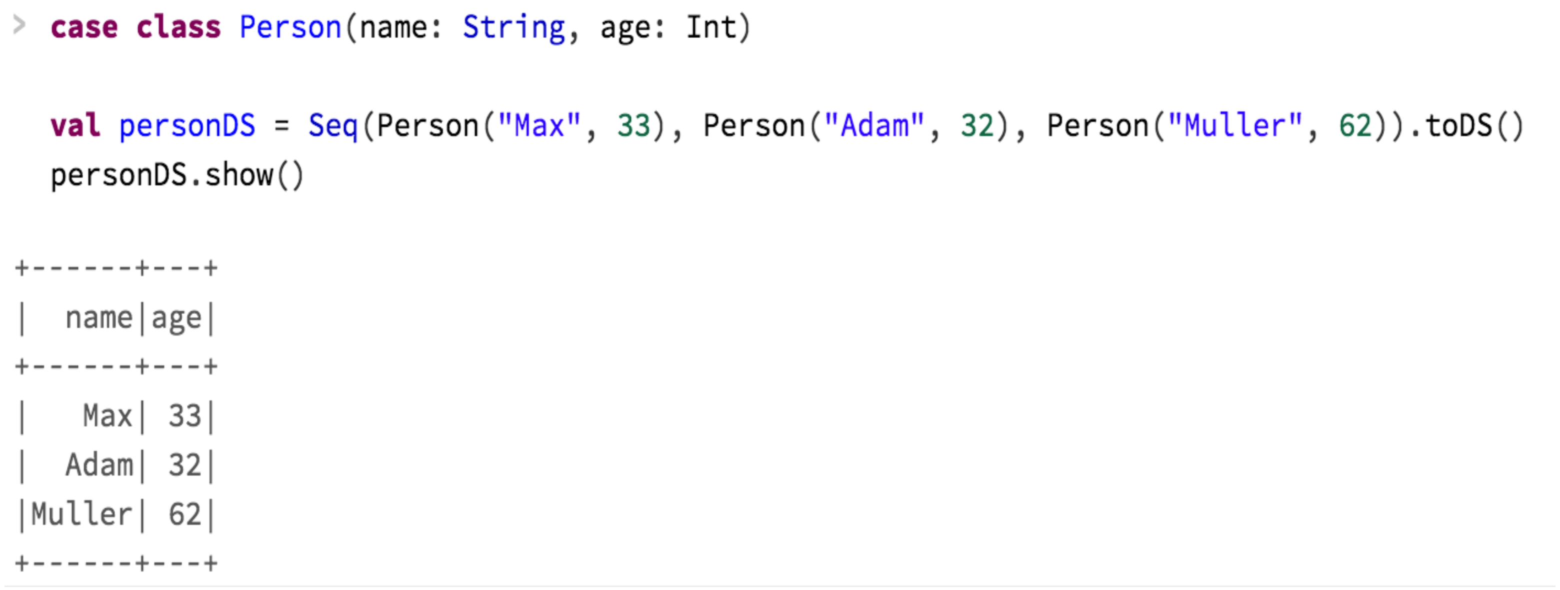


```python
fifa_df.printSchema()
```

```
root
 |-- RoundID: integer (nullable = true)
 |-- MatchID: integer (nullable = true)
 |-- Team Initials: string (nullable = true)
 |-- Coach Name: string (nullable = true)
 |-- Line-up: string (nullable = true)
 |-- Player Name: string (nullable = true)
 |-- Position: string (nullable = true)
 |-- Event: string (nullable = true)
```

DataFrames & Datasets

- DataFrames are Datasets of a special **Row object**
- **Row** is a generic untyped JVM object
- Dataset is a collection of strongly-typed JVM
- Python doesn´t support Spark Datasets



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

val personDS = Seq(Person("Max", 33), Person("Adam", 32), Person("Muller", 62)).toDS()
personDS.show()
```

```
+------+---+
|  name|age|
+------+---+
|   Max| 33|
|  Adam| 32|
|Muller| 62|
+------+---+
```

DataFrames & Datasets

- DataFrame

```
data.groupBy( “dept”)).avg(“age”)
```

- SQL

```
spark.sql(“select dept, avg(age) from data group by dept”)
```

- RDD

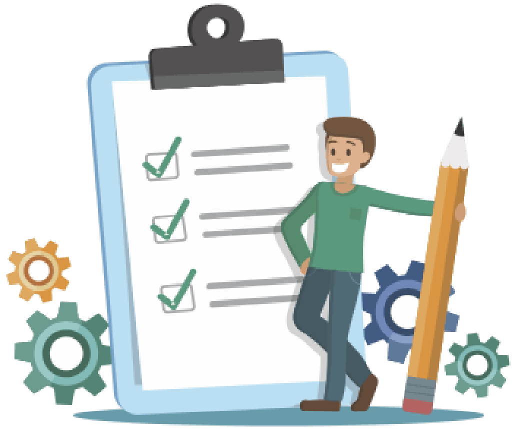
```
data.map {  case (dept, age) => dept -> (age,1) }
      .reduceByKey {  case ( (a1, c1), (a2, c2) ) => (a1 + a2, c1 + c2) }
      .map{ case (dept, (age, c)) => dept -> age/ c }
```

Spark SQL – DataFrames – Datasets

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

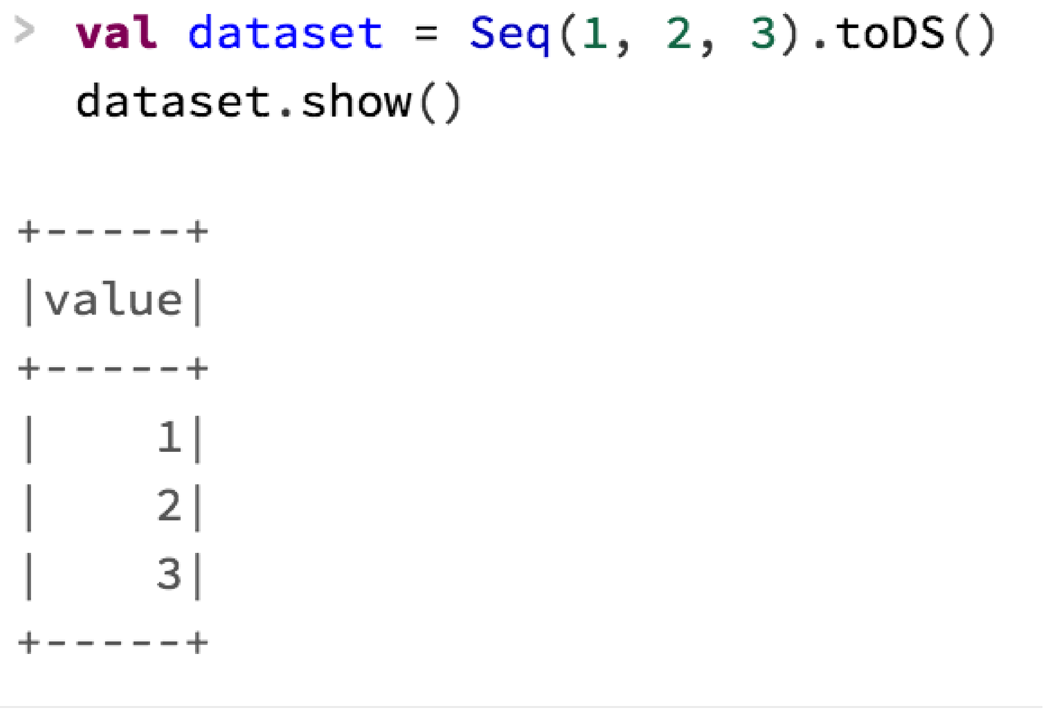
Hands-on

- Open “01.Spark_SQL.ipynb” in Google Colab:
    - Execute examples 1 and 2



Dataset Example in Scala

- You can simply call **.toDS()** on a sequence to convert the sequence to a Dataset

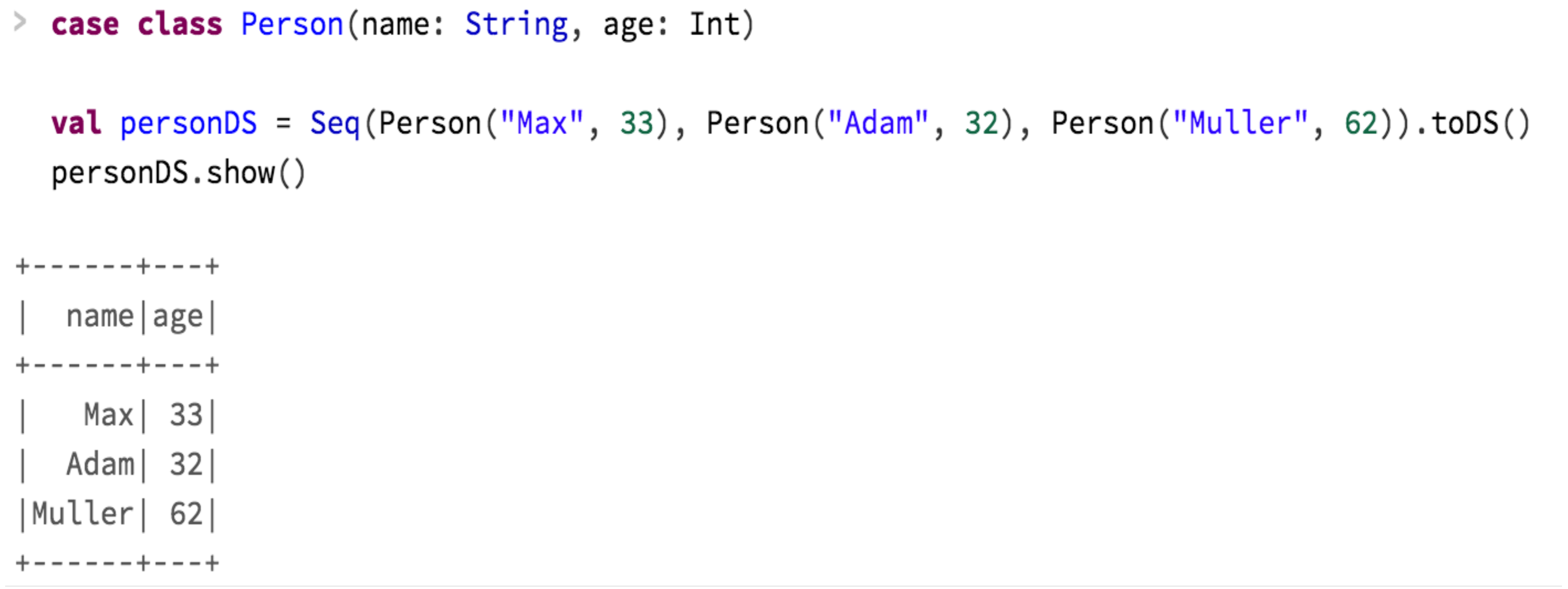


```
> val dataset = Seq(1, 2, 3).toDS()
  dataset.show()

+-----+
|value|
+-----+
|    1|
|    2|
|    3|
+-----+
```

Dataset Example in Scala

- *Encoders are also created for case classes -similar to a DTO pattern*



```
case class Person(name: String, age: Int)

val personDS = Seq(Person("Max", 33), Person("Adam", 32), Person("Muller", 62)).toDS()
personDS.show()
```

```
+------+---+
|  name|age|
+------+---+
|   Max| 33|
|  Adam| 32|
|Muller| 62|
+------+---+
```

Dataset Example in Scala

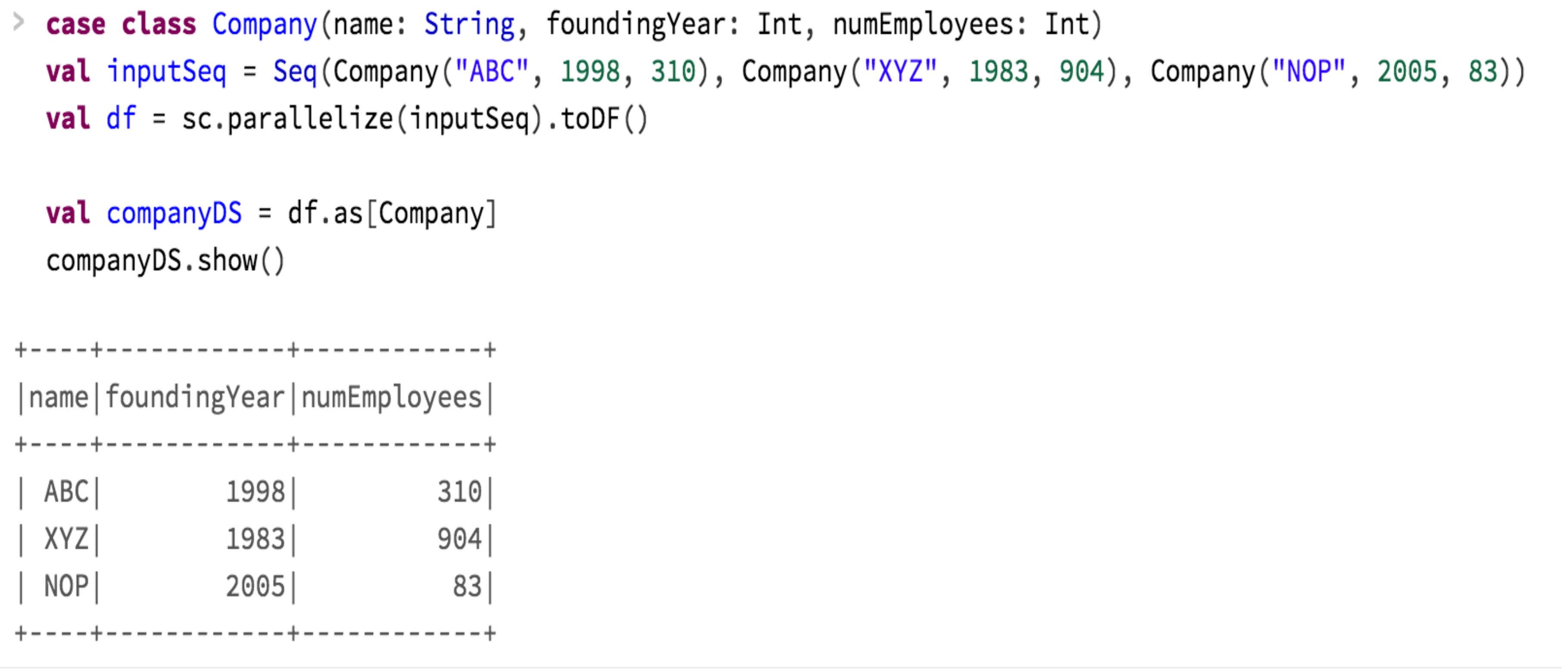
- *Create Dataset from a RDD*
  - *Use rdd.toDS()*

```
> val rdd = sc.parallelize(Seq((1, "Spark"), (2, "Databricks")))
  val integerDS = rdd.toDS()
  integerDS.show()
```

```
+---+----------+
| _1|        _2|
+---+----------+
|  1|     Spark|
|  2|Databricks|
+---+----------+
```

Dataset Example in Scala

- *Create Dataset from a DataFrame*
  - *Use df.as[SomeCaseClass]*



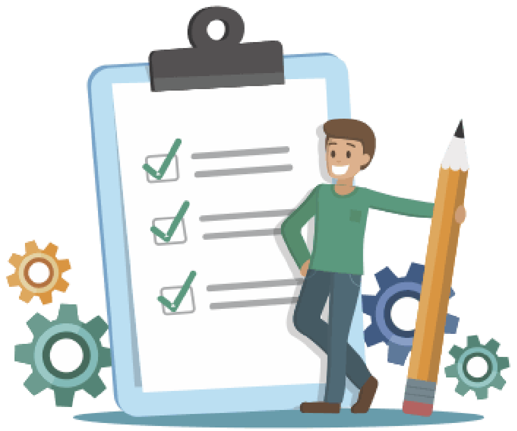
```
case class Company(name: String, foundingYear: Int, numEmployees: Int)
val inputSeq = Seq(Company("ABC", 1998, 310), Company("XYZ", 1983, 904), Company("NOP", 2005, 83))
val df = sc.parallelize(inputSeq).toDF()

val companyDS = df.as[Company]
companyDS.show()
```

```
+----+------------+------------+
|name|foundingYear|numEmployees|
+----+------------+------------+
| ABC|        1998|         310|
| XYZ|        1983|         904|
| NOP|        2005|          83|
+----+------------+------------+
```

Hands-on

- Open “01.Spark_SQL.ipynb” in Google Colab:
    - Try the exercises 1, 2, 3 and 4



DataFrame – Run SQL directly on files

- Instead of using read API to load a file into DataFrame and query it, you can also query that file directly with SQL

```
df = spark.sql("SELECT * FROM parquet.`./resources/users.parquet`")
```

Repartition vs Coalesce

- Repartition can be used to either increase and decrease the number of partitions
  - It’s a full shuffle operation
  - Partitions equally distributed

```
val new_df = df.repartition(100)
val new_df = df.repartition(100, $”id”)
```

- Coalesce reduces the number of partitions.
  - Avoids shuffle

```
val new_df = df.coalesce(10)
```

Repartition vs Coalesce

Repartitioning	Coalesce
19M repartition/part-00000	
19M repartition/part-00001	
19M repartition/part-00002	33M coalesce/part-00000
19M repartition/part-00003	29M coalesce/part-00001
19M repartition/part-00004	30M coalesce/part-00002
19M repartition/part-00005	31M coalesce/part-00003
19M repartition/part-00006	32M coalesce/part-00004
19M repartition/part-00007	33M coalesce/part-00005
19M repartition/part-00008	
19M repartition/part-00009	

Save To Persistent Tables

- When working with a HiveContext, DataFrames can also be saved as persistent tables using the **saveAsTable** command
  - Unlike the **registerTempTable** command, saveAsTable will materialize the contents of the dataframe and create a pointer to the data in the HiveMetastore
  - Persistent tables will still exist even after your Spark program has restarted
  - **By default saveAsTable will create a "managed table"**, meaning that the location of the data will be controlled by the metastore
    - Managed tables will also have their data deleted automatically when a table is dropped

SparkContext – SQLContext - HiveContext

SparkContext

- The driver program use the SparkContext to connect and communicate with the cluster (YARN, Mesos)
- Using SparkContext you can actually get access to SQLContext and HiveContext
- Setting configuration parameters

```
val sparkConf = new SparkConf().setAppName("Example").setMaster("yarn")
val sc = new SparkContext(sparkConf)
```

SQLContext

- SQLContext is your gateway to SparkSQL

```
// sc is an existing SparkContext. (Deprecated)
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

HiveContext

- HiveContext is your gateway to Hive
- HiveContext extends SQLContext, including Hive functionalities

```
// sc is an existing SparkContext. (Deprecated)
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

SparkSession

- **SparkSession** was introduce in Spark 2.0
- All contexts unified in only one
- Having access to SparkSession, we automatically have Access to the SparkContext
- SparkSession is now **the new entry point**
- We can start working with DataFrame and Dataset having access to SparkSession

```
val spark = SparkSession
  .builder
  .enableHiveSupport()
  .config("spark.master", "local[*]")
  .appName("Simple Application")
  .getOrCreate()


val persons = spark
  .read
  .json(filePath)
```

SparkSession (PySpark)

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Spark Scala Application

```scala
val spark = SparkSession
  .builder
  .enableHiveSupport()
  .config("spark.master", "local[*]")
  .appName("Simple Application")
  .getOrCreate()

val filePath = "person.json"
val persons = spark
  .read
  .json(filePath)

val personsByName = persons
  .groupBy(col("name"))
  .count()

personsByName.show()
```

```
+-------+-----+
|   name|count|
+-------+-----+
|Michael|    2|
|   John|    1|
+-------+-----+
```

DataFrame UDF

- User-defined functions provide you with ways to extend the DataFrame and SQL APIs while keeping the Catalyst optimizer
  - Perfomance issues with Python, since data must still be transferred out of the JVM

```
val squared = (s: Long) => { s * s }

spark.udf.register("square", squared)

spark.sql("select id, square(id) as id_squared from test")
```

- Using UDFs with DataFrames

```
import org.apache.spark.sql.functions.{col, udf}
val squared = udf((s: Long) => s * s)
df.select(squared(col("id")) as "id_squared"))
```

DataFrame UDF – PySpark

- The default return type is *StringType*

```
def squared(s): return s * s
spark.udf.register("squaredWithPython", squared)
```

- You can optionally set the return type of your UDF

```
from pyspark.sql.types import LongType

def squared_typed(s): return s * s

spark.udf.register("squaredWithPython", squared_typed, LongType())
```

DataFrame UDF – PySpark

- Spark SQL (including SQL and the DataFrame and Dataset API) does not guarantee **the order of evaluation of subexpressions**
- There's no guarantee that the null check will happen before invoking the UDF. For example:

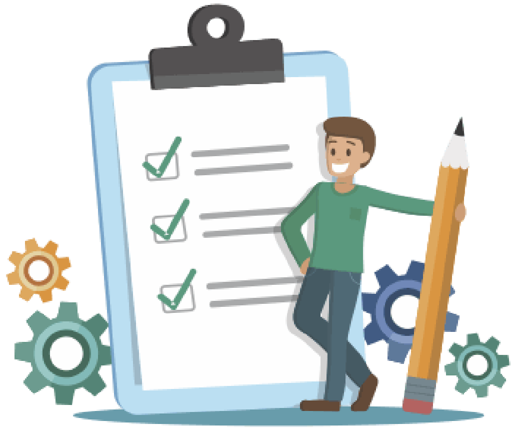
```
spark.udf.register("strlen", lambda s: (s), "int")
spark.sql("select s from test1 where s is not null and strlen(s) > 1") # no guarantee
```

- To perform proper null checking, we recommend that you do either of the following:
  - **Make the UDF itself null-aware** and do null checking inside the UDF itself
  - **Use IF or CASE WHEN expressions to do the null check** and invoke the UDF in a conditional branch

```
spark.udf.register("strlen_nullsafe", lambda s:(s) if not s is None else -1, "int")
spark.sql("select s from test1 where s is not null and strlen_nullsafe(s) > 1") // ok
spark.sql("select s from test1 where if(s is not null, strlen(s), null) > 1") // ok
```

Hands-on

- Open “01.Spark_SQL.ipynb” in Google Colab:
    - Try the exercises 5 and 6



Windows Partitioning

- There are two kinds of functions supported by SparkSQL that could be used to calculated a single return value:
  - UDFs like “substr”, “round”, etc.
  - Aggregated functions like SUM, MAX, etc
- **Window function** calculates a return value for every input row of a table based on a group of rows, called the Frame
- **Window function** makes them more powerful that other functions and allows users to express various data processing tasks that are hard (if not impossible)

Windows Partitioning

productRevenue

product	category	revenue
Thin	Cell phone	6000
Normal	Tablet	1500
Mini	Tablet	5500
Ultra thin	Cell phone	5000
Very thin	Cell phone	6000
Big	Tablet	2500
Bendable	Cell phone	3000
Foldable	Cell phone	3000
Pro	Tablet	4500
Pro2	Tablet	6500

"What are the best-selling and the second best-selling products in every category?"

```
SELECT
  product,
  category,
  revenue
FROM (
  SELECT
    product,
    category,
    revenue,
    dense_rank() OVER (PARTITION BY category ORDER BY revenue DESC) as rank
  FROM productRevenue) tmp
WHERE
  rank <= 2
```

product	category	revenue
Pro2	Tablet	6500
Mini	Tablet	5500
Thin	Cell Phone	6000
Very thin	Cell Phone	6000
Ultra thin	Cell Phone	5500

Windows Partitioning

productRevenue

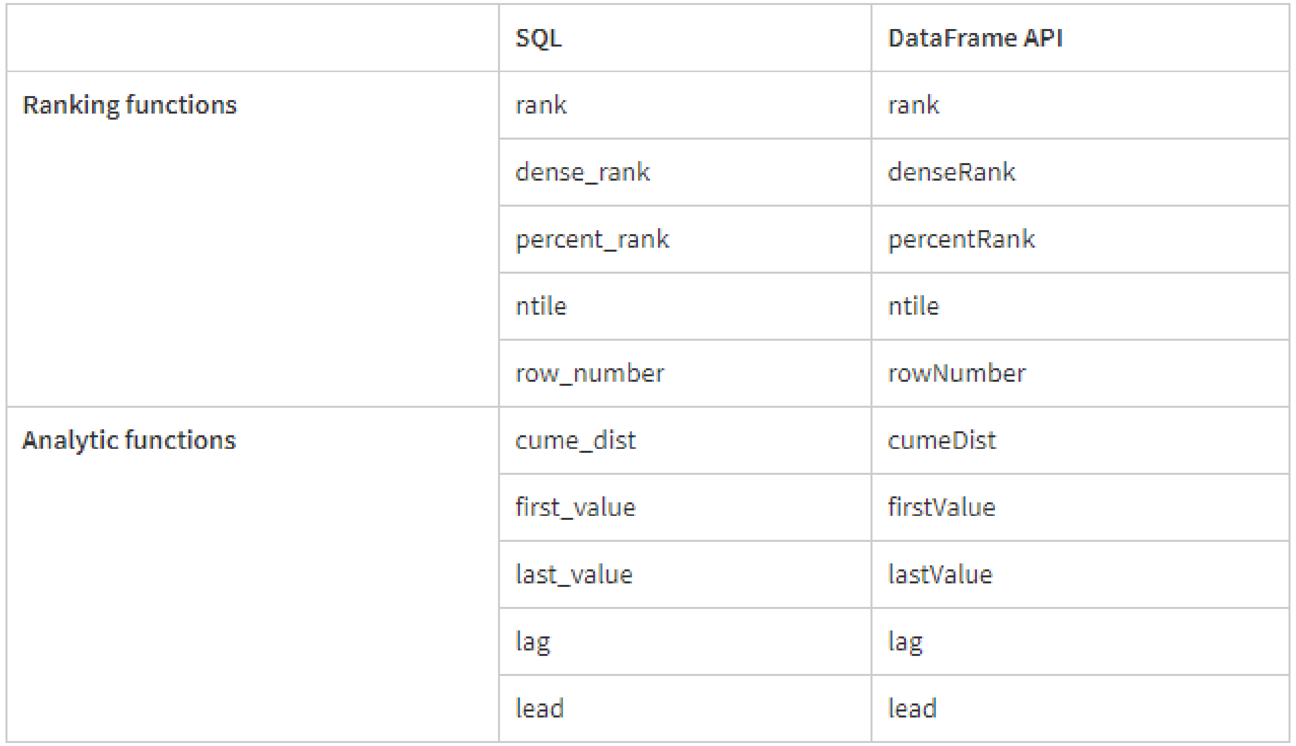
product	category	revenue
Thin	Cell phone	6000
Normal	Tablet	1500
Mini	Tablet	5500
Ultra thin	Cell phone	5000
Very thin	Cell phone	6000
Big	Tablet	2500
Bendable	Cell phone	3000
Foldable	Cell phone	3000
Pro	Tablet	4500
Pro2	Tablet	6500

"What is the difference between the revenue of each product and the revenue of the best selling product in the same category as that product?"

```python
import sys
from pyspark.sql.window import Window
import pyspark.sql.functions as func
windowSpec = \
  Window
    .partitionBy(df['category']) \
    .orderBy(df['revenue'].desc()) \
    .rangeBetween(-sys.maxsize, sys.maxsize)
dataFrame = sqlContext.table("productRevenue")
revenue_difference = \
  (func.max(dataFrame['revenue']).over(windowSpec) - dataFrame['revenue'])
dataFrame.select(
  dataFrame['product'],
  dataFrame['category'],
  dataFrame['revenue'],
  revenue_difference.alias("revenue_difference"))
```

product	category	revenue	revenue_difference
Pro2	Tablet	6500	0
Mini	Tablet	5500	1000
Pro	Tablet	4500	2000
Big	Tablet	2500	4000
Normal	Tablet	1500	5000
Thin	Cell Phone	6000	0
Very thin	Cell Phone	6000	0
Ultra thin	Cell Phone	5500	500
Foldable	Cell Phone	3000	3000
Bendable	Cell Phone	3000	3000

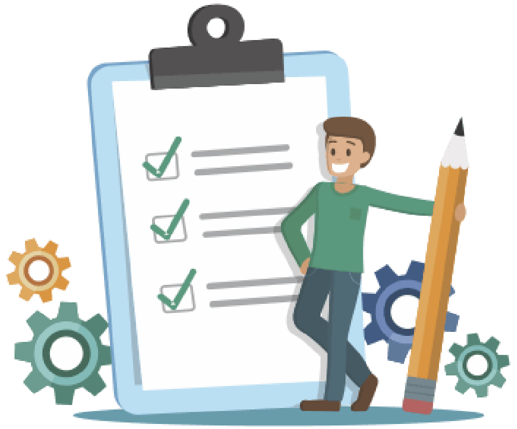
Windows Partitioning



	SQL	DataFrame API
Ranking functions	rank	rank
	dense_rank	denseRank
	percent_rank	percentRank
	ntile	ntile
	row_number	rowNumber
Analytic functions	cume_dist	cumeDist
	first_value	firstValue
	last_value	lastValue
	lag	lag
	lead	lead

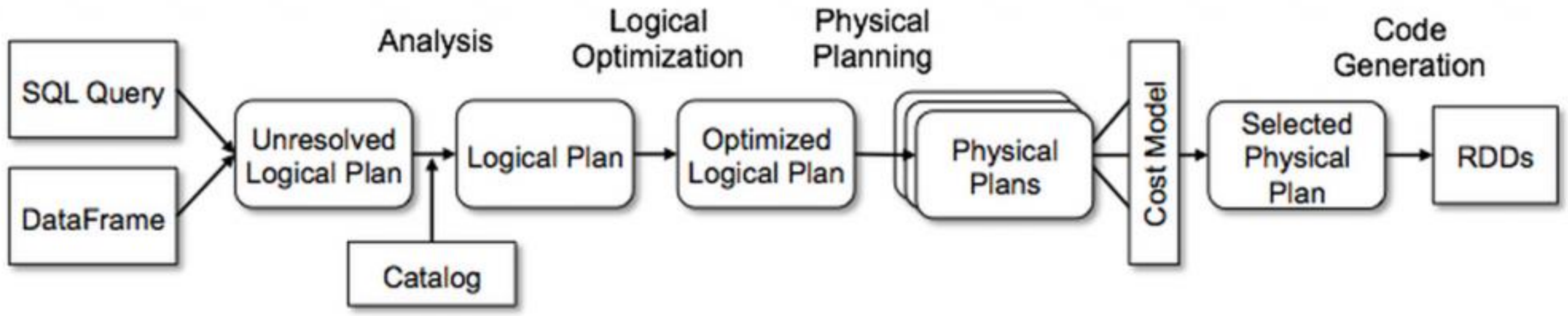
Hands-on

- Open “02.Spark_SQL_Advanced.ipynb” in Google Colab:
    - Example 1
    - Try exercises 1 and 2



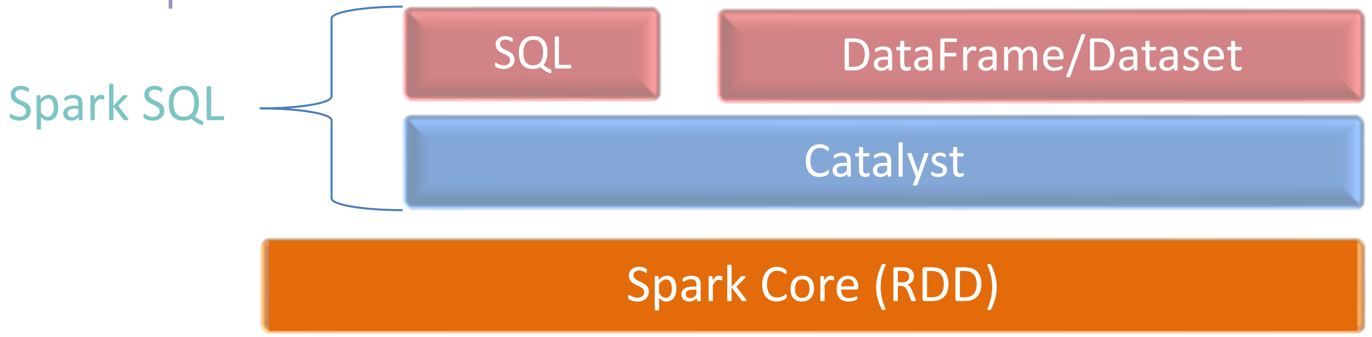
EDEM
Escuela de Empresarios

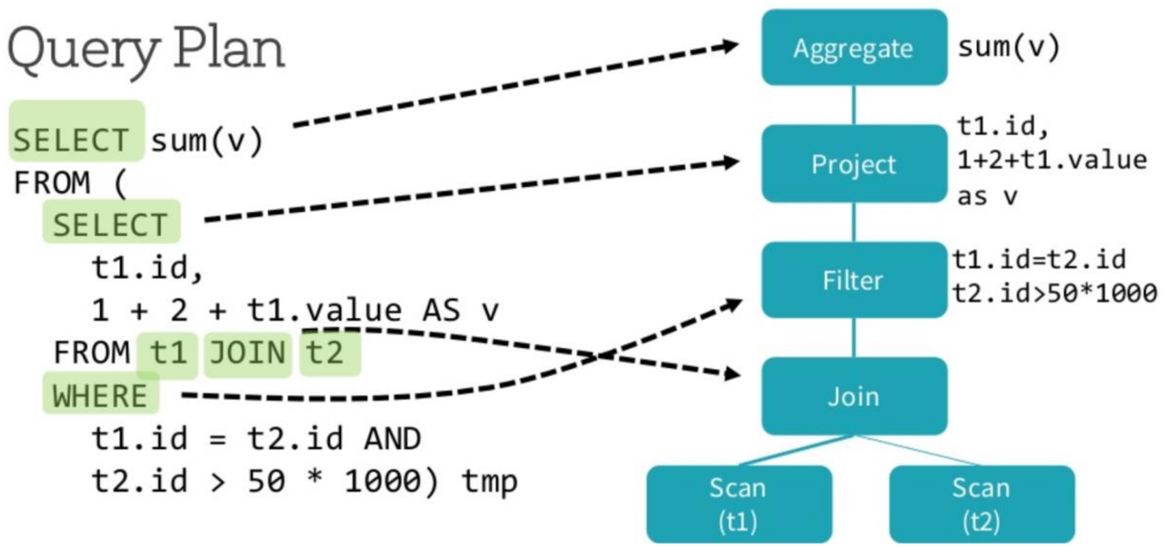
Spark SQL Architecture



Catalyst

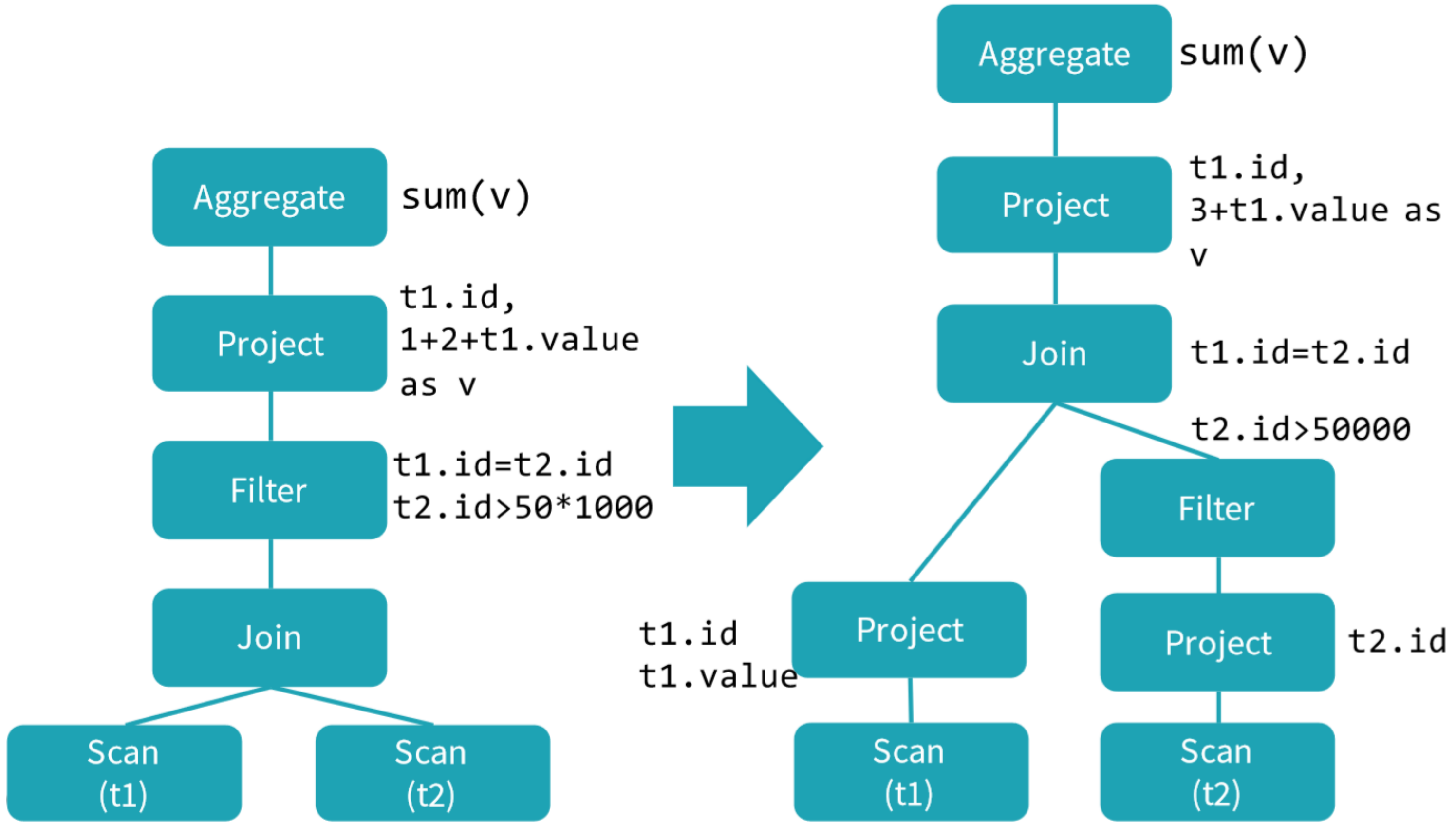
- Catalyst finds out the most efficient plan to execute





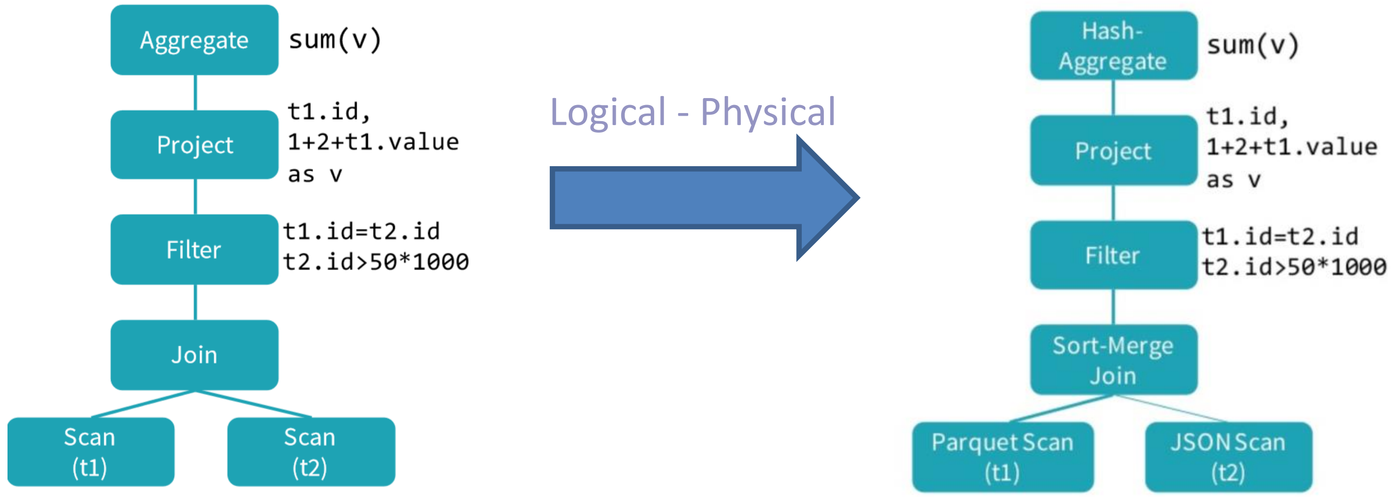
```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```

Spark SQL Architecture

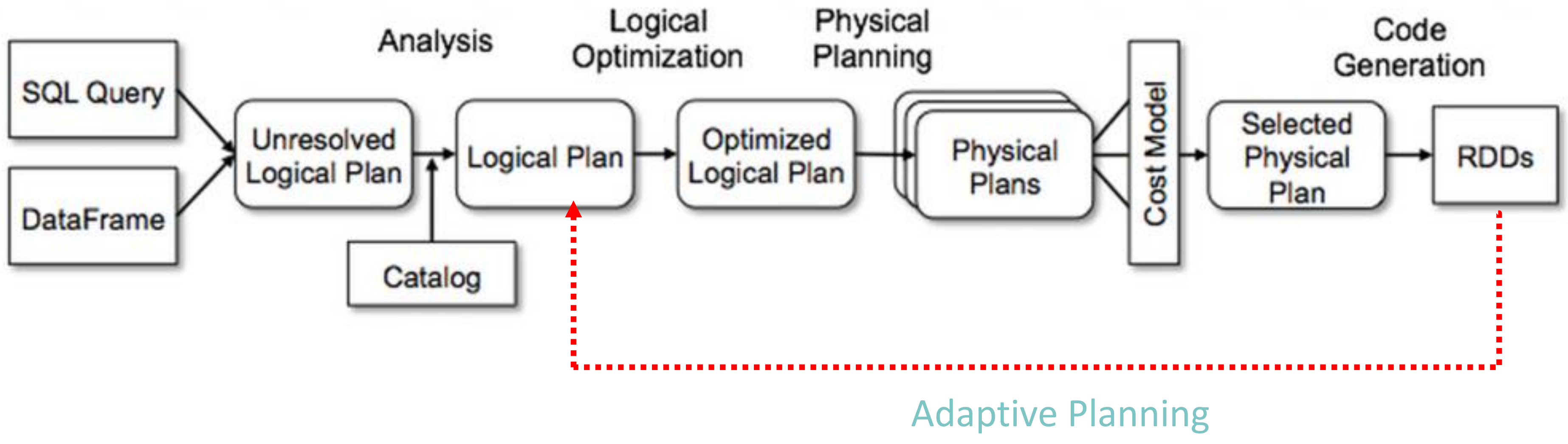


Spark SQL Architecture

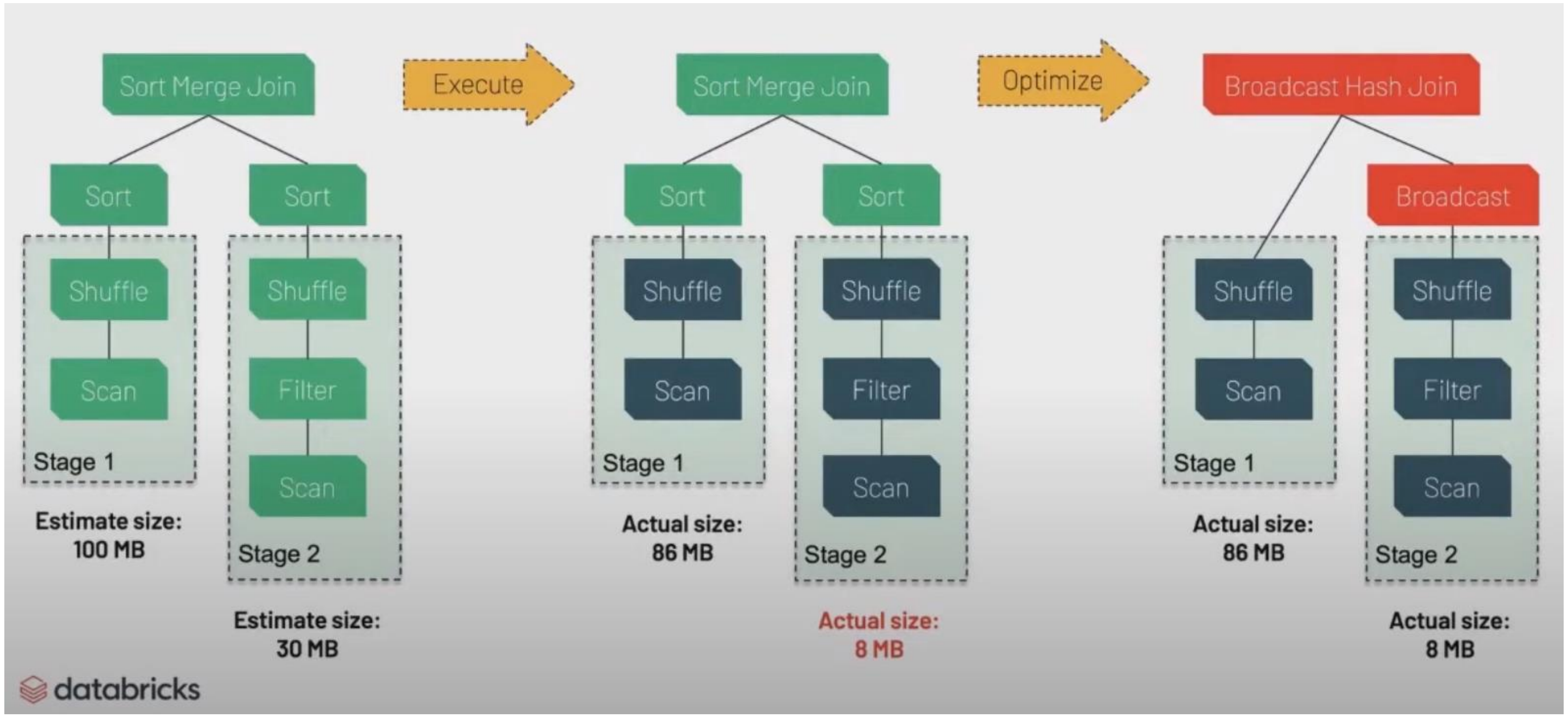
- A logical plan describes computation on datasets without defining how to conduct the computation
- A physical plan describes computation on datasets with specific definitions on how to conduct the computation



Spark 3.0 – Adaptive Planning

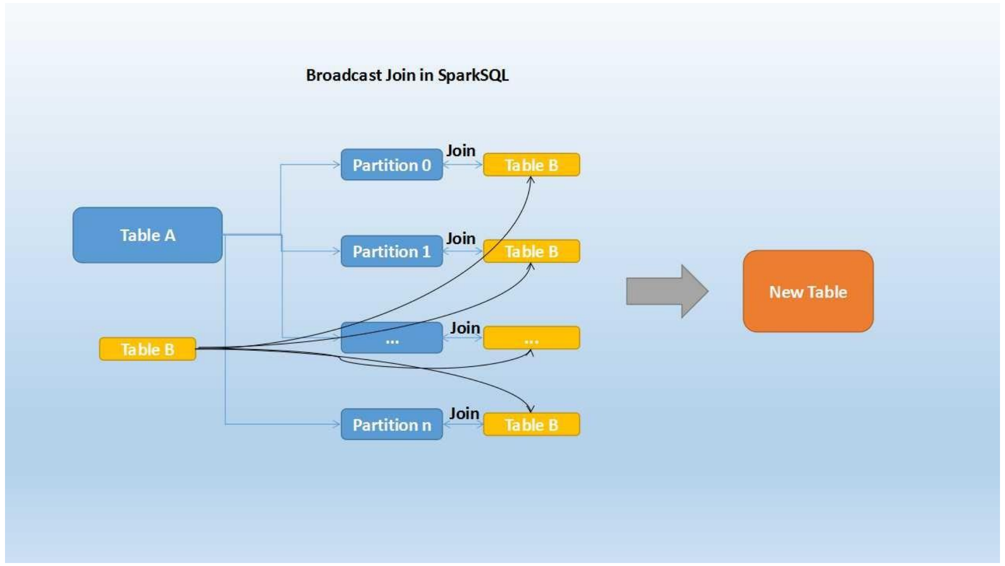


Spark 3.0 – Adaptive Planning



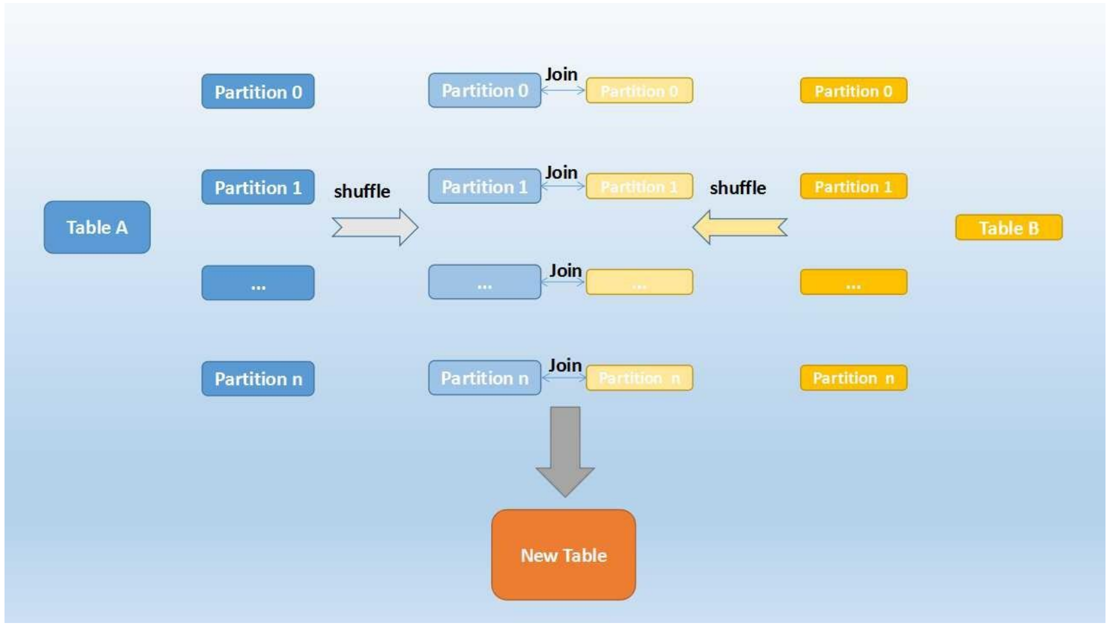
Broadcast Hash Join

- Big/Medium table – Small table
- One table is small enough to be replicated for each executor
- The main idea is to avoid the shuffle
- Small Table needs to be broadcasted less than *spark.sql.autoBroadcastJoinThreshold* (10M) or adding *broadcast* hint



Shuffled Hash Join

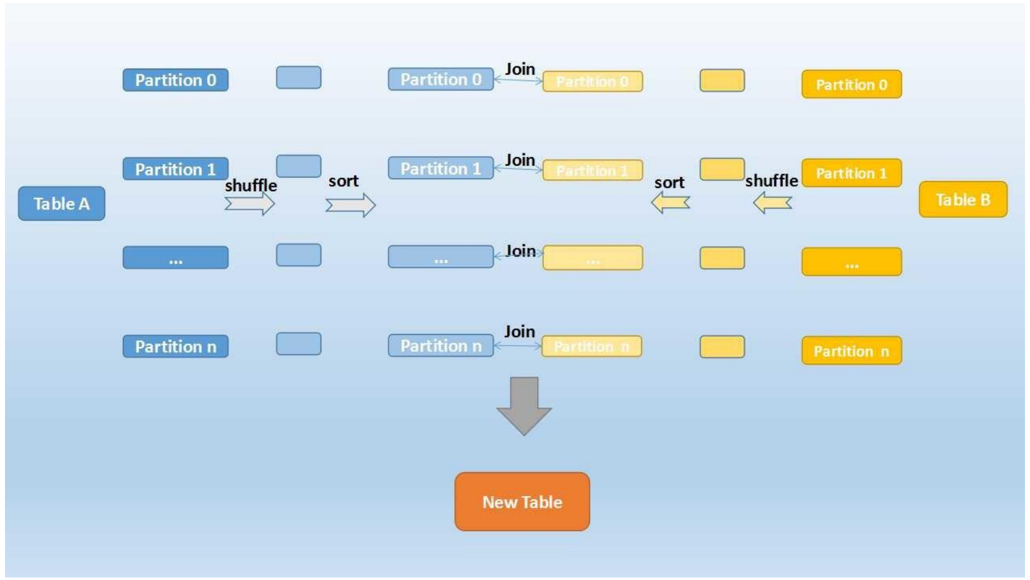
- **Shuffled Hash Join** is the default implementation of a join in Spark
- This join needs to fit a hash table in memory
- Memory issues if the smaller table is not small enough



EDEM
Escuela de Empresarios

Sort Merge Join

- This is the standard join when both tables are large
- Data is sorted before the join
- Comparing with shuffle hash join, this join could spill to disk
- *spark.sql.join.preferSortMergeJoin property* is true by default



Join – Examples

Table **A**

```
+---------+---+
|     name| id|
+---------+---+
|   Pirate|  1|
|   Monkey|  2|
|    Ninja|  3|
|Spaghetti|  4|
+---------+---+
```

Table **B**

```
+-----------+---+
|       name| id|
+-----------+---+
|   Rutabaga|  1|
|     Pirate|  2|
|      Ninja|  3|
|Darth Vader|  4|
+-----------+---+
```

INNER JOIN

```
inner_join = ta.join(tb, ta.name == tb.name)
inner_join.show()
```

```
+------+---+------+---+
|  name| id|  name| id|
+------+---+------+---+
| Ninja|  3| Ninja|  3|
|Pirate|  1|Pirate|  2|
+------+---+------+---+
```

LEFT JOIN

```
left_join = ta.join(tb, ta.name == tb.name,how='left') # Could also use 'left_outer'
left_join.show()
```

```
+---------+---+------+----+
|     name| id|  name|  id|
+---------+---+------+----+
|Spaghetti|  4|  null|null|
|    Ninja|  3| Ninja|   3|
|   Pirate|  1|Pirate|   2|
|   Monkey|  2|  null|null|
+---------+---+------+----+
```

FULL OUTER JOIN

```
full_outer_join = ta.join(tb, ta.name == tb.name,how='full') # Could also use 'full_outer'
full_outer_join.show()
```

```
+---------+----+-----------+----+
|     name|  id|       name|  id|
+---------+----+-----------+----+
|     null|null|   Rutabaga|   1|
|Spaghetti|   4|       null|null|
|    Ninja|   3|      Ninja|   3|
|   Pirate|   1|     Pirate|   2|
|   Monkey|   2|       null|null|
|     null|null|Darth Vader|   4|
+---------+----+-----------+----+
```

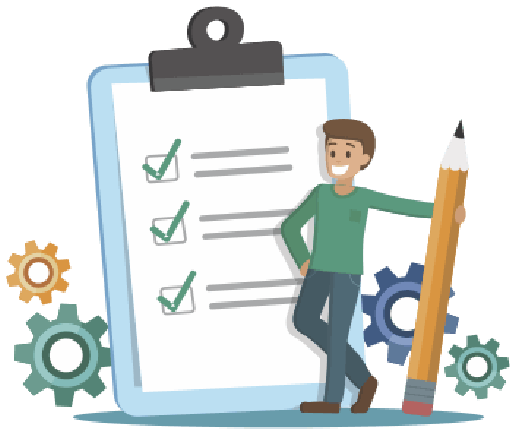
RIGHT JOIN

```
right_join = ta.join(tb, ta.name == tb.name,how='right') # Could also use 'right_outer'
right_join.show()
```

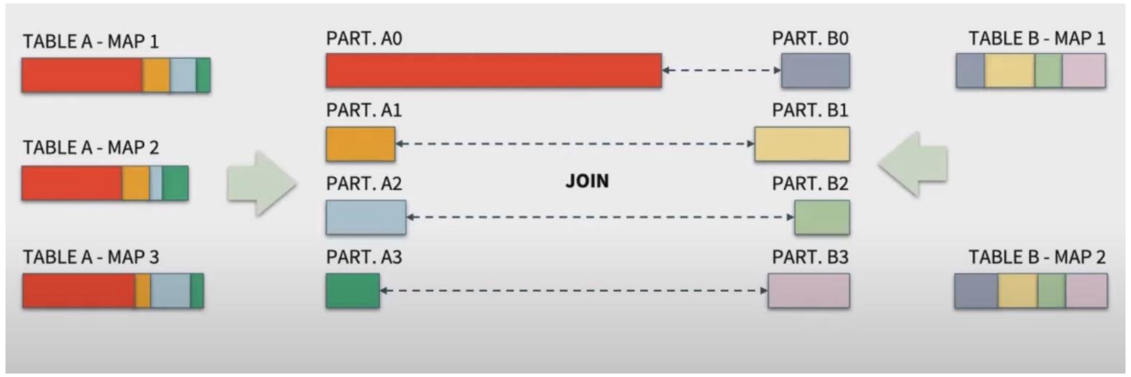
```
+------+----+-----------+---+
|  name|  id|       name| id|
+------+----+-----------+---+
|  null|null|   Rutabaga|  1|
| Ninja|   3|      Ninja|  3|
|Pirate|   1|     Pirate|  2|
|  null|null|Darth Vader|  4|
+------+----+-----------+---+
```

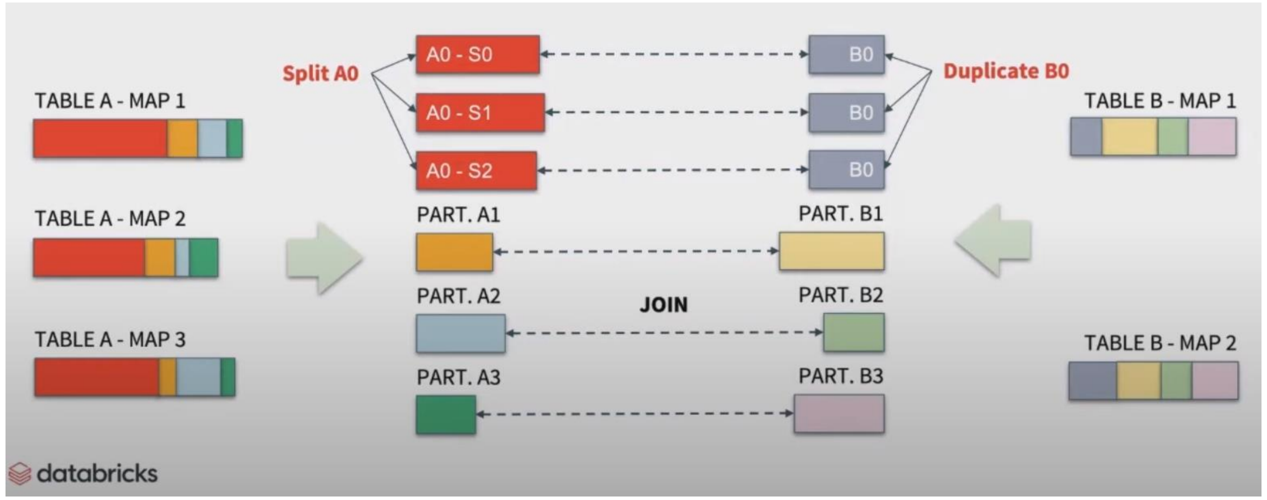
Hands-on

- Open “02.Spark_SQL_Advanced.ipynb” in Google Colab:
  - Try exercises 3, 4, 5, 6 and 7



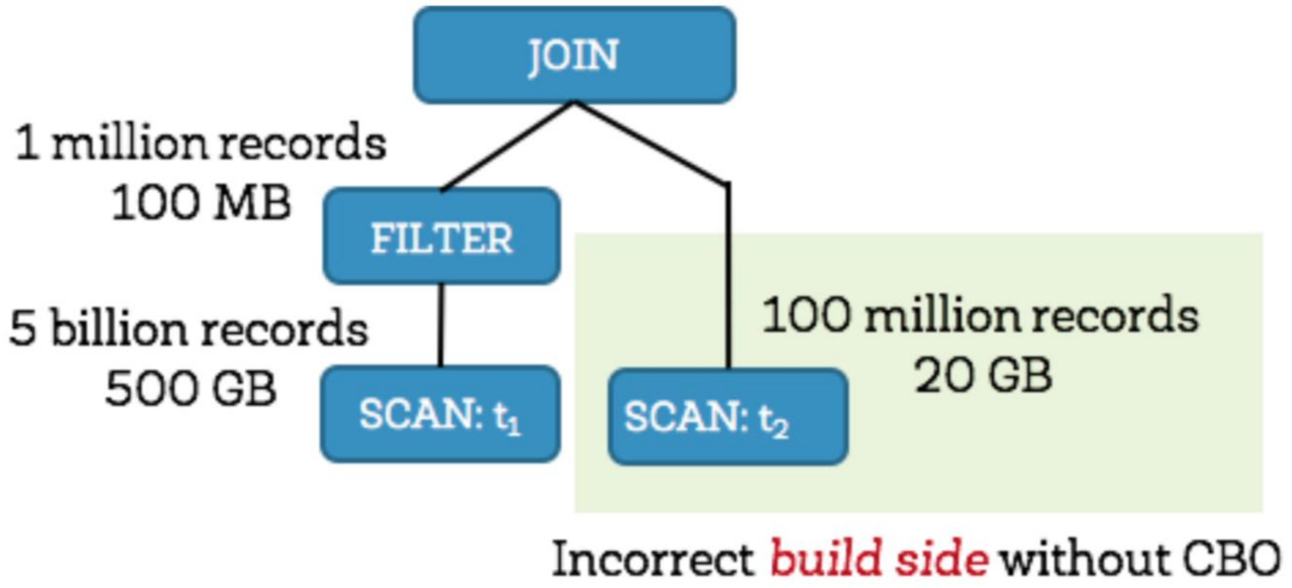
Spark 3.0 – Data Skew Optimization





CBO – 2.2.0

- Spark implements this query using a hash join by choosing the smaller join relation as the build side

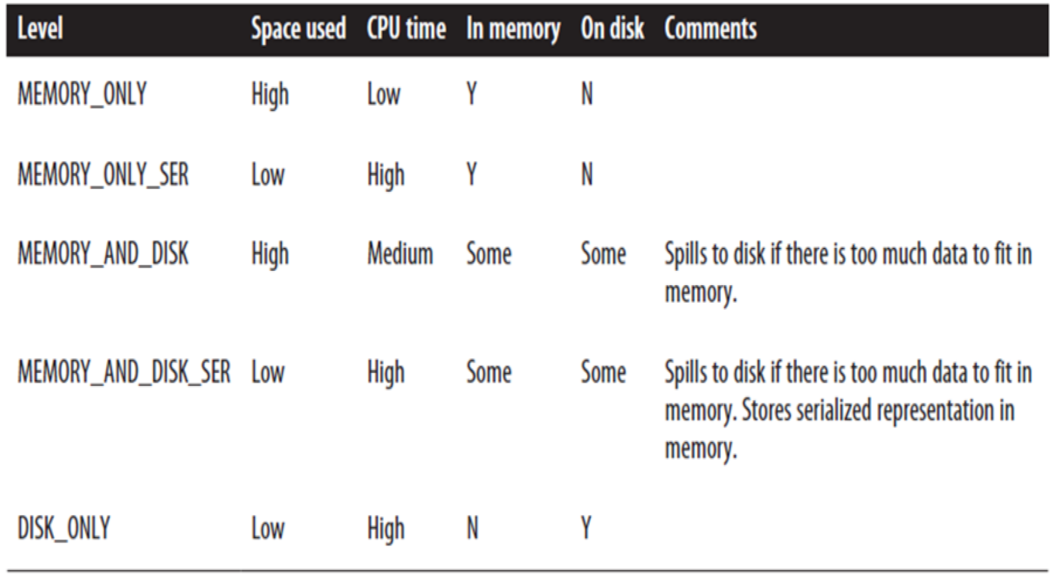


CBO – 2.2.0

- CBO relies on detailed statistics to optimize a query plan
- To collect these statistics, users can issue these new SQL commands:
  - **ANALYZE TABLE table_name COMPUTE STATISTICS**
  - This SQL statement can collect table level statistics such as number of rows and table size in bytes
  - **ANALYZE TABLE table_name COMPUTE STATISTICS FOR COLUMNS column-name1, column-name2, ….**
- Not necessary to specify every column of a table in the ANALYZE statement—only those that are used in a filter/join condition, or in group by clauses etc

DataFrames – Caching

- Cache()
  - Cache in **Memory only**
- Persist()
  - Used to store it to **user-defined storage level**
- UnPersist()
  - Drop Spark DataFrame from Cache



Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Spark SQL – Caching

- Caching in Spark SQL works different:
  - You can also cache using HiveSQL statements

```
CACHE TABLE tableName;
UNCACHE TABLE tableName;
```

- When caching a table Spark SQL represents the data in an in-memory columnar format (Parquet-like)

- The cached table will remain in memory only for the life of our driver program