2. SPARK SQL

Apache Spark - 2022

EDEM

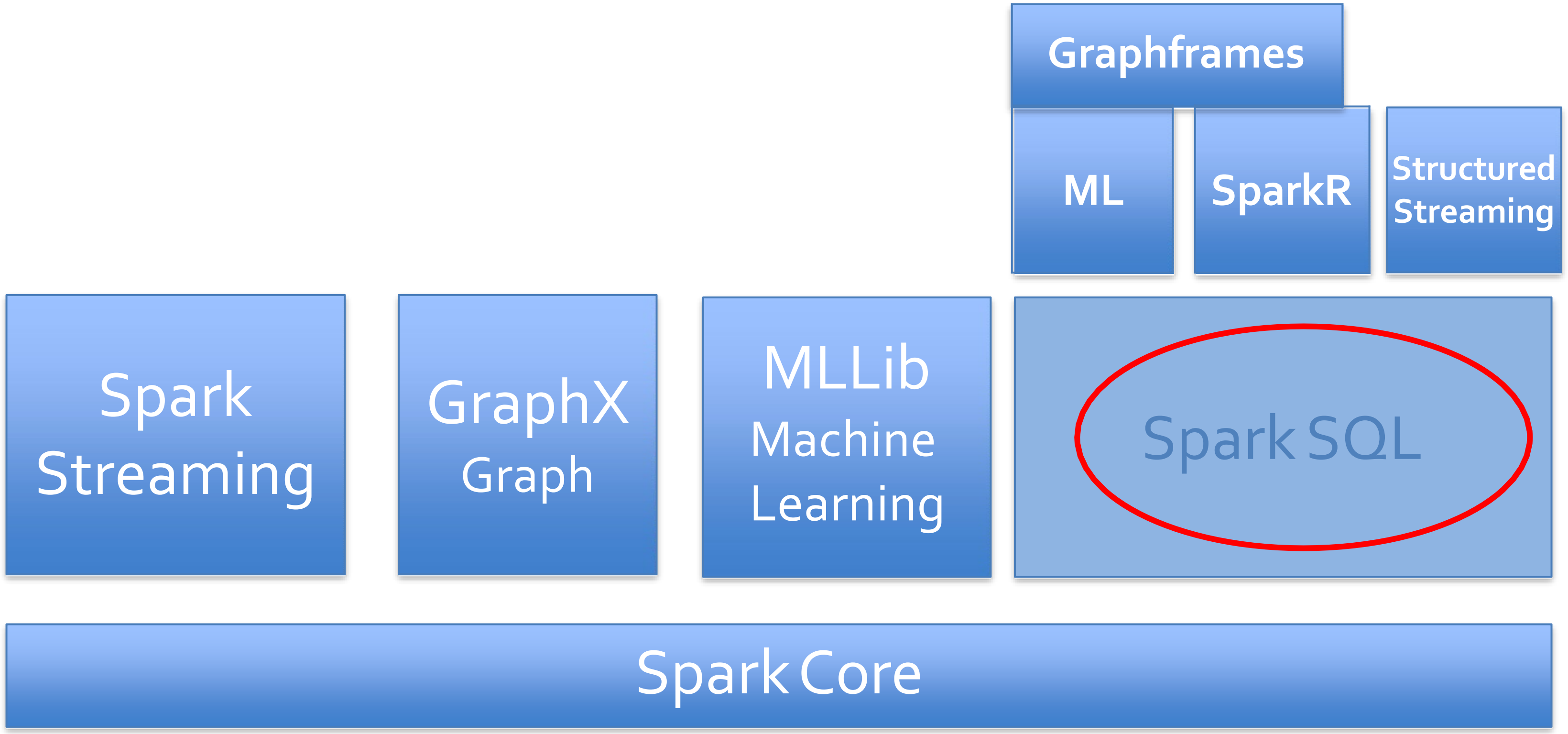
Escuela de Empresarios

EDEM
Escuela de Empresarios

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**

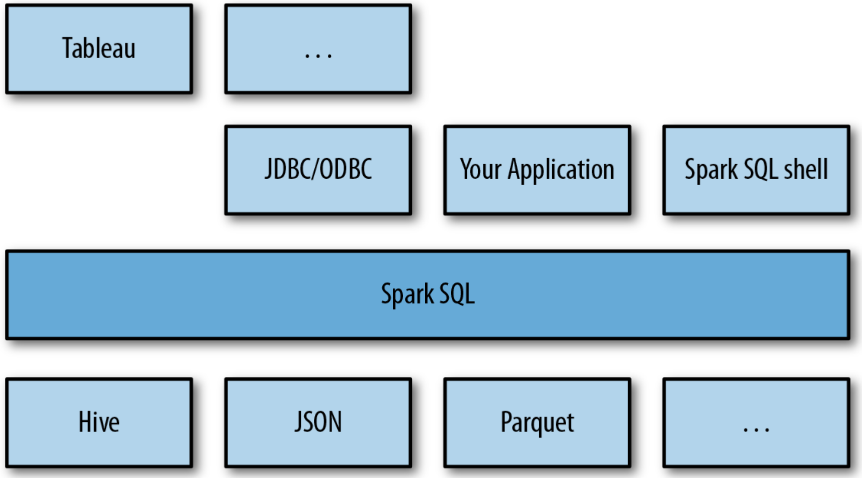


Introduction to Spark SQL

- 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



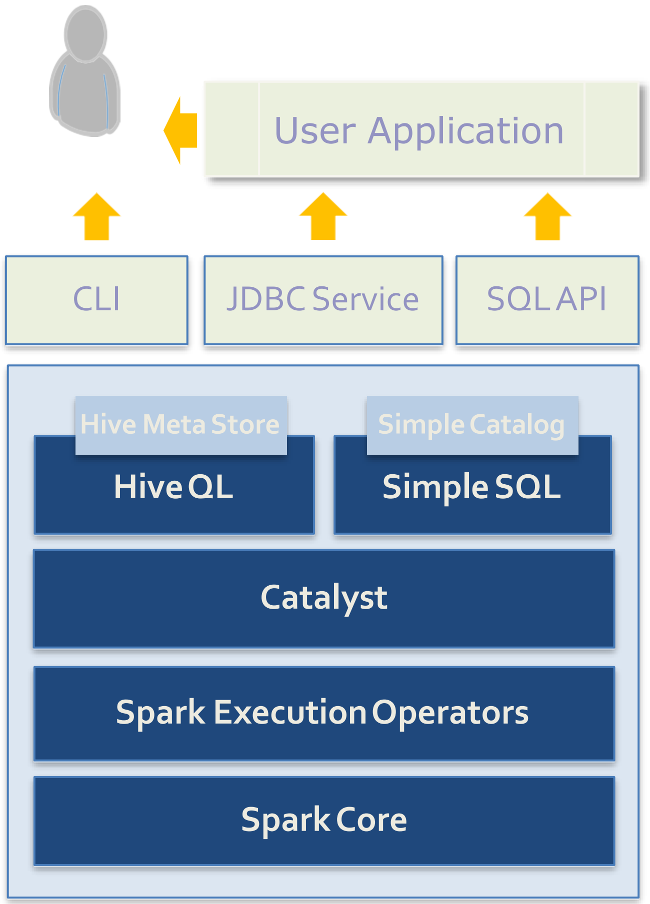
Introduction to Spark SQL

- 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:

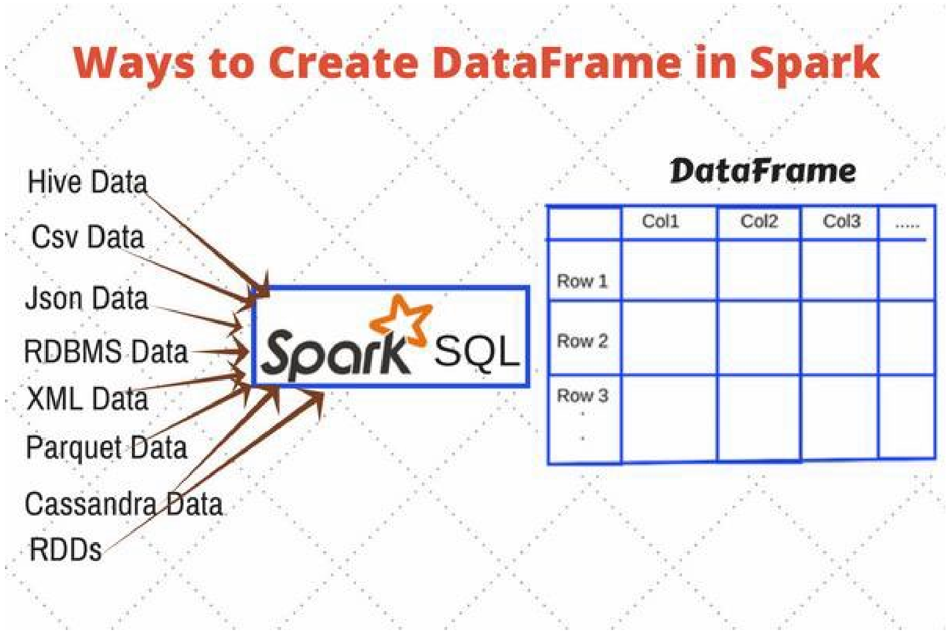
- Hive-like interface(JDBC Service / CLI)
- SQL API support (LINQ-like)
- Both HiveQL & Simple SQL dialects are Supported
- DDL is 100% compatible with Hive Metastore
- HiveQL aims to 100% compatible with Hive DML
- Simple SQL dialect is now very weak in functionality, but easy to extend



1. Introduction to Spark SQL
2. DataFrame & Datasets
3. Spark SQL. Applications
4. Windows Partitioning
5. Catalyst
6. Joins in Spark SQL
7. Cost-Based Optimizer
8. Caching

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
  - …



History of Spark APIs

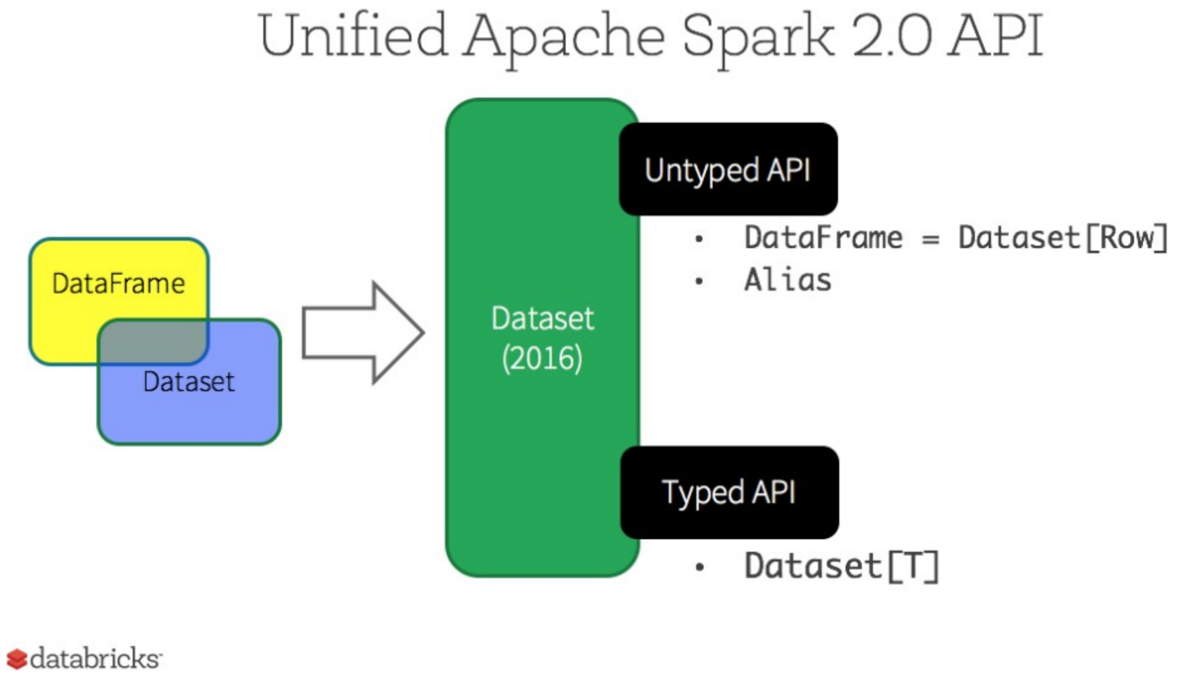
RDD (2011) → **DataFrame (2013)** → **DataSet (2015)**

RDD (2011)	DataFrame (2013)	DataSet (2015)
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	

databricks

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

```
1 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

```
> 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 }
```

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

- Run “DataFramesBasics.ipynb” (Professor)

- Open “Exercises_01_DFBasics.ipynb” in Google Colab:
    - Try DataFrames Basics Exercises 1 and 2

Datasets vs DataFrames

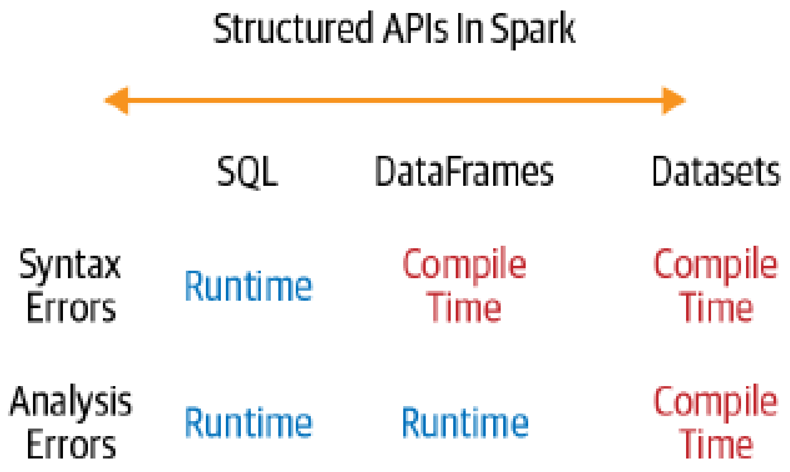
In many cases either will work, depending on the languages you are working in. However there are some situations where one is preferable to the other.

Here are a few examples:

- If you want strict compile-time type safety and don't mind creating multiple case classes for a specific Dataset[T], use Datasets
- If your processing dictates relational transformations similar to SQL-like queries, use DataFrames
- If you want to take advantage of and benefit from Tungsten's efficient serialization with Encoders, use Datasets
- If you want unification, code optimization, and simplification of APIs across Spark components, use DataFrames
- If you are an R user, use DataFrames
- If you are a Python user, use DataFrames and drop down to RDDs if you need more control.
- If you want space and speed efficiency, use DataFrames

Datasets vs DataFrames

And if you want errors caught during compilation rather than at runtime, choose the appropriate API as depicted in Figure below



1. Introduction to Spark SQL
2. DataFrame & Datasets
3. Spark SQL. Applications
4. Windows Partitioning
5. Catalyst
6. Joins in Spark SQL
7. Cost-Based Optimizer
8. Caching

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

```
spark = SparkSession \
        .builder \
        .appName("DataFramesBasics Exercises") \
        .master("local[*]") \
        .getOrCreate()
```

DataFrame – Basic Operations

- Named columns in DataFrames are conceptually similar to named columns in pandas or R DataFrames, or in an RDBMS table
- Can perform operations on their values using relational or computational expressions

```
carsDF.show(3)
```

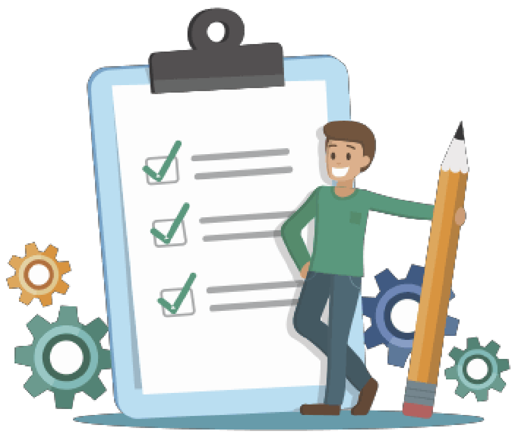
```
+------------+---------+------------+----------+---------------+--------------------+------+-------------+----------+
|Acceleration|Cylinders|Displacement|Horsepower|Miles_per_Gallon|                Name|Origin|Weight_in_lbs|      Year|
+------------+---------+------------+----------+---------------+--------------------+------+-------------+----------+
|        12.0|        8|       307.0|       130|           18.0|chevrolet chevell...|   USA|         3504|1970-01-01|
|        11.5|        8|       350.0|       165|           15.0|   buick skylark 320|   USA|         3693|1970-01-01|
|        11.0|        8|       318.0|       150|           18.0|  plymouth satellite|   USA|         3436|1970-01-01|
+------------+---------+------------+----------+---------------+--------------------+------+-------------+----------+
only showing top 3 rows
```

```
carsDF.select(col("Name")).show(3, False)
```

```
+-------------------------+
|Name                     |
+-------------------------+
|chevrolet chevelle malibu|
|buick skylark 320        |
|plymouth satellite       |
+-------------------------+
only showing top 3 rows
```

Hands-on

- Run “ColumnsAndExpressions.ipynb” (Professor)
- Open “Exercises_01_DFBasics.ipynb” in Google Colab:
  - Try Columns and Expressions Exercises 1 to 3



DataFrame – Aggregations

- A handful of transformations and actions on DataFrames, such as groupBy(), orderBy(), or count() offer the ability to aggregate by column names
- We have also operations like sum, max, or average

```
# using sql functions, NOT including NULLS
genresCountDF = moviesDF.select(count(col("Major_Genre")))
```

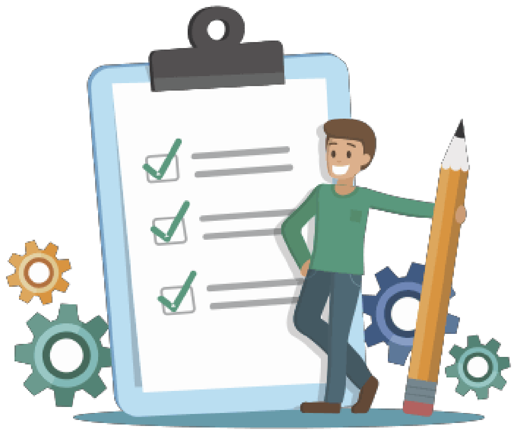
```
+------------------+
|count(Major_Genre)|
+------------------+
|              2926|
+------------------+
```

```
moviesDF.select(sum(moviesDF.US_DVD_Sales).alias("salesUS")).show()
```

```
+-----------+
|    salesUS|
+-----------+
|19684472405|
+-----------+
```

Hands-on

- Run “Aggregations.ipynb” (Professor)
- Open “Exercises_02_SQL.ipynb” in Google Colab:
  - Try Aggregations Exercises 1 to 5



DataFrame – Joins

- A common DataFrame operation is to join two DataFrames (or tables) togeth
- By default, a Spark SQL join is an inner join,
- With the options being inner, cross, outer, full, full_outer, left, left_outer, right, right_outer, left_semi, and left_anti

Table **A**

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

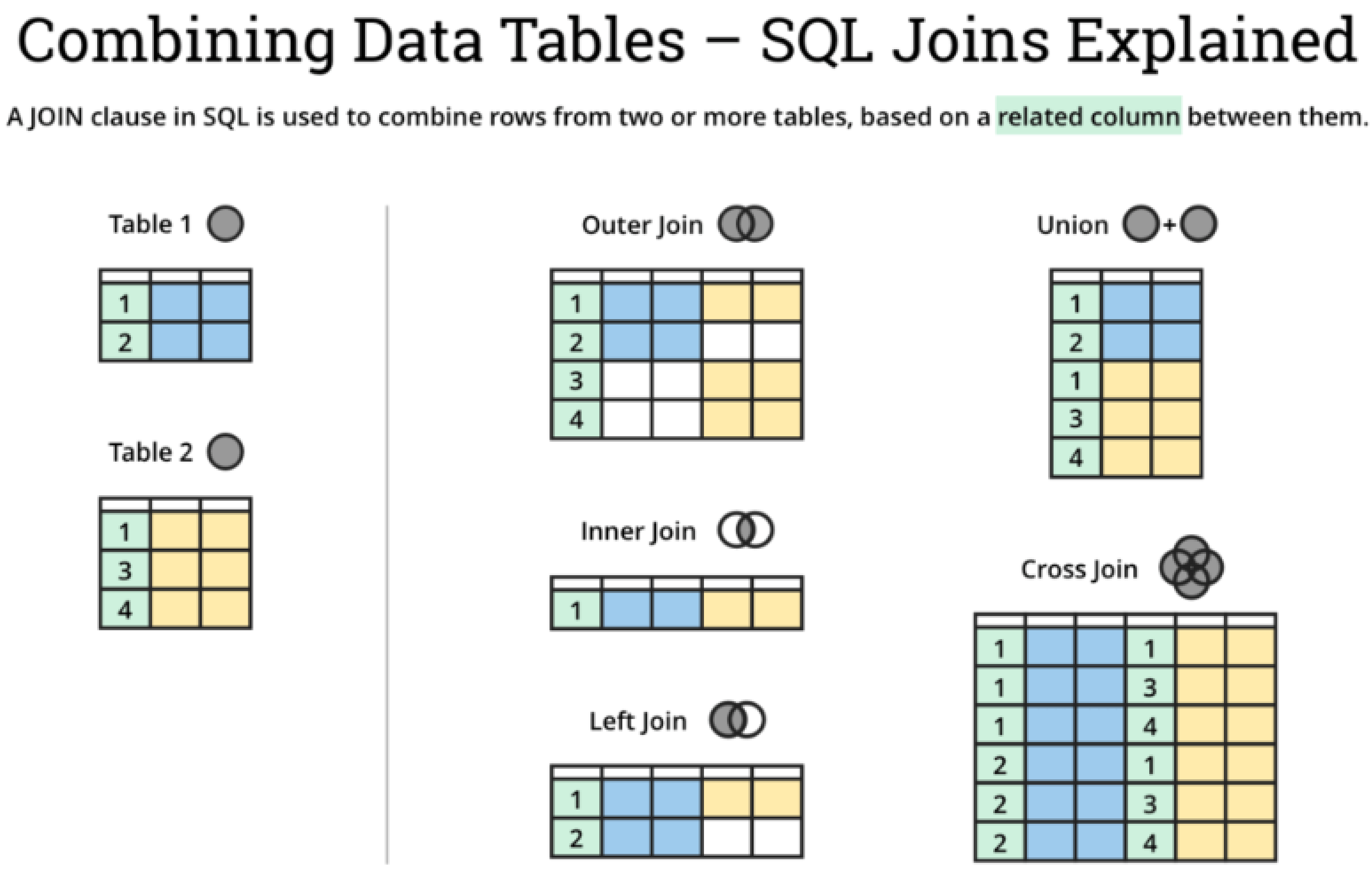
Table **B**

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

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

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

DataFrame – Joins



DataFrame – Joins

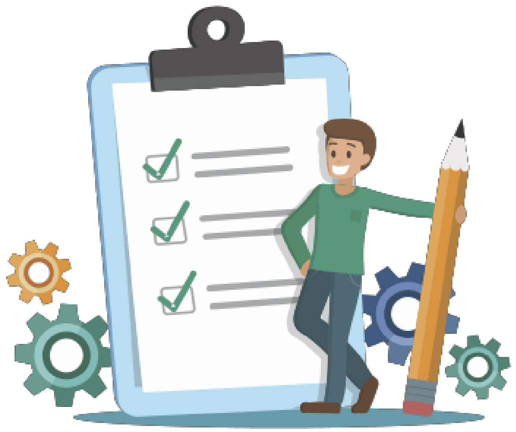
As a brief summary:

- Inner Join = everything from BOTH DFs for which there is a row in the right DF satisfying the condition
- Left Outer = everything in the inner join + all the rows in the LEFT DF, with nulls in where the data is missing
- Right Outer = everything in the inner join + all the rows in the RIGHT DF, with nulls in where the data is missing
- Outer Join = everything in the inner join + all the rows in BOTH DFs, with nulls in where the data is missing
- Semi-joins = everything in the LEFT DF for which there is a row in the right DF satisfying the condition
- Anti-joins = everything in the LEFT DF for which there is NO row in the right DF satisfying the condition
- Cross-join = takes every instance in the LEFT DF and create a new one for every instance in the right DF (not very common)
- Union Join is just an Union (note it only works if both DFs have the same schema)

Hands-on

- Run "Joins.ipynb" (Professor)

- Open "Exercises_02_SQL.ipynb" in Google Colab:
    - Try Joins Exercises 1 to 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`")
```

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
```

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

1. Introduction to Spark SQL
2. DataFrame & Datasets
3. Spark SQL. Applications
4. Windows Partitioning
5. Catalyst
6. Joins in Spark SQL
7. Cost-Based Optimizer
8. Caching

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?"

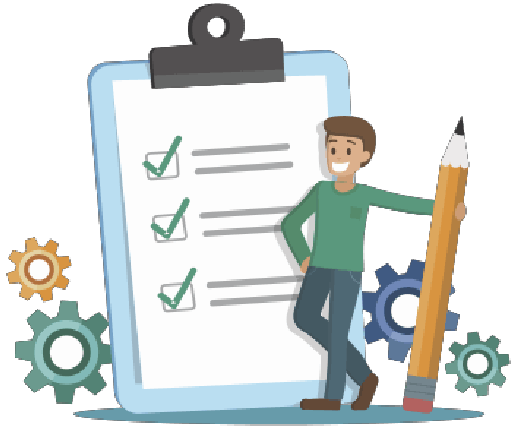
```
import sys
from pyspark.sql.window import Window
import pyspark.sql.functions as func
windowSpec = \
  Window
    .partitionBy(df['category']) \
    .orderBy(df['revenue'].desc()) \
    .rangeBetween(-sys.maxsize, sys.maxsize)
dataFrame = sqlContext.table("productRevenue")
revenue_difference = \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

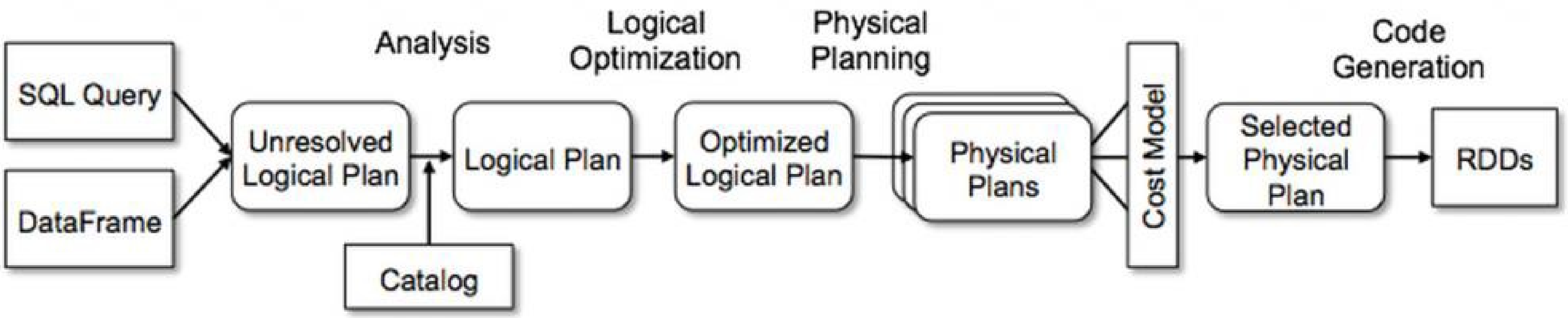
	SQL	DataFrame API
Ranking functions	rank	rank
	dense_rank	denseRank
	percent_rank	percentRank
	ntile	ntile
	row_number	rowNumber
Analytic functions	cume_dist	cumeDist
	first_value	firstValue
	last_value	lastValue
	lag	lag
	lead	lead

Hands-on



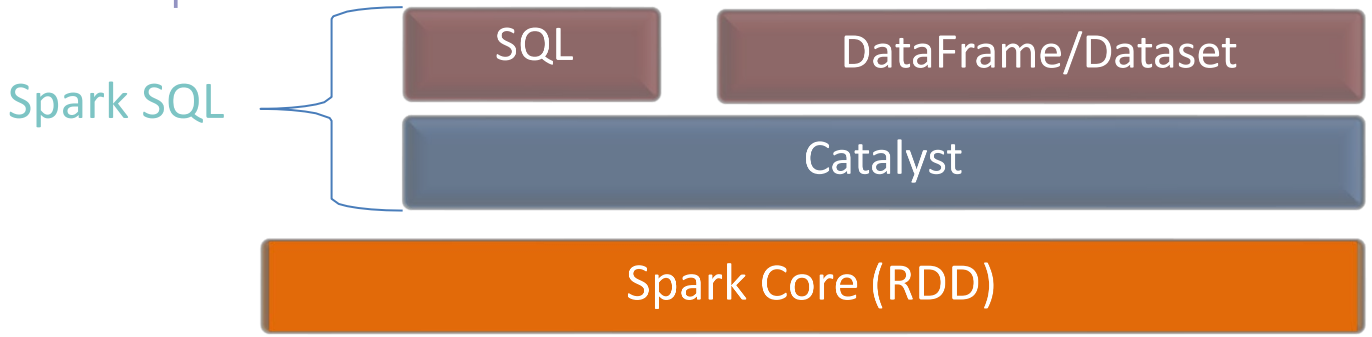
- Run “WindowPartitioning.ipynb” (Professor)
- Open “Exercises_03_WindowPartitioning.ipynb” in Google Colab:
  - Try exercises 1 and 2

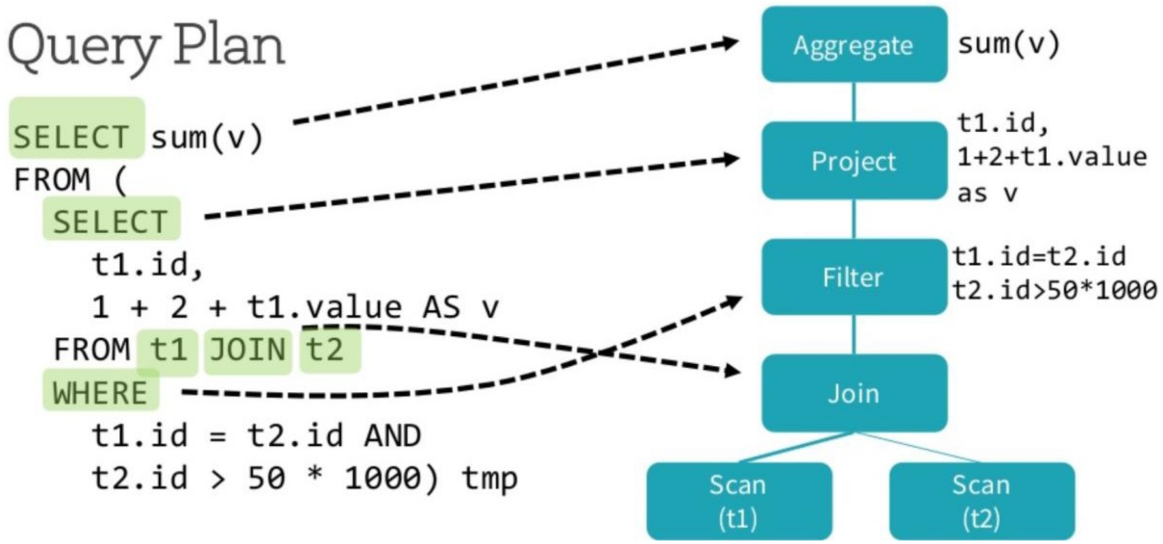
Spark SQL Architecture



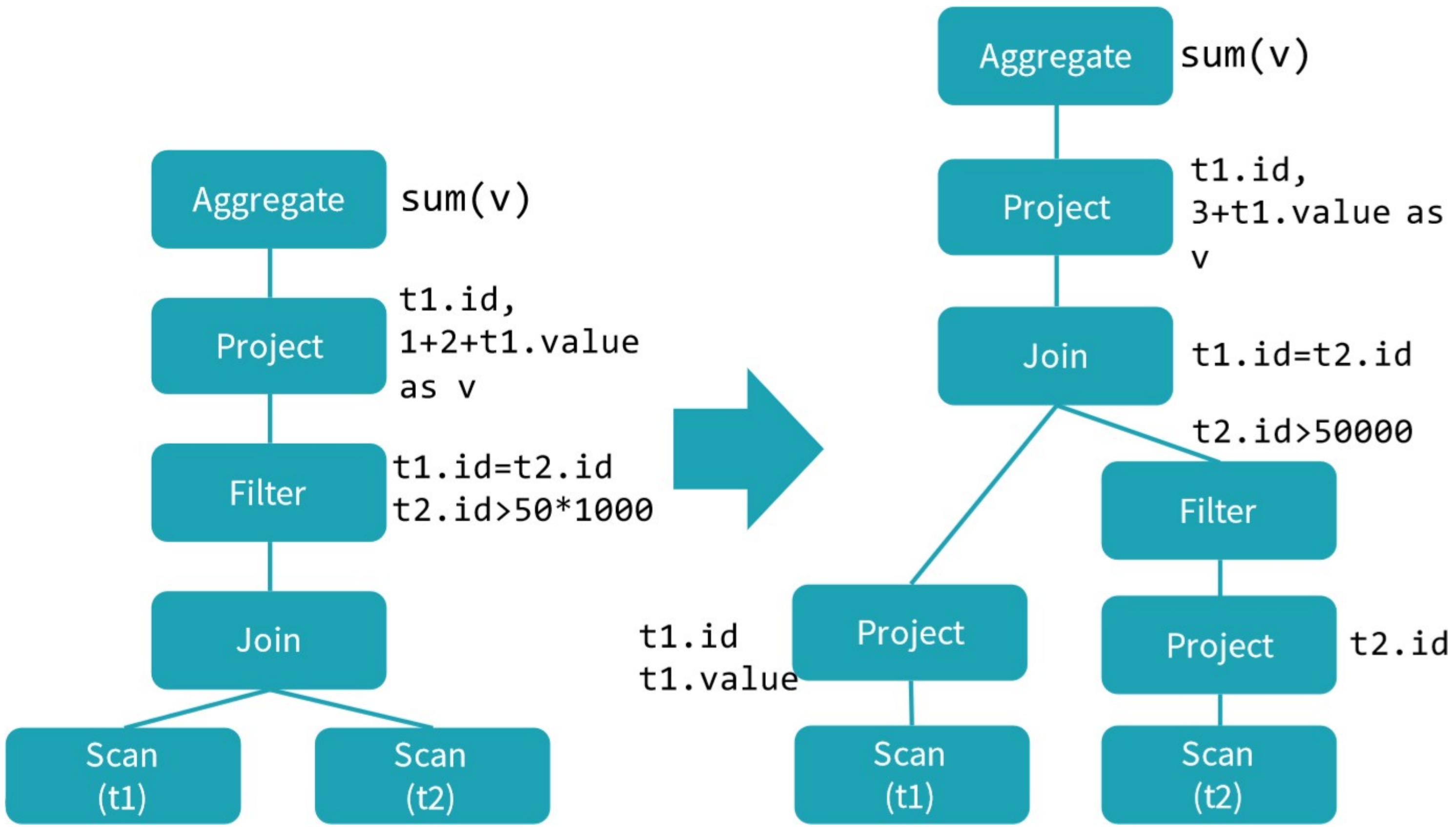
Catalyst

- Catalyst finds out the most efficient plan to execute



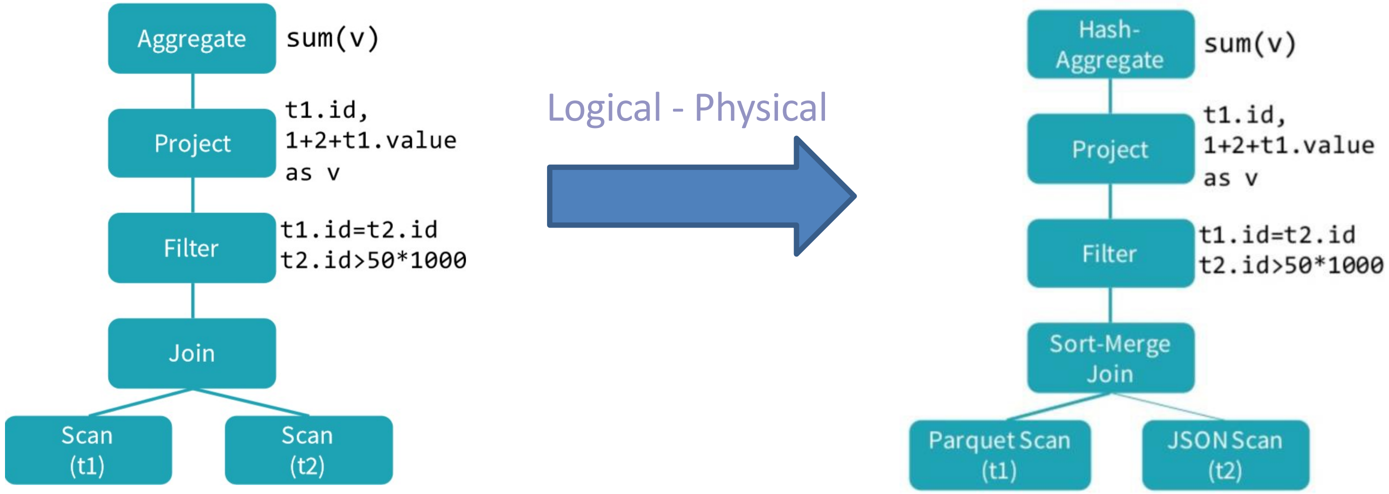


Spark SQL Architecture

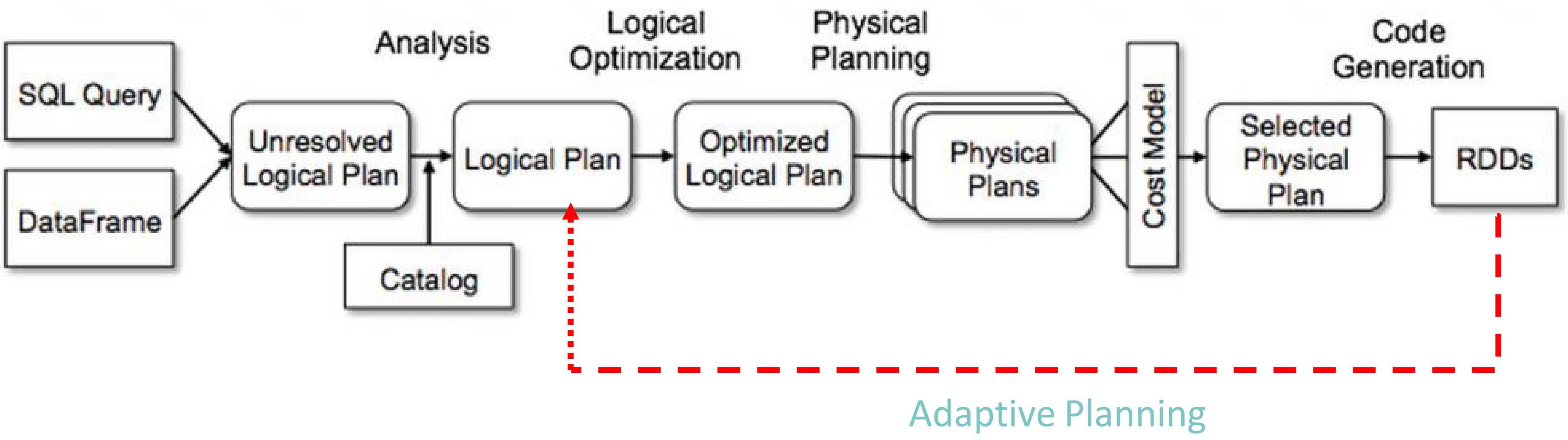


Spark SQL Architecture

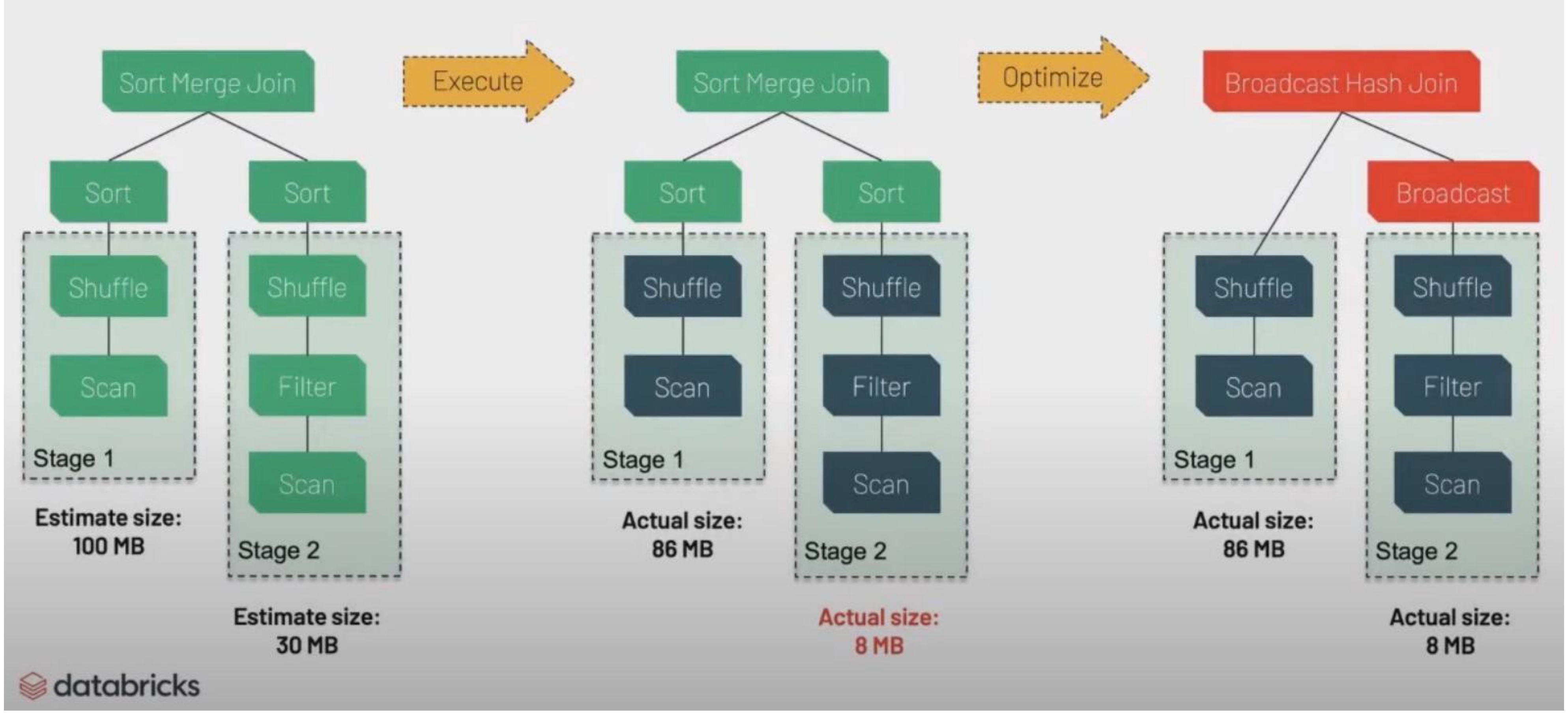
- A logical plan describes computation on datasets without defining how to conduct the computation
- A physical plan describes computation on datasets with specific definitions on how to conduct the computation



Spark 3.0 – Adaptive Planning



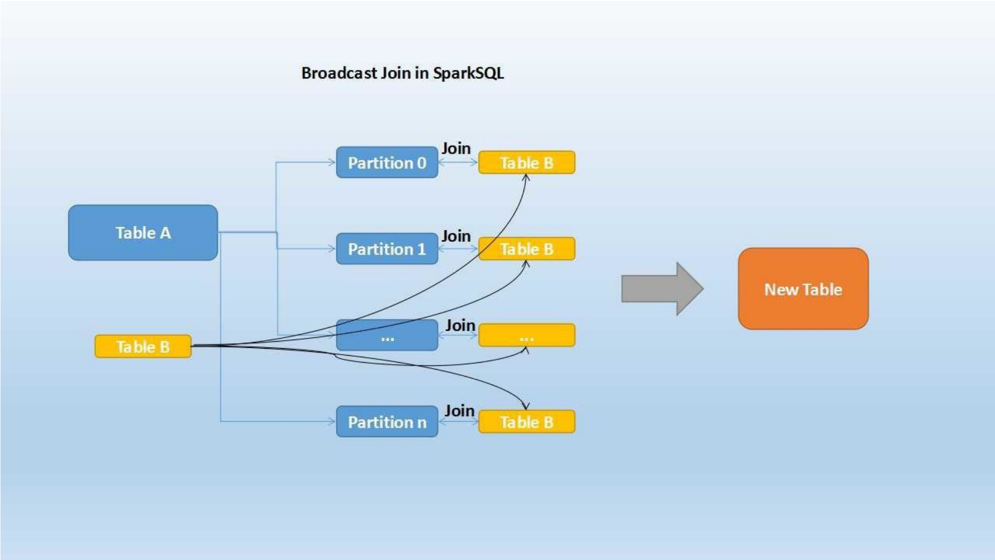
Spark 3.0 – Adaptive Planning



1. Introduction to Spark SQL
2. DataFrame & Datasets
3. Spark SQL. Applications
4. Windows Partitioning
5. Catalyst
6. Joins in Spark SQL
7. Cost-Based Optimizer
8. Caching

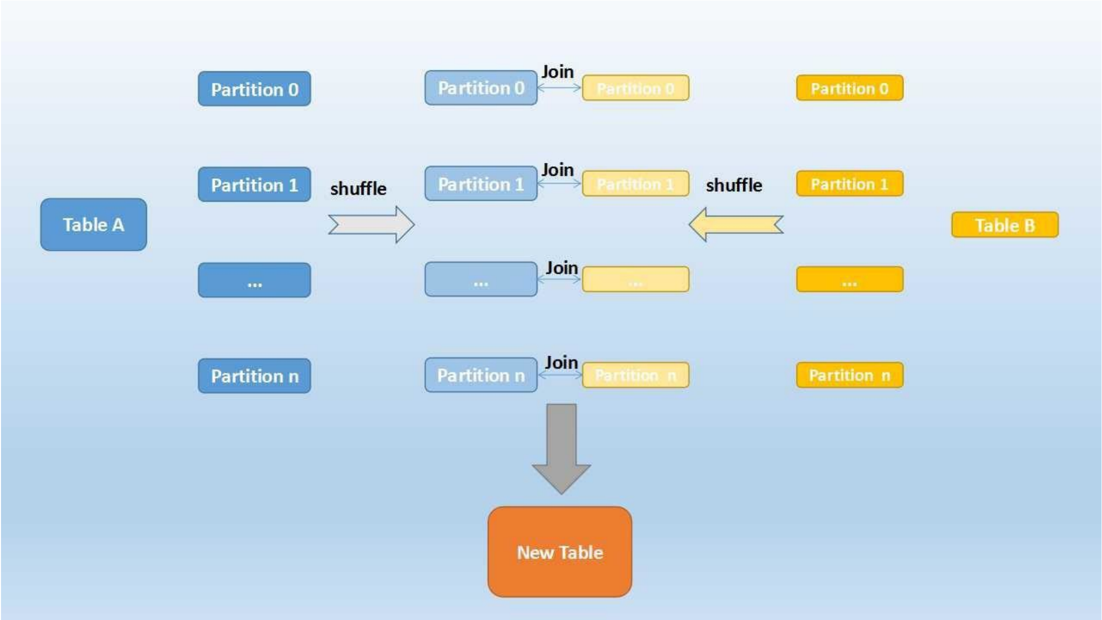
Broadcast Hash Join

- 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



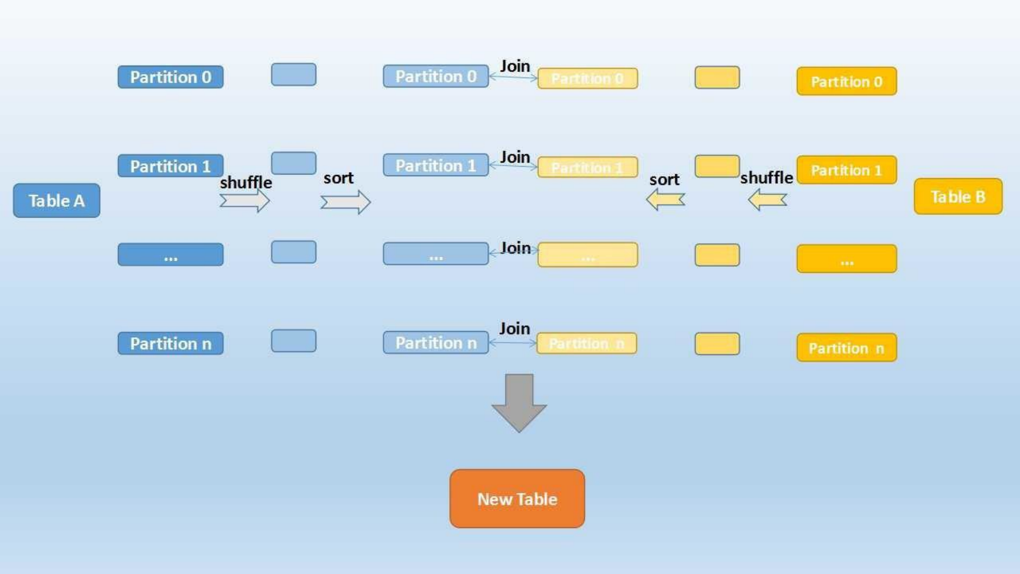
Shuffled Hash Join

- **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

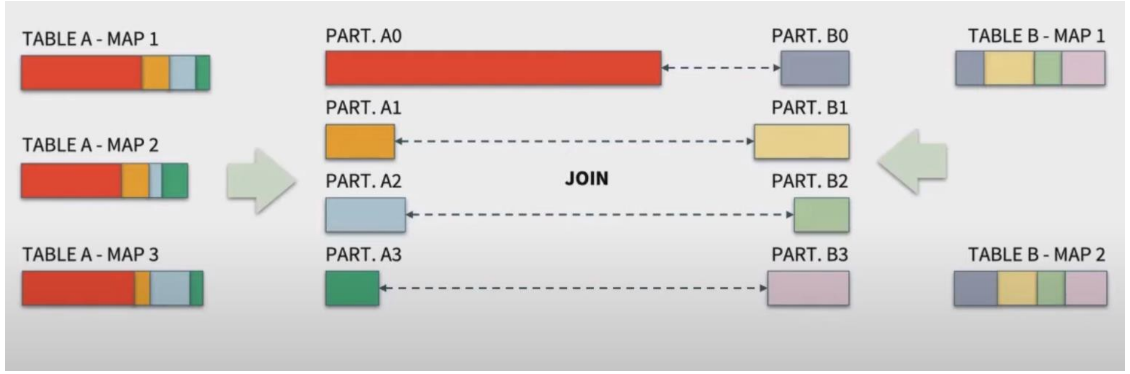


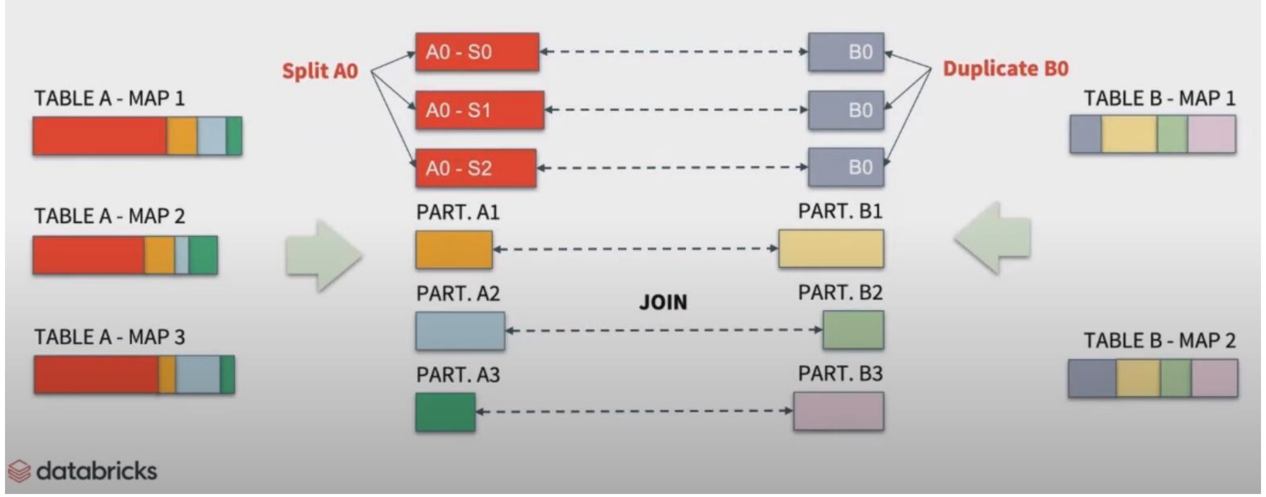
Sort Merge Join

- 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



Spark 3.0 – Data Skew Optimization

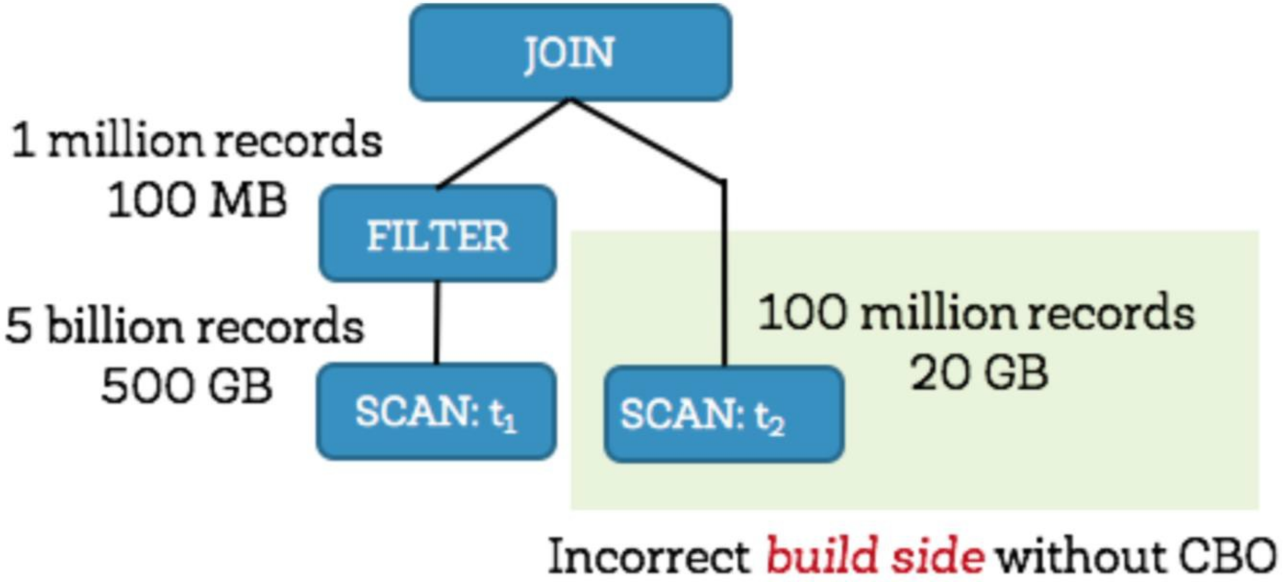




1. Introduction to Spark SQL
2. DataFrame & Datasets
3. Spark SQL. Applications
4. Windows Partitioning
5. Catalyst
6. Joins in Spark SQL
7. Cost-Based Optimizer
8. Caching

CBO – 2.2.0

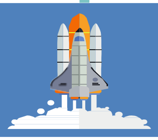
- Spark implements this query using a hash join by choosing the smaller join relation as the build side



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

1. Introduction to Spark SQL
2. DataFrame & Datasets
3. Spark SQL. Applications
4. Windows Partitioning
5. Catalyst
6. Joins in Spark SQL
7. Cost-Based Optimizer
8. Caching



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