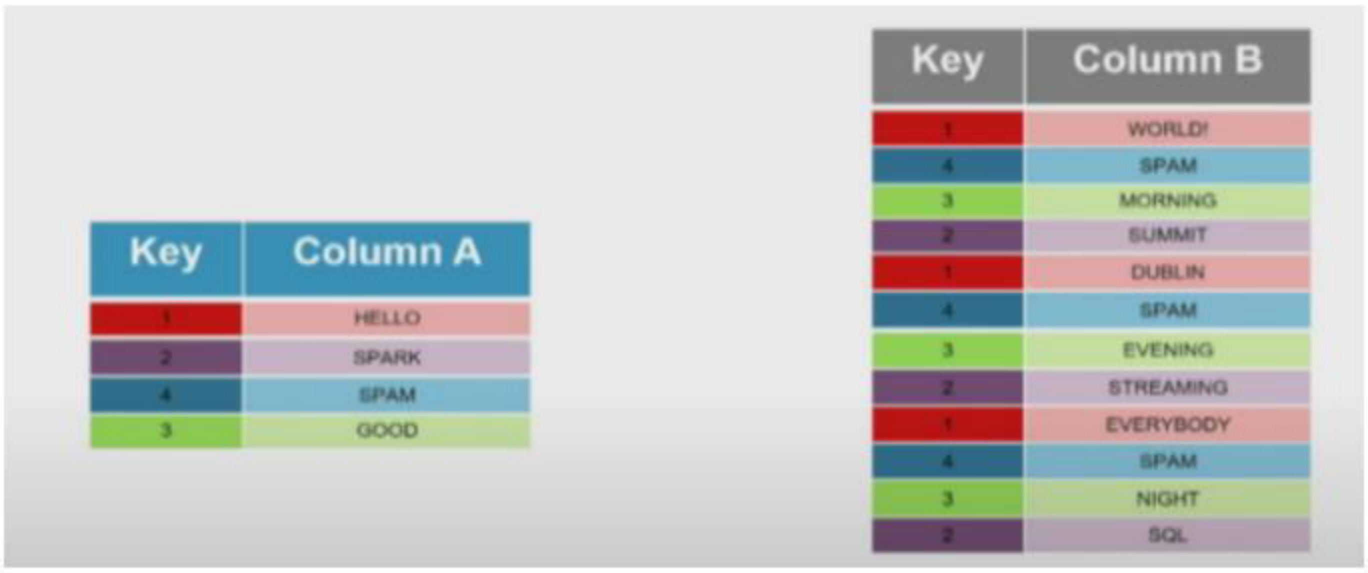
Spark Joins:

- **Parallelization** is Spark's bread and butter. The back bone of Spark Architecture is data should be split into partitions and allocate each piece to an executor in cluster, so multiple executors can work on different pieces of data in parallel.
- **Shuffling:** All the nodes and executors should exchange the data across the network and re-arrange partitions in such a way that each node/executor should receive a specific key data.
- **Joining** two datasets is a heavy operation and needs lots of data movement (shuffling) across the network, to ensure rows with matching join keys get co-located physically (on the same node).

1) Sort Merge Join:

- By **default** Spark uses this method while **joining** data frames. It's **two-step** process.
- First **all executors** should **exchange** data **across network** to **sort** and **re-allocate sorted partitions**.
- At the end of this stage, each executor should have **same key valued data** on **both** DataFrame **partitions** so that **executor** can do **merge** operation.
- **Merge** is very quick thing.

Let's examine this **Sort Merge Join** with an example. Two data frames A and B have four key columns (1, 2, 3, 4) and let's say we have 2 node cluster.

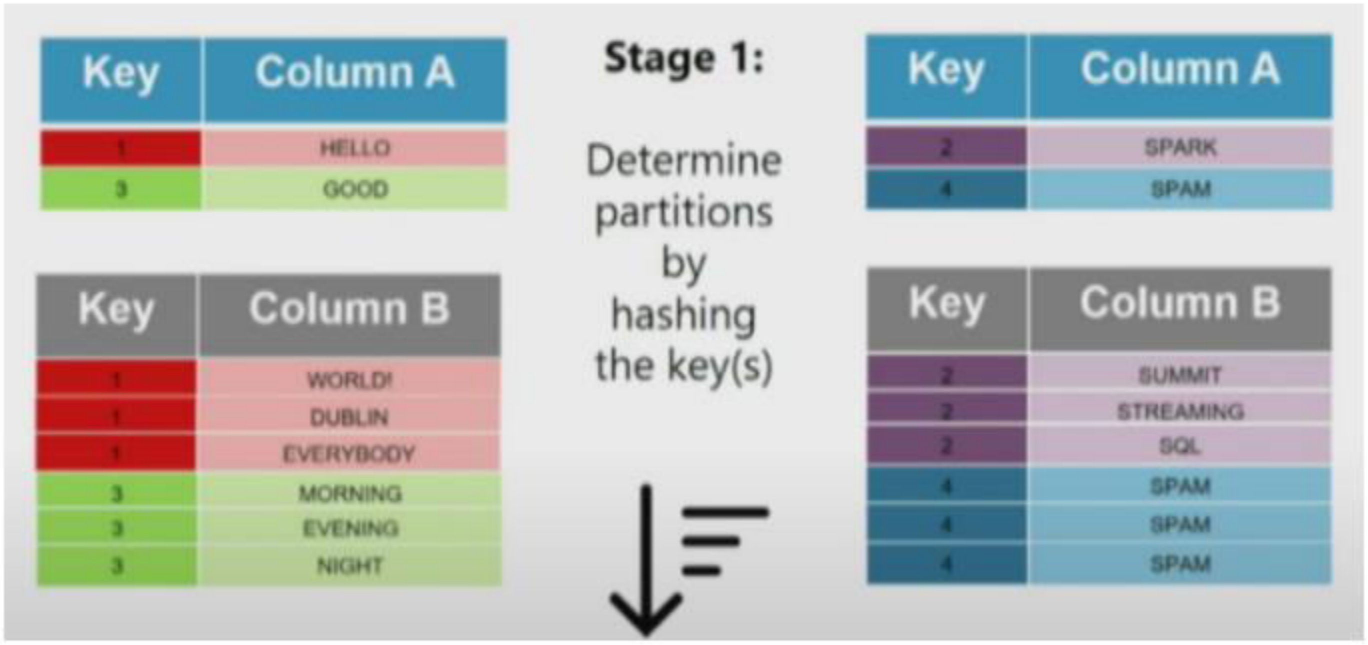


Key	Column A
1	HELLO
2	SPARK
4	SPAM
3	GOOD

Key	Column B
1	WORLD!
4	SPAM
3	MORNING
2	SUMMIT
1	DUBLIN
4	SPAM
3	EVENING
2	STREAMING
1	EVERYBODY
4	SPAM
3	NIGHT
2	SQL

Sort Phase:

- As you can see, both A and B are **sorted** by **Join** key i.e. **Key** column and **sorted data** is split into 2 partitions.
- Each **partition** should have **specific key** data.
- There should **not** be any **overlapping** of **Keys** between **partitions** which is **whole** idea of **shuffling**.



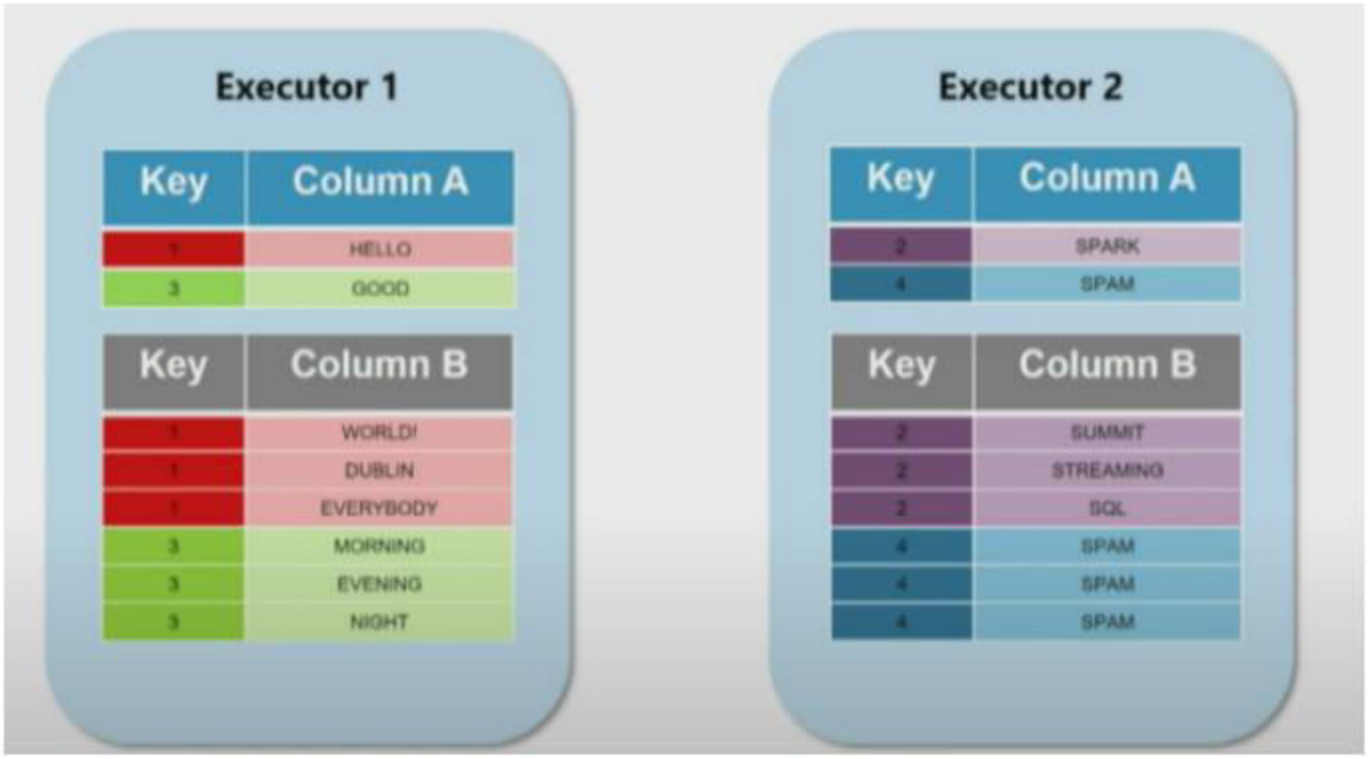
Key	Column A
1	HELLO
3	GOOD

Key	Column B
1	WORLD!
1	DUBLIN
1	EVERYBODY
3	MORNING
3	EVENING
3	NIGHT

Key	Column A
2	SPARK
4	SPAM

Key	Column B
2	SUMMIT
2	STREAMING
2	SQL
4	SPAM
4	SPAM
4	SPAM

Assign Sorted Partitions to Executors:



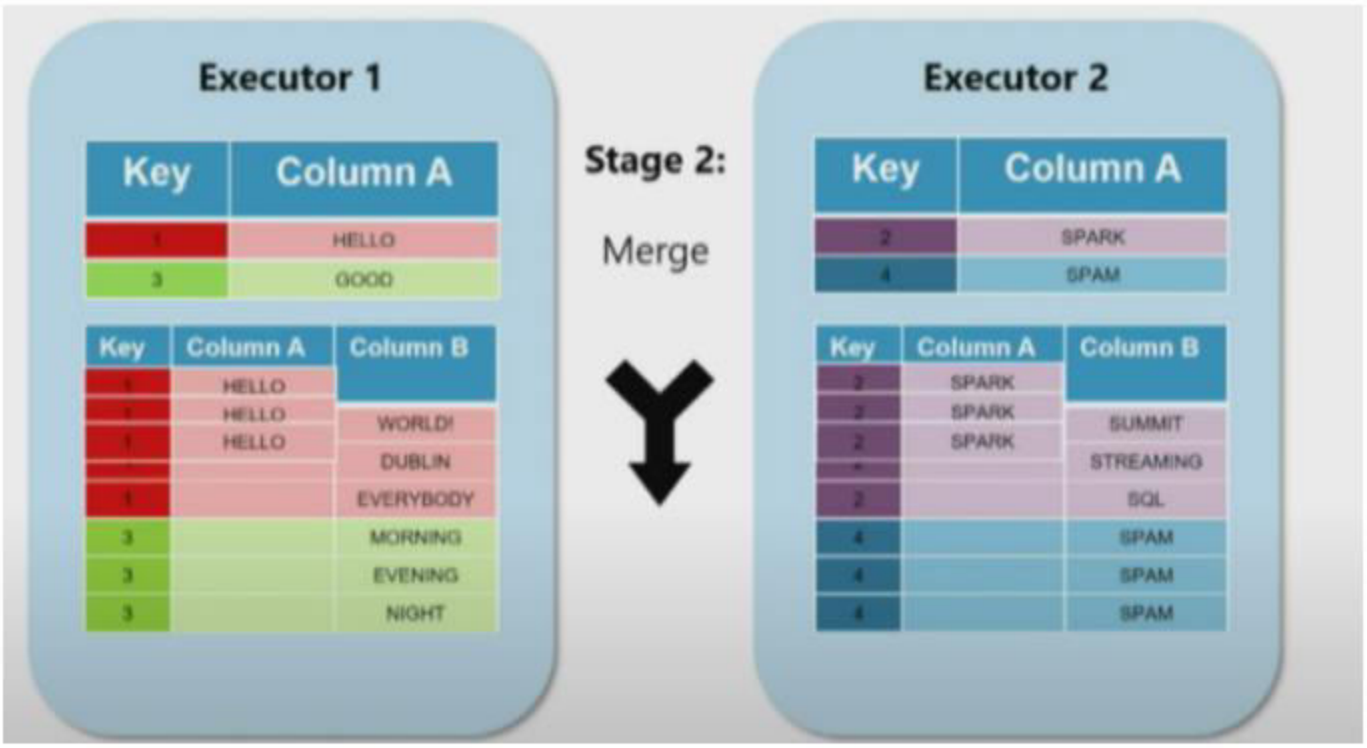
Key	Column A
1	HELLO
3	GOOD

Key	Column B
1	WORLD!
1	DUBLIN
1	EVERYBODY
3	MORNING
3	EVENING
3	NIGHT

Key	Column A
2	SPARK
4	SPAM

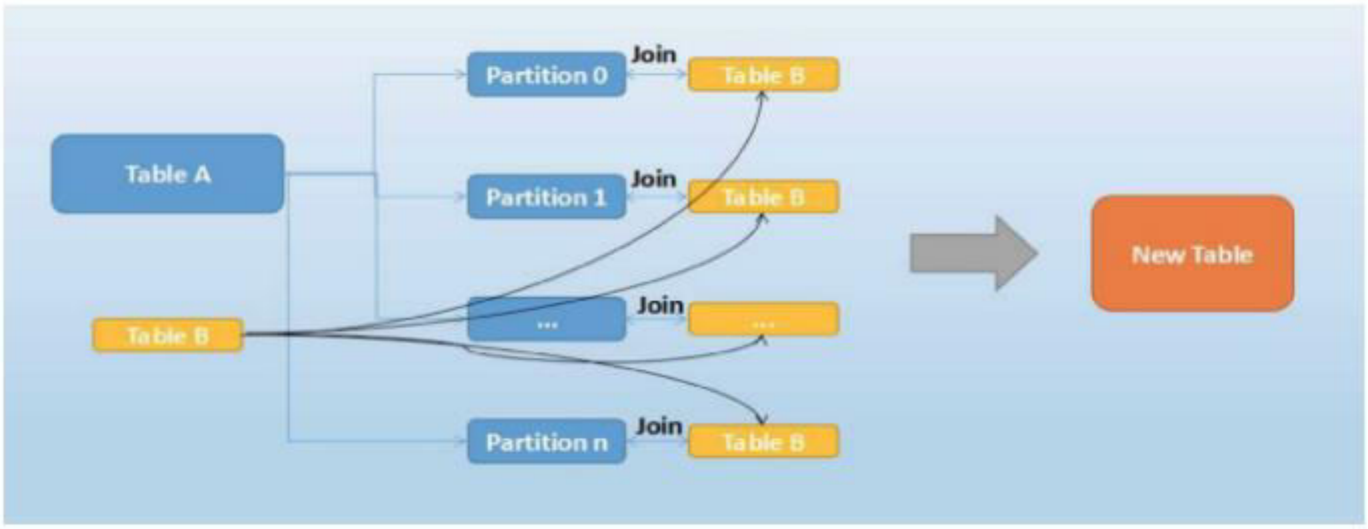
Key	Column B
2	SUMMIT
2	STREAMING
2	SQL
4	SPAM
4	SPAM
4	SPAM

Merge Phase: Merging **sorted** data by **keys** is very simple and quickest operation.



Broad Cast Join:
This type of join strategy is suitable when **one** side of the dataset in the join is fairly **small**. (The threshold can be configured using "**spark. sql. autoBroadcastJoinThreshold**" which is by default **10MB**).

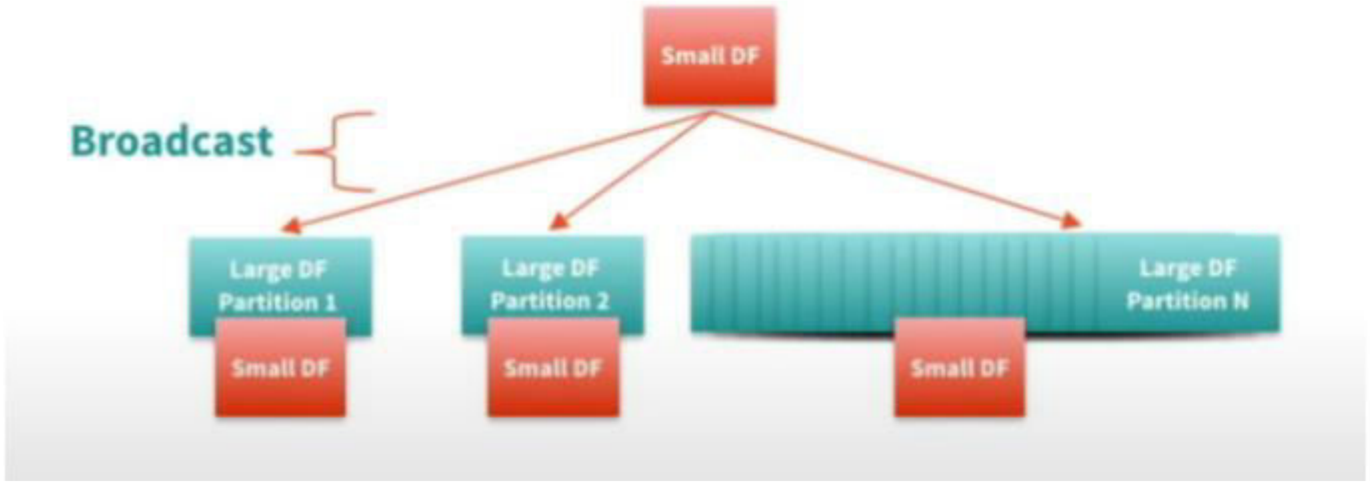
Consider the following example where **Table A** and small **Table B** (less than 10 MB) have to be joined. In this case, the Spark driver **broadcasts Table B** to **all nodes** on the **cluster** where **partitions** of **Table A** are **present**.



Now **Table B** is **present** on **all** the **nodes** where we have **data** for **Table A**, **no more data shuffling** is **required** and each **partition** of **Table A** can **join** with the required entries of **Table B**.

This is the **fastest** type of **join** (as the **bigger** table requires **no data shuffling**) but has the **limitation** that **one** table in the **join** has to be **small**.

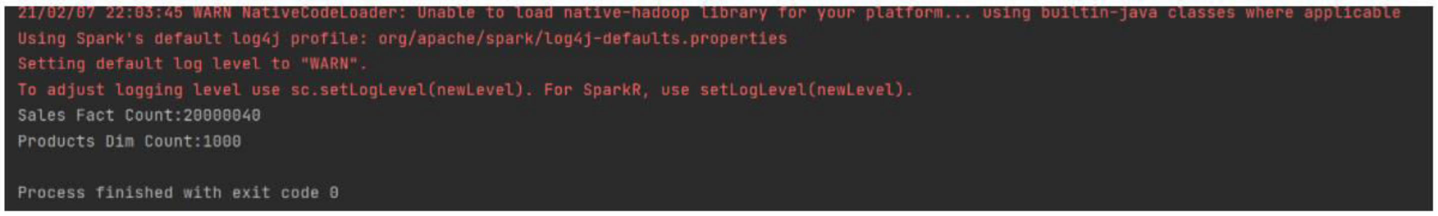
Comparison:
We have observed, **Sort-Merge** join requires **full shuffling** of **both** datasets via **network** which is **heavy** task for Spark. What if we can eliminate shuffling? How can we do it? **Replicate the whole small table to all executors**.



Sort-Merge vs Broadcast:
I have an example data set **Sales Fact** table and **Products Dimension** table. Sales Fact Table is very big in size and Products is quite simple.

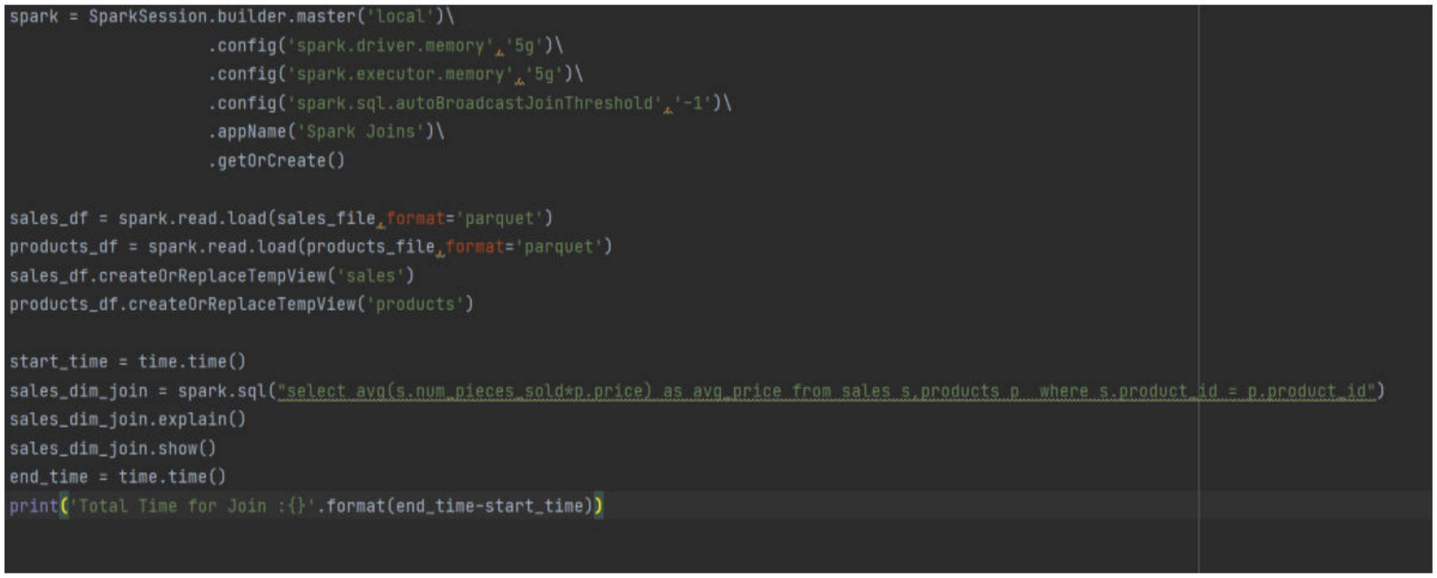
Verdict: Broadcast Join is 4 times faster if one of the table is small enough to fit in memory.

Please find below code snippets and results.

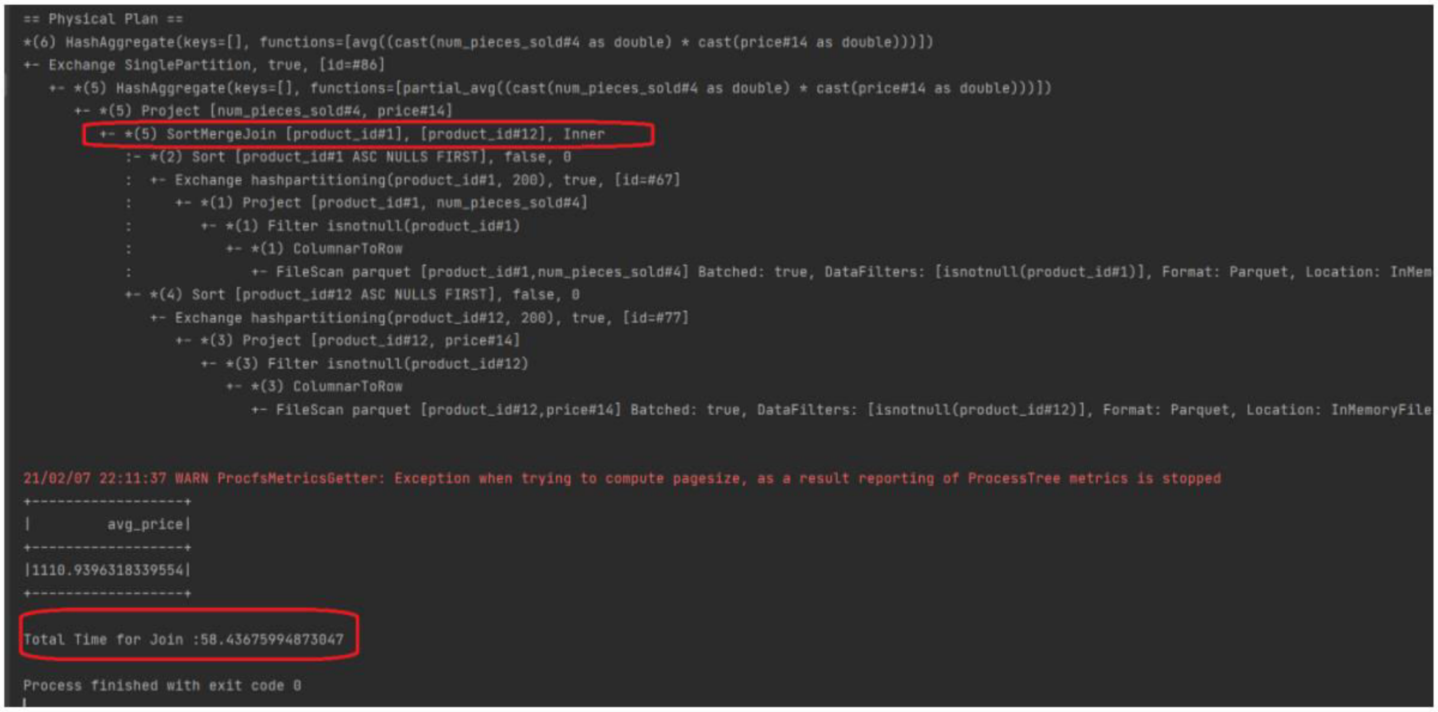


```
21/02/07 22:03:45 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Sales Fact Count:20000040
Products Dim Count:1000

Process finished with exit code 0
```

Sort-Merge Join (58 seconds):



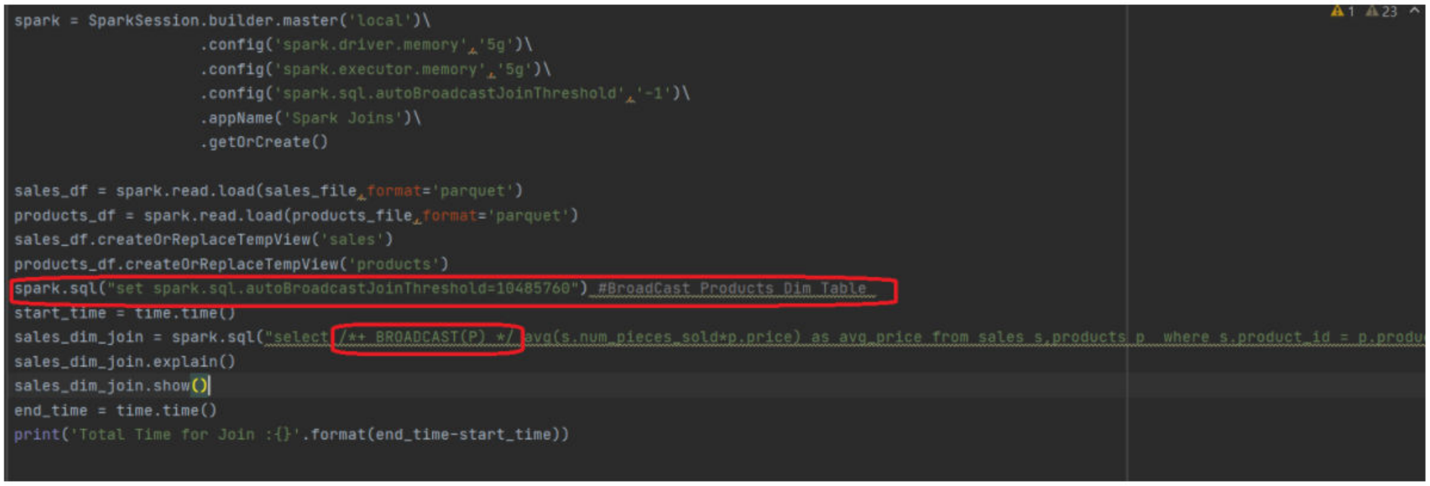
```
spark = SparkSession.builder.master('local')\
                    .config('spark.driver.memory','5g')\
                    .config('spark.executor.memory','5g')\
                    .config('spark.sql.autoBroadcastJoinThreshold','-1')\
                    .appName('Spark Joins')\
                    .getOrCreate()

sales_df = spark.read.load(sales_file,format='parquet')
products_df = spark.read.load(products_file,format='parquet')
sales_df.createOrReplaceTempView('sales')
products_df.createOrReplaceTempView('products')

start_time = time.time()
sales_dim_join = spark.sql("select avg(s.num_pieces_sold*p.price) as avg_price from sales s,products p  where s.product_id = p.product_id")
sales_dim_join.explain()
sales_dim_join.show()
end_time = time.time()
print('Total Time for Join :{}'.format(end_time-start_time))
```



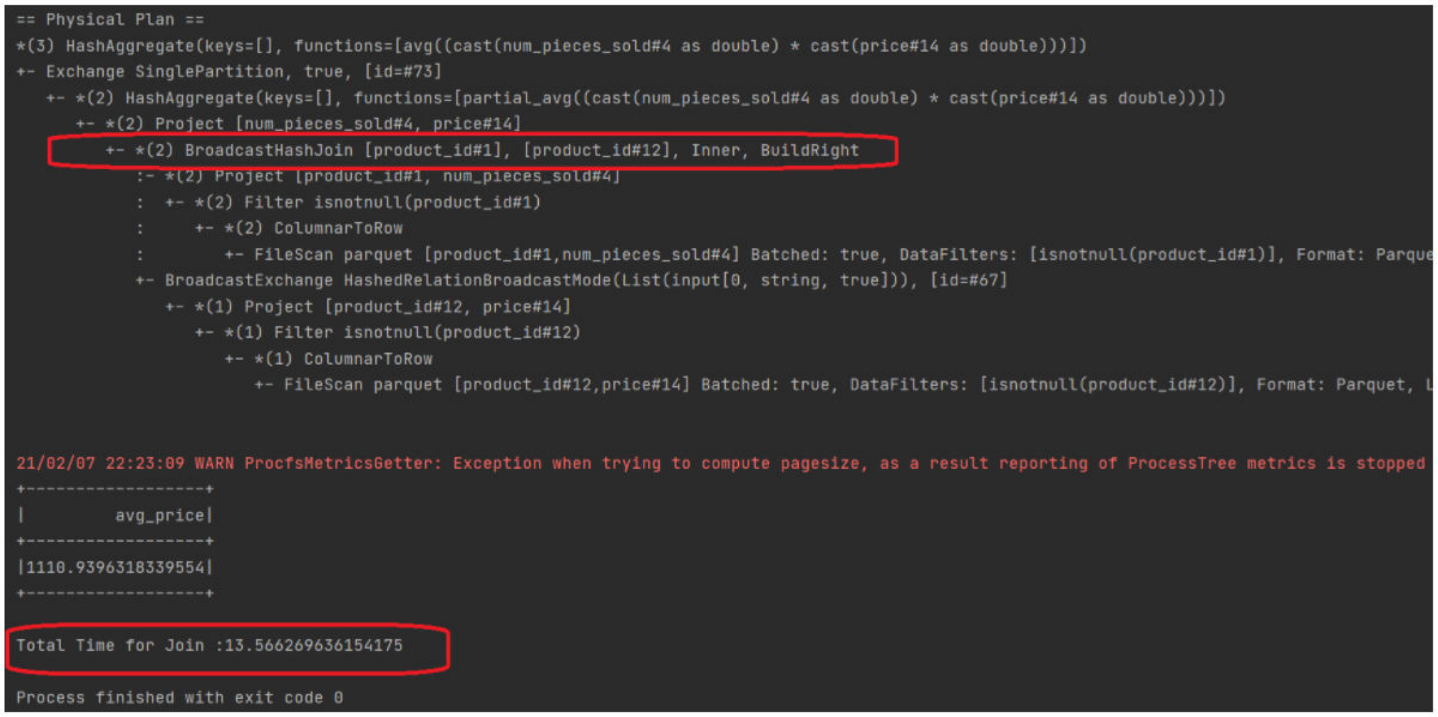
```
== Physical Plan ==
*(6) HashAggregate(keys=[], functions=[avg((cast(num_pieces_sold#4 as double) * cast(price#14 as double)))])
+- Exchange SinglePartition, true, [id=#86]
   +- *(5) HashAggregate(keys=[], functions=[partial_avg((cast(num_pieces_sold#4 as double) * cast(price#14 as double)))])
      +- *(5) Project [num_pieces_sold#4, price#14]
         +- *(5) SortMergeJoin [product_id#1], [product_id#12], Inner
            :- *(2) Sort [product_id#1 ASC NULLS FIRST], false, 0
            :  +- Exchange hashpartitioning(product_id#1, 200), true, [id=#67]
            :     +- *(1) Project [product_id#1, num_pieces_sold#4]
            :        +- *(1) Filter isnotnull(product_id#1)
            :           +- *(1) ColumnarToRow
            :              +- FileScan parquet [product_id#1,num_pieces_sold#4] Batched: true, DataFilters: [isnotnull(product_id#1)], Format: Parquet, Location: InMem
            +- *(4) Sort [product_id#12 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(product_id#12, 200), true, [id=#77]
                  +- *(3) Project [product_id#12, price#14]
                     +- *(3) Filter isnotnull(product_id#12)
                        +- *(3) ColumnarToRow
                           +- FileScan parquet [product_id#12,price#14] Batched: true, DataFilters: [isnotnull(product_id#12)], Format: Parquet, Location: InMemoryFile


21/02/07 22:11:37 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of ProcessTree metrics is stopped
+------------------+
|         avg_price|
+------------------+
|1110.9396318339554|
+------------------+

Total Time for Join :58.43675994873047

Process finished with exit code 0
```

Broad Cast Join (13 seconds):

- ➢ Enabled **spark.sql.autoBroadcastJoinThreshold** parameter to 10 MB (default) and added **hint** in SQL query **explicitly**. **Hint** is **not required**.
- ➢ **Spark Catalyst Optimizer** automatically does **broadcasting** a small table if it's less than **10mb** size. But I **intentionally** added **hint** to **demonstrate**.



```
spark = SparkSession.builder.master('local')\
                    .config('spark.driver.memory','5g')\
                    .config('spark.executor.memory','5g')\
                    .config('spark.sql.autoBroadcastJoinThreshold','-1')\
                    .appName('Spark Joins')\
                    .getOrCreate()

sales_df = spark.read.load(sales_file,format='parquet')
products_df = spark.read.load(products_file,format='parquet')
sales_df.createOrReplaceTempView('sales')
products_df.createOrReplaceTempView('products')
spark.sql("set spark.sql.autoBroadcastJoinThreshold=10485760") #BroadCast Products Dim Table
start_time = time.time()
sales_dim_join = spark.sql("select /*+ BROADCAST(P) */ avg(s.num_pieces_sold*p.price) as avg_price from sales s,products p  where s.product_id = p.produ
sales_dim_join.explain()
sales_dim_join.show()
end_time = time.time()
print('Total Time for Join :{}'.format(end_time-start_time))
```



```
== Physical Plan ==
*(3) HashAggregate(keys=[], functions=[avg((cast(num_pieces_sold#4 as double) * cast(price#14 as double)))])
+- Exchange SinglePartition, true, [id=#73]
   +- *(2) HashAggregate(keys=[], functions=[partial_avg((cast(num_pieces_sold#4 as double) * cast(price#14 as double)))])
      +- *(2) Project [num_pieces_sold#4, price#14]
         +- *(2) BroadcastHashJoin [product_id#1], [product_id#12], Inner, BuildRight
            :- *(2) Project [product_id#1, num_pieces_sold#4]
            :  +- *(2) Filter isnotnull(product_id#1)
            :     +- *(2) ColumnarToRow
            :        +- FileScan parquet [product_id#1,num_pieces_sold#4] Batched: true, DataFilters: [isnotnull(product_id#1)], Format: Parque
            +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true])), [id=#67]
               +- *(1) Project [product_id#12, price#14]
                  +- *(1) Filter isnotnull(product_id#12)
                     +- *(1) ColumnarToRow
                        +- FileScan parquet [product_id#12,price#14] Batched: true, DataFilters: [isnotnull(product_id#12)], Format: Parquet, L


21/02/07 22:23:09 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of ProcessTree metrics is stopped
+------------------+
|         avg_price|
+------------------+
|1110.9396318339554|
+------------------+

Total Time for Join :13.566269636154175

Process finished with exit code 0
```

Is broadcasting always a good solution?
Absolutely **NO**. If you are **joining** two data sets both are **very large** Broadcasting any table would **kill** your **spark cluster** and **fails** your job.

Why?

- Under the hood the **driver node** should **start replicating broadcasted table** into **one** of the **executors**.
- Once it's **finished** the **executor writes** it to **another executor** and it **continues** until **all** the **executors gets** the **copy** of **broadcast table**.
- As you already got an idea, this is **hitting** the **process** and **memory** of **all nodes** in **cluster** including **driver node**.
- Be cautious while doing broadcast join. Thumb rule **one** small table to fit into **memory**.

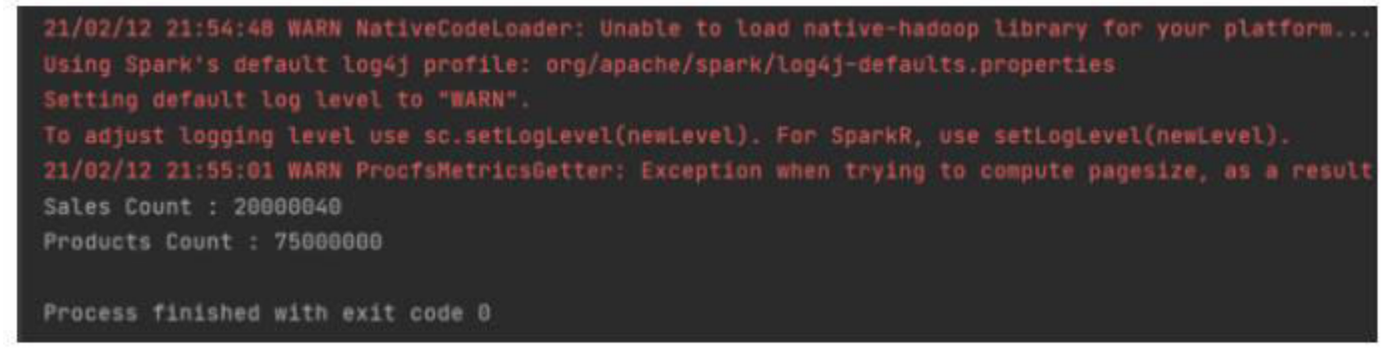
Part 2: Shuffling

- While doing any multi-row operations like joins, grouping and aggregating all nodes in the spark cluster should exchange data so that each node should get a piece of data.
- Shuffling is very costly operation which requires all nodes should exchange data via network. But there is no other go when performing aggregates by grouping.
- By default Spark sets shuffle partitions as 200. This must (**Must) be changed when dealing with large data sets.

Let's take an example:

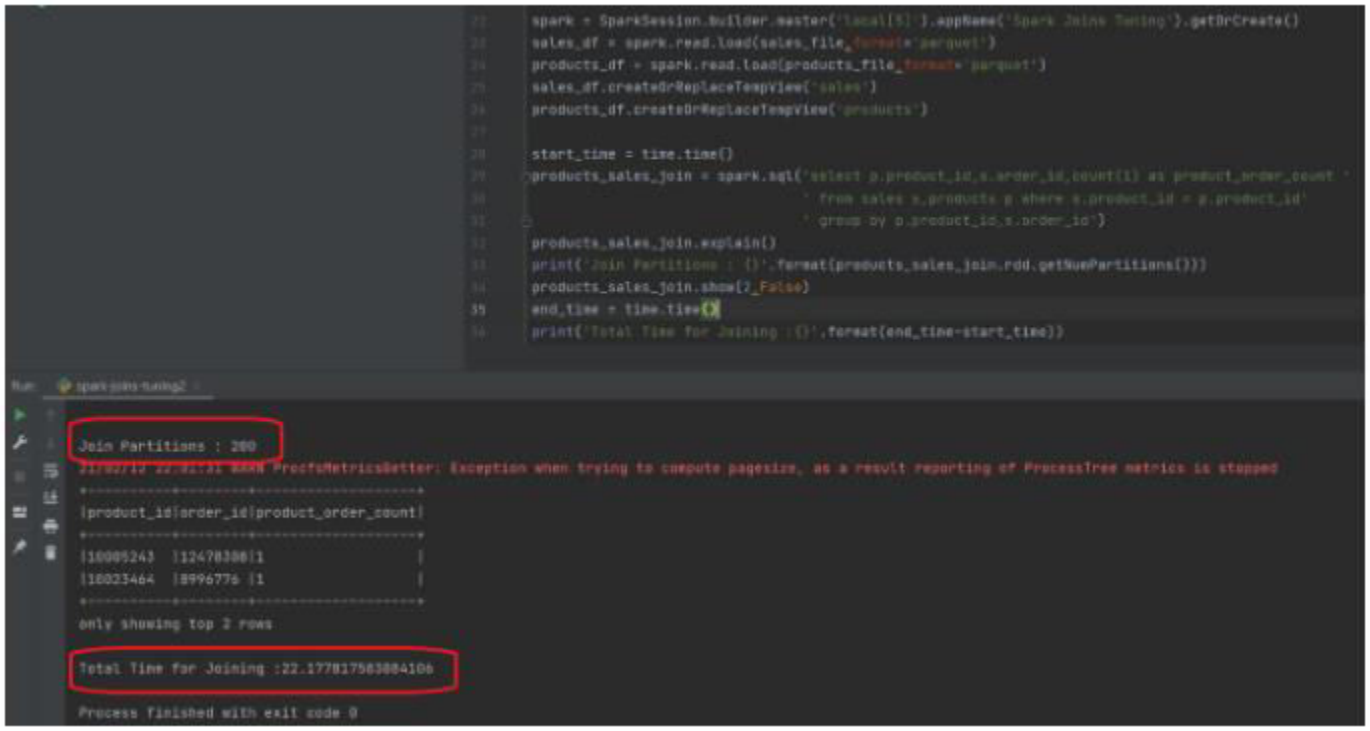
Sales Data: 20M **Products Data:** 75M **Allocated Executors:** 5[Local]

Verdict: Changing default shuffle partitions (200) either by increasing or coalescing partitions based on spark configuration and data size would significantly improve join performance.



```
21/02/12 21:54:48 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform...
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
21/02/12 21:55:01 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result
Sales Count : 20000040
Products Count : 75000000

Process finished with exit code 0
```

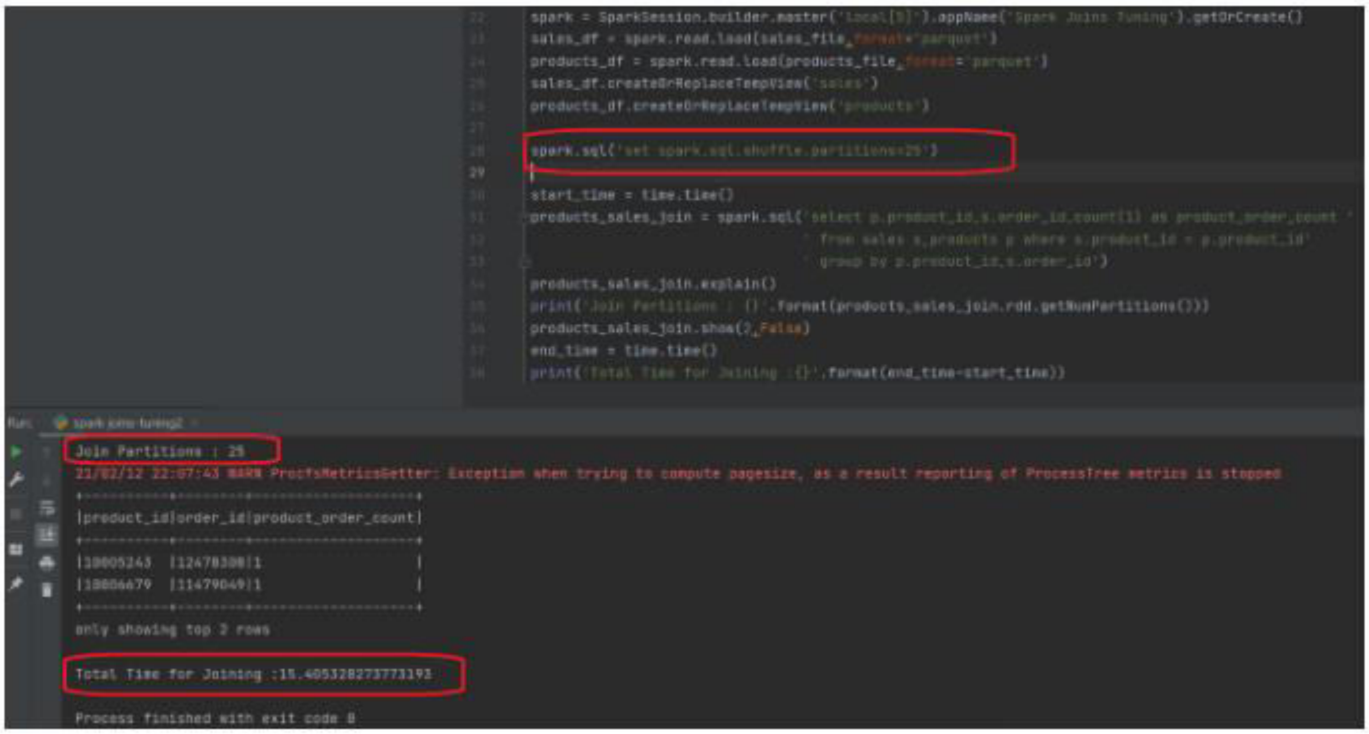
Join with Default Shuffle Partitions (200): 22 Seconds



```
spark = SparkSession.builder.master('local[5]').appName('Spark Joins Tuning').getOrCreate()
sales_df = spark.read.load(sales_file,format='parquet')
products_df = spark.read.load(products_file,format='parquet')
sales_df.createOrReplaceTempView('sales')
products_df.createOrReplaceTempView('products')

start_time = time.time()
products_sales_join = spark.sql('select p.product_id,s.order_id,count(1) as product_order_count '
                                ' from sales s,products p where s.product_id = p.product_id'
                                ' group by p.product_id,s.order_id')
products_sales_join.explain()
print('Join Partitions : {}'.format(products_sales_join.rdd.getNumPartitions()))
products_sales_join.show(2,False)
end_time = time.time()
print('Total Time for Joining : {}'.format(end_time-start_time))
```

```
Join Partitions : 200
21/02/12 22:01:31 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of ProcessTree metrics is stopped
+----------+---------+-------------------+
|product_id|order_id|product_order_count|
+----------+---------+-------------------+
|10005243  |12478388|1                  |
|10023464  |8996776 |1                  |
+----------+---------+-------------------+
only showing top 2 rows

Total Time for Joining :22.177817563884106

Process finished with exit code 0
```

Change Default Shuffle partitions: 15 seconds



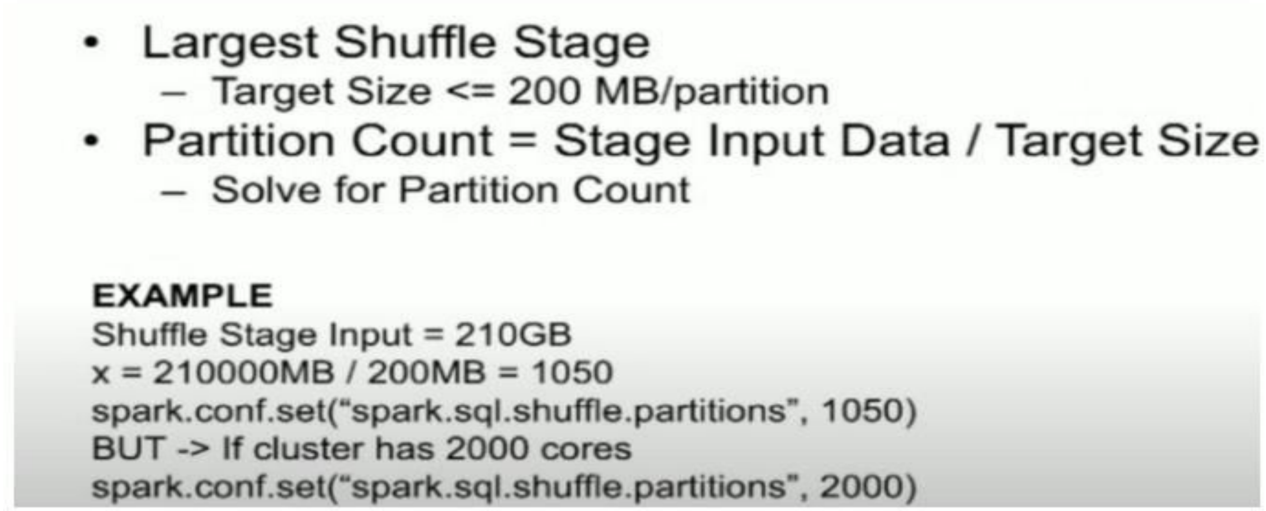
```
spark = SparkSession.builder.master('local[5]').appName('Spark Joins Tuning').getOrCreate()
sales_df = spark.read.load(sales_file,format='parquet')
products_df = spark.read.load(products_file,format='parquet')
sales_df.createOrReplaceTempView('sales')
products_df.createOrReplaceTempView('products')

spark.sql('set spark.sql.shuffle.partitions=25')

start_time = time.time()
products_sales_join = spark.sql('select p.product_id,s.order_id,count(1) as product_order_count '
                                ' from sales s,products p where s.product_id = p.product_id'
                                ' group by p.product_id,s.order_id')
products_sales_join.explain()
print('Join Partitions : {}'.format(products_sales_join.rdd.getNumPartitions()))
products_sales_join.show(2,False)
end_time = time.time()
print('Total Time for Joining :{}'.format(end_time-start_time))
```

```
Join Partitions : 25
21/02/12 22:07:43 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of ProcessTree metrics is stopped
+----------+---------+-------------------+
|product_id|order_id|product_order_count|
+----------+---------+-------------------+
|18005243  |12478308|1                  |
|18006679  |11479049|1                  |
+----------+---------+-------------------+
only showing top 2 rows

Total Time for Joining :15.405328273773193

Process finished with exit code 0
```

As you can see, having decent compact **25 shuffle partitions** rather than **small 200 partitions** made a difference in **join** performance. You can see **significant** improvement on big Spark cluster with **correct shuffle partitions**.

How many shuffle partitions?
This is **tricky question** and also **complex**. Having said that, it depends on Spark **physicals resources**, **configuration** and **size** of the **data** that you are dealing. It's not that **easy** to find the **correct number**. You just have to **try** for **different values** and **evaluate** to **get** any **conclusion**.

The guidelines given by Databricks:



- Largest Shuffle Stage
  - Target Size <= 200 MB/partition
- Partition Count = Stage Input Data / Target Size
  - Solve for Partition Count

EXAMPLE
Shuffle Stage Input = 210GB
x = 210000MB / 200MB = 1050
spark.conf.set("spark.sql.shuffle.partitions", 1050)
BUT -> If cluster has 2000 cores
spark.conf.set("spark.sql.shuffle.partitions", 2000)

Accumulators:

- **Accumulators** are **updateable** variables that are added through an **associative** & **commutative** operation on **all nodes**.
- They are used to implement **counters** or **sums**.
- Spark natively supports **numeric accumulators**.

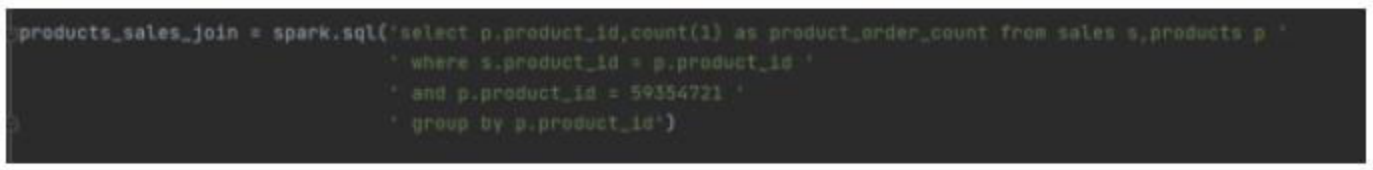
Spark AQE (Adaptive Query Execution):
Adaptive Query Execution (AQE) is an **optimization** technique in **Spark SQL** that makes use of the **runtime statistics** to choose the **most efficient query execution plan**.

In **Spark 3.0**, the **AQE framework** is **shipped** with **three features**:

- Dynamically coalescing shuffle partitions
- Dynamically switching join strategies
- Dynamically optimizing skew joins

With the Current Dataset I have, I am going to show **dynamically switching join strategies.**

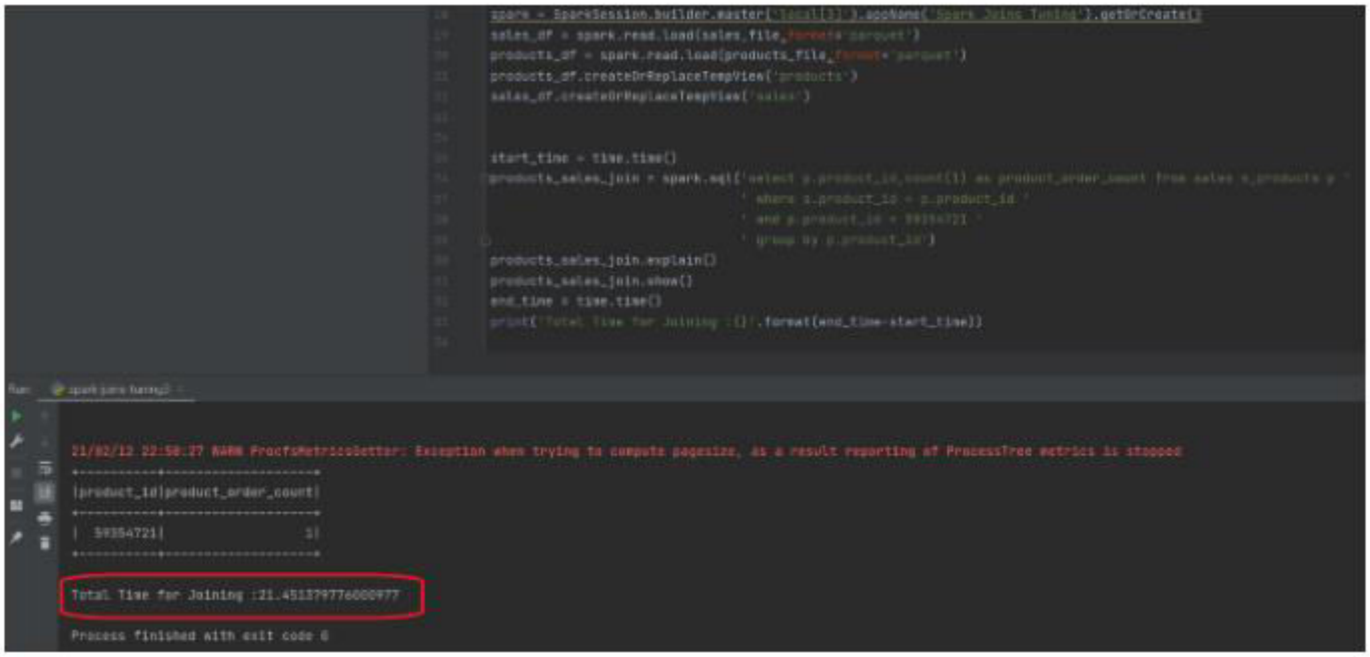
Dynamically Switching Join:
AQE converts **sort-merge** join to **broadcast** join when the **runtime statistics** of any **join** side is **smaller** than the **broadcast join threshold.** This is not as efficient as planning a broadcast hash join in the first place, but it's better than keep doing the sort-merge join, as we can save the sorting of both the join sides.



```
products_sales_join = spark.sql('select p.product_id,count(1) as product_order_count from sales s,products p '
                                ' where s.product_id = p.product_id '
                                ' and p.product_id = 59354721 '
                                ' group by p.product_id')
```

```
-- Get total sales count grouped by sales order for product with a
product id 59354721.
-- The selectivity of the filter by product id is not known in static
planning, so the initial plan opts for sort merge join.
-- But in fact, the "products" table after filtering is very
small(just 1 row), so the query can do a broadcast hash join instead.

-- Static explain shows the initial plan with sort merge join.
-- Without AQE enabled , Spark does Sort Merge Join
-- AQE enabled, Spark does Broadcast join during run time.
```

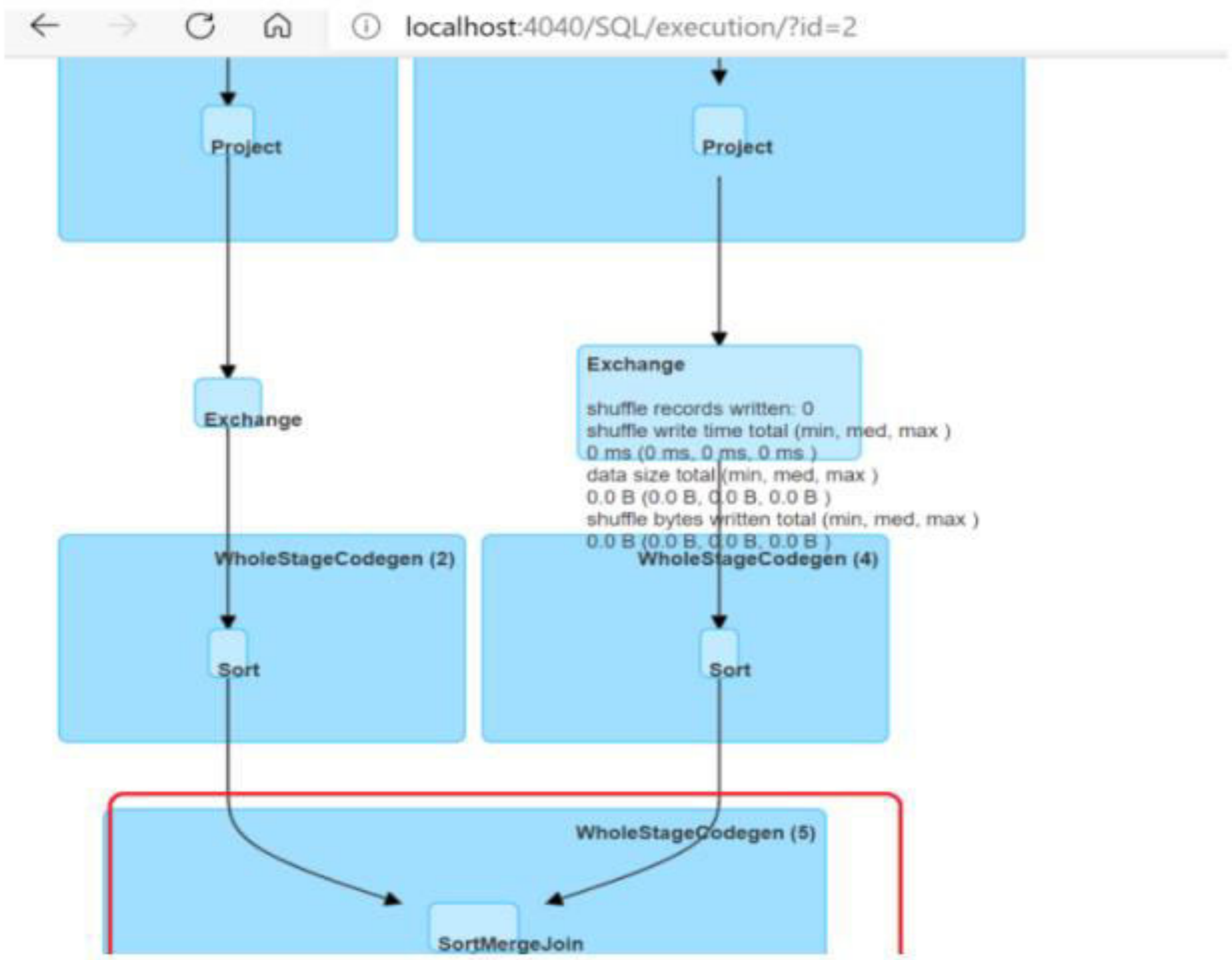
Without AQE: 21 seconds and Join type is Sort Merge



```
spark = SparkSession.builder.master('local[3]').appName('Spark Joins Testing').getOrCreate()
sales_df = spark.read.load(sales_file, format='parquet')
products_df = spark.read.load(products_file, format='parquet')
products_df.createOrReplaceTempView('products')
sales_df.createOrReplaceTempView('sales')


start_time = time.time()
products_sales_join = spark.sql('select p.product_id,count(1) as product_order_count from sales s,products p '
                                ' where s.product_id = p.product_id '
                                ' and p.product_id = 59354721 '
                                ' group by p.product_id')
products_sales_join.explain()
products_sales_join.show()
end_time = time.time()
print('Total Time for Joining :{}'.format(end_time-start_time))
```

```
21/02/12 22:58:27 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of ProcessTree metrics is stopped
+----------+-------------------+
|product_id|product_order_count|
+----------+-------------------+
|  59354721|                  1|
+----------+-------------------+

Total Time for Joining :21.451379776000977

Process finished with exit code 0
```

Without AQE

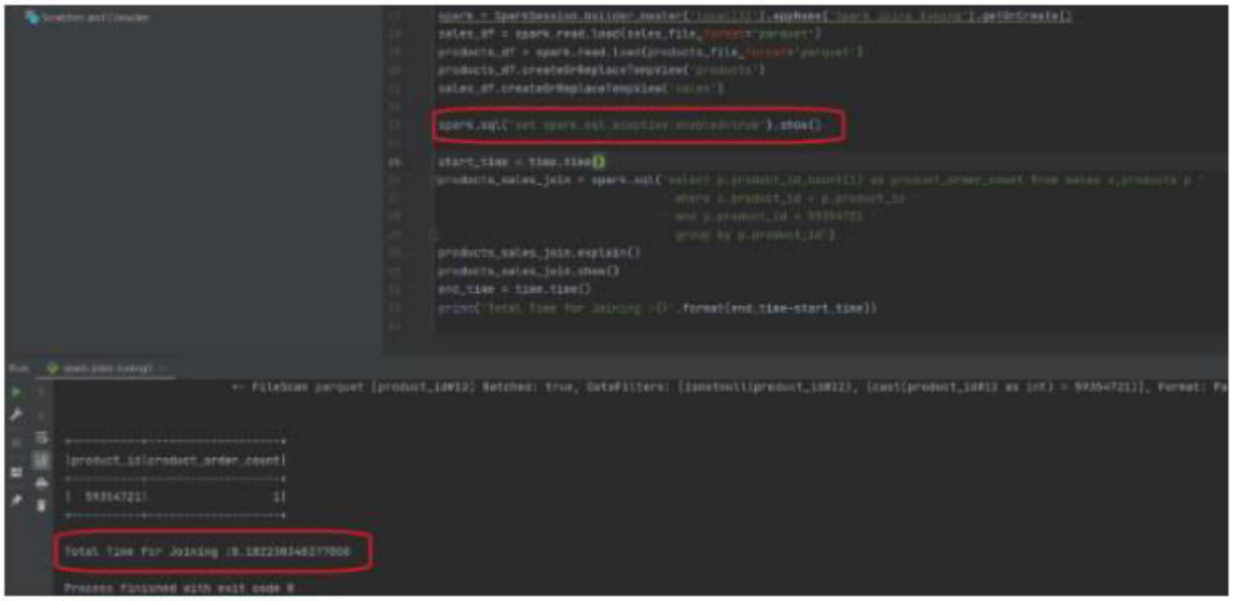


localhost:4040/SQL/execution/?id=2

Project
Project
Exchange
Exchange
shuffle records written: 0
shuffle write time total (min, med, max)
0 ms (0 ms, 0 ms, 0 ms)
data size total (min, med, max)
0.0 B (0.0 B, 0.0 B, 0.0 B)
shuffle bytes written total (min, med, max)
0.0 B (0.0 B, 0.0 B, 0.0 B)
WholeStageCodegen (2)
WholeStageCodegen (4)
Sort
Sort
WholeStageCodegen (5)
SortMergeJoin

Without AQE Join Type

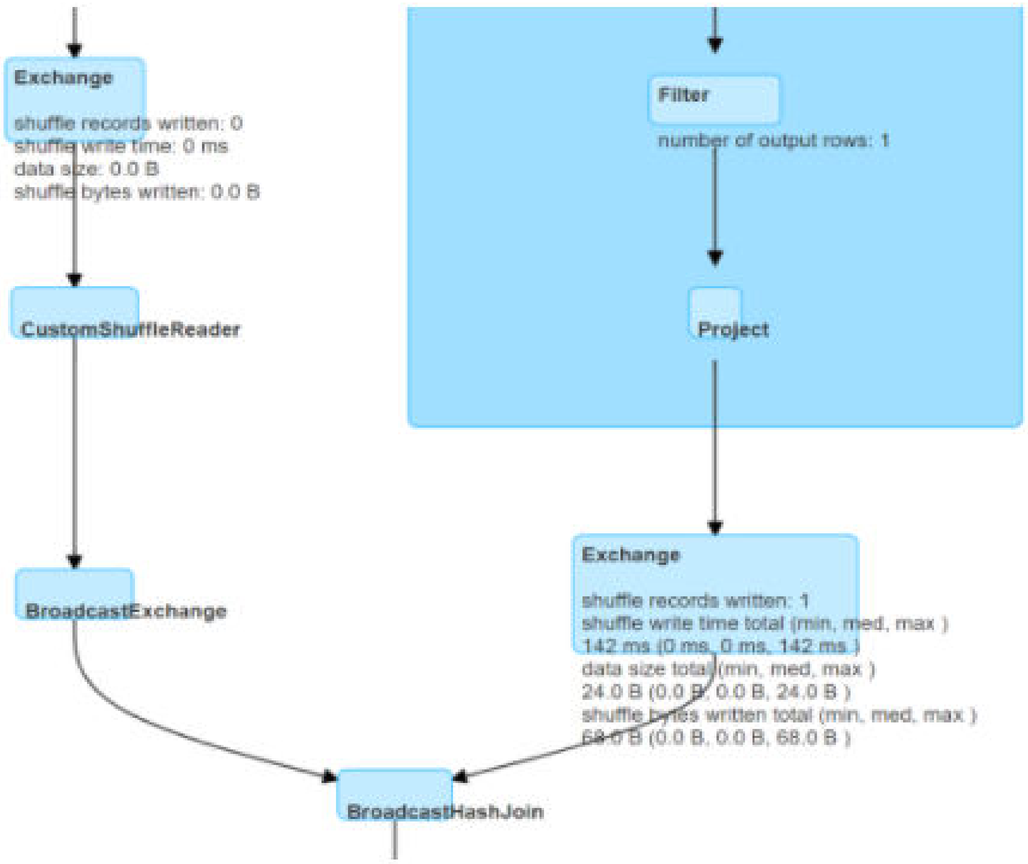
AQE Enabled: 8 seconds and Join Type Broadcast



```
spark = SparkSession.builder.master('local[3]').appName('spark joins tuning').getOrCreate()
sales_df = spark.read.load(sales_file, format='parquet')
products_df = spark.read.load(products_file, format='parquet')
products_df.createOrReplaceTempView('products')
sales_df.createOrReplaceTempView('sales')

spark.sql('set spark.sql.adaptive.enabled=true').show()

start_time = time.time()
products_sales_join = spark.sql('select p.product_id,count(1) as product_order_count from sales s,products p '
                                'where s.product_id = p.product_id '
                                'and p.product_id = 59354721 '
                                'group by p.product_id')

products_sales_join.explain()
products_sales_join.show()
end_time = time.time()
print('Total time for Joining :{}'.format(end_time-start_time))
```

```
      +- FileScan parquet [product_id#12] Batched: true, DataFilters: [isnotnull(product_id#12), (cast(product_id#12 as int) = 59354721)], Format: Pa

+----------+-------------------+
|product_id|product_order_count|
+----------+-------------------+
|  59354721|                  1|
+----------+-------------------+

Total time for Joining :8.182230346277008

Process finished with exit code 0
```

AQE Enabled Join


Exchange
shuffle records written: 0
shuffle write time: 0 ms
data size: 0.0 B
shuffle bytes written: 0.0 B
CustomShuffleReader
BroadcastExchange
Filter
number of output rows: 1
Project
Exchange
shuffle records written: 1
shuffle write time total (min, med, max)
142 ms (0 ms, 0 ms, 142 ms)
data size total (min, med, max)
24.0 B (0.0 B, 0.0 B, 24.0 B)
shuffle bytes written total (min, med, max)
68.0 B (0.0 B, 0.0 B, 68.0 B)
BroadcastHashJoin

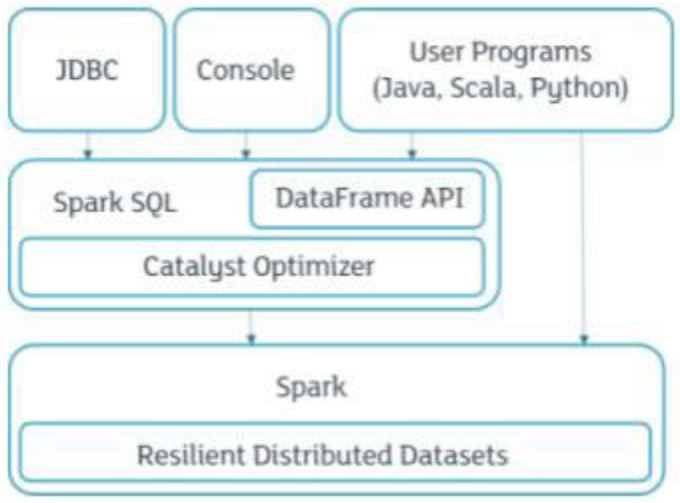
Dynamically changed join type to Broadcast

Catalyst Optimization:

- Spark SQL is an Apache Spark module for structured data processing.
- One of the big differences with the Spark API RDD is that its interfaces provide additional information to perform more efficient processes.
- This information is also useful for Spark SQL to benefit internally from using its Catalyst optimizer and improve performance in data processing.

What is Catalyst?

- Spark SQL was designed with an optimizer called Catalyst based on the functional programming of Scala.
- Its two main purposes are: first, to add new optimization techniques to solve some problems with "big data" and second, to allow developers to expand and customize the functions of the optimizer.


JDBC
Console
User Programs (Java, Scala, Python)
Spark SQL
DataFrame API
Catalyst Optimizer
Spark
Resilient Distributed Datasets

Catalyst Spark SQL architecture and Catalyst optimizer integration

Using Catalyst in Spark SQL:

The Catalyst Optimizer in Spark offers rule-based and cost-based optimization. Rule-based optimization indicates how to execute the query from a set of defined rules. Meanwhile, cost-based optimization generates multiple execution plans and compares them to choose the lowest cost one.

Phases: The four phases of the transformation that Catalyst performs are as follows:

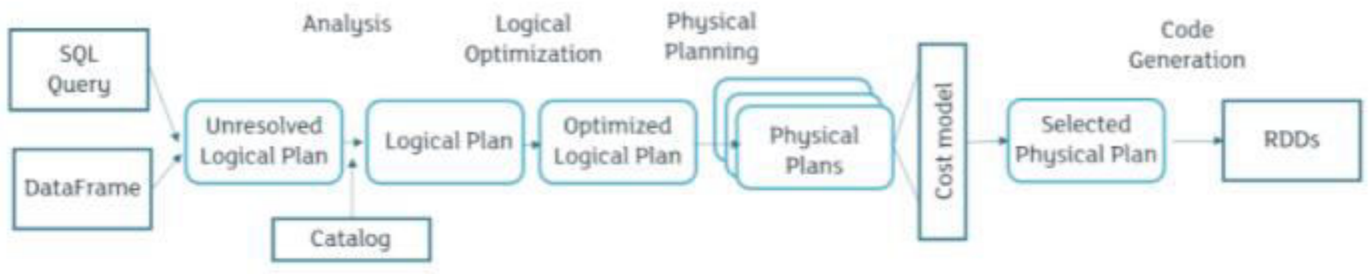
1) Analysis:

The first phase of Spark SQL optimization is the analysis. Spark SQL starts with a relationship to be processed that can be in two ways. A serious form from an AST (abstract syntax tree) returned by an SQL parser, and on the other hand from a DataFrame object of the Spark SQL API.

2) Logic Optimization Plan:
The second phase is the logical optimization plan. In this phase, rule-based optimization is applied to the logical plan. It is possible to easily add new rules.

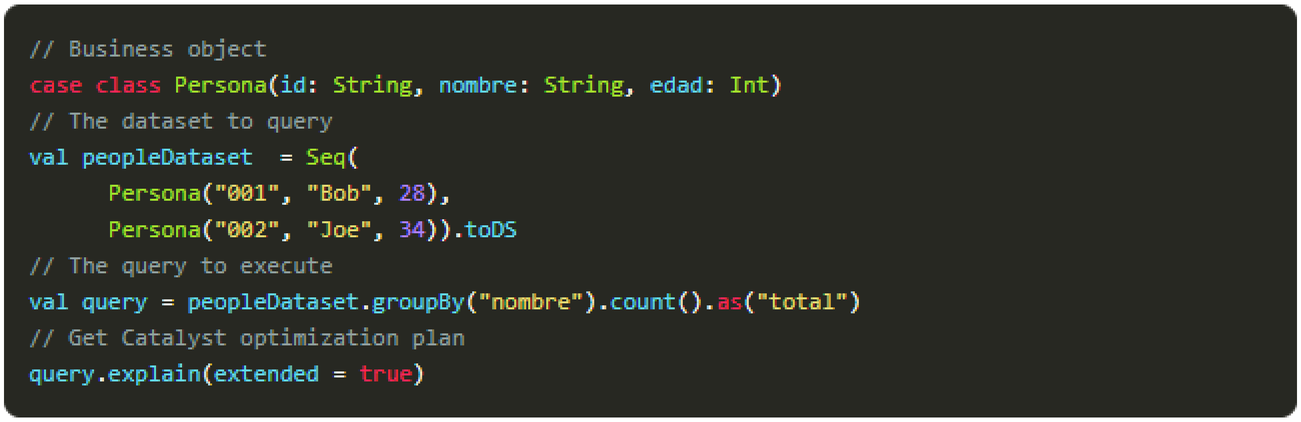
3) Physical plan:
In the physical plan phase, Spark SQL takes the logical plan and generates one or more physical plans using the physical operators that match the Spark execution engine. The plan to be executed is selected using the cost-based model (comparison between model costs).

4) Code generation:
Code generation is the final phase of optimizing Spark SQL. To run on each machine, it is necessary to generate Java code byte code.

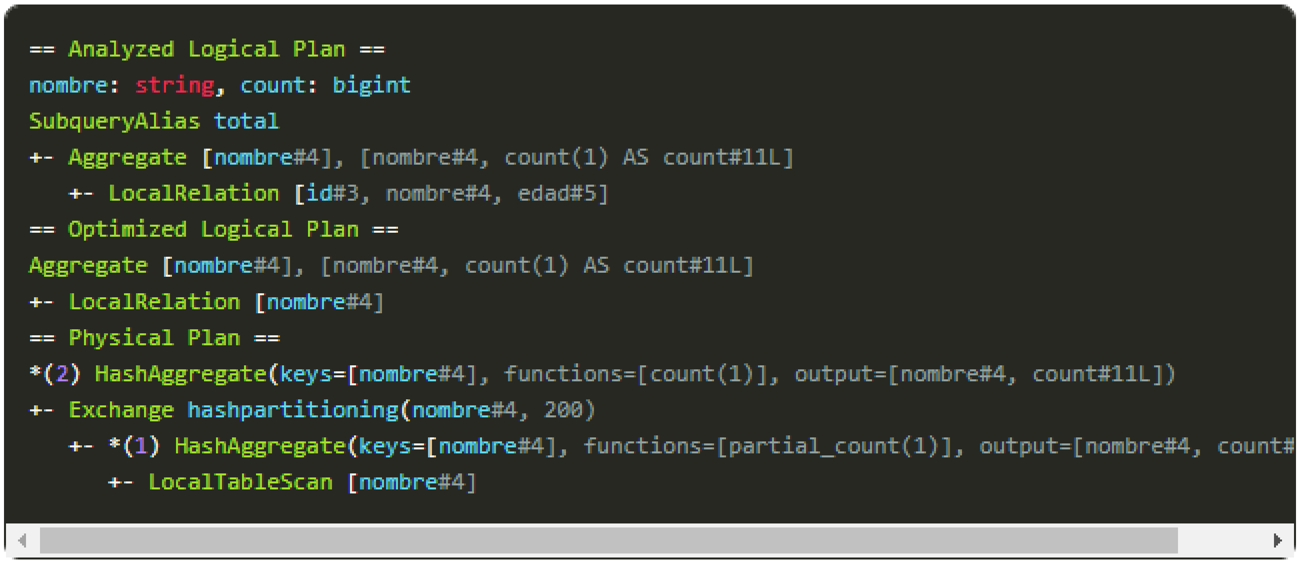


Phases of the query plan in Spark SQL. Rounded squares represent the Catalyst trees

Example:
The Catalyst optimizer is enabled by default as of Spark 2.0, and contains optimizations to manipulate datasets.


```
// Business object
case class Persona(id: String, nombre: String, edad: Int)
// The dataset to query
val peopleDataset  = Seq(
      Persona("001", "Bob", 28),
      Persona("002", "Joe", 34)).toDS
// The query to execute
val query = peopleDataset.groupBy("nombre").count().as("total")
// Get Catalyst optimization plan
query.explain(extended = true)
```

As a result, the detailed plan for the consultation is obtained:



```
== Analyzed Logical Plan ==
nombre: string, count: bigint
SubqueryAlias total
+- Aggregate [nombre#4], [nombre#4, count(1) AS count#11L]
   +- LocalRelation [id#3, nombre#4, edad#5]
== Optimized Logical Plan ==
Aggregate [nombre#4], [nombre#4, count(1) AS count#11L]
+- LocalRelation [nombre#4]
== Physical Plan ==
*(2) HashAggregate(keys=[nombre#4], functions=[count(1)], output=[nombre#4, count#11L])
+- Exchange hashpartitioning(nombre#4, 200)
   +- *(1) HashAggregate(keys=[nombre#4], functions=[partial_count(1)], output=[nombre#4, count#
      +- LocalTableScan [nombre#4]
```

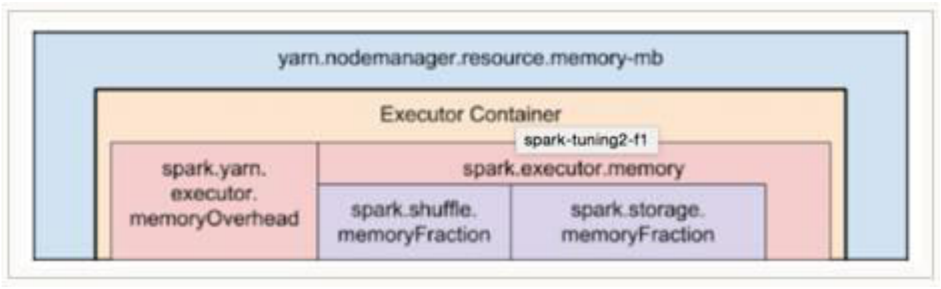
Memory Calculation:
Following list captures some recommendations to keep in mind while configuring them:

1) Hadoop/Yarn/OS Deamons: When we run spark application using a cluster manager like Yarn, there'll be several daemons that'll run in the background like NameNode, Secondary NameNode, DataNode, Resource Manager and Node Manager. So, while specifying num-executors, we need to make sure that we leave aside enough cores (~1 core per node) for these daemons to run smoothly.

2) Yarn ApplicationMaster (AM): ApplicationMaster is responsible for negotiating resources from the ResourceManager and working with the NodeManagers to execute and monitor the containers and their resource consumption. If we are running Spark on Yarn, then we need to budget in the resources that AM would need (~1024MB and 1 Executor).

3) HDFS Throughput: HDFS client has trouble with tons of concurrent threads. It was observed that HDFS achieves full write throughput with ~5 tasks per executor. So it's good to keep the number of cores per executor below that number.

4) MemoryOverhead: Following picture depicts spark-yarn-memory-usage.


yarn.nodemanager.resource.memory-mb
Executor Container
spark-tuning2-f1
spark.yarn.executor.memoryOverhead
spark.executor.memory
spark.shuffle.memoryFraction
spark.storage.memoryFraction

Two things to make note of from this picture:
Full memory requested to yarn per executor = spark-executor-memory + spark.yarn.executor.memoryOverhead.
spark.yarn.executor.memoryOverhead = Max (384MB, 7% of spark.executor - memory)

So, if we request 20GB per executor, AM will actually get 20GB + memoryOverhead = 20 + 7% of 20GB = ~23GB memory for us.

Running executors with too much memory often results in excessive garbage collection delays.

spark-submit
--executor-cores **???**
--num-executors **???**
--executor-memory **???**

1) Hadoop/Yarn/OS Daemons:
==> 1 core per node
2) YARN:
==> 1024MB or 1 Executor
3) HDFS Throughput:
==> 5 Tasks per executor
4) MemoryOverhead:
Max (384MB, 7% of executor)

Practical Understanding:
Now, let's consider a 10 node cluster with following config and analyses different possibilities of executors-core-memory distribution:

Cluster Config:
10 Nodes / 16 cores per Node / 64GB RAM per Node

Let's assign 5 core per executors =>
--executor-cores = 5 (for good HDFS throughput)
Leave 1 core per node for Hadoop/Yarn daemons =>
Num cores available per node = 16-1 = 15
So, Total available of cores in cluster = 15 x 10 = 150

Number of available executors = (total cores/num-cores-per-executor) = 150/5 = 30
Leaving 1 executor for ApplicationManager => --num-executors = 29

Number of executors per node = 30/10 = 3
Memory per executor = 64GB/3 = 21GB

Counting off heap overhead = 7% of 21GB = 1.5GB.
So, actual --executor-memory = 21 – 1.5 = 19.5GB
So, recommended config is: 29 executors, 19.5GB memory each and 5 cores each!!

Analysis: It is obvious as to how this third approach has found right balance between Fat vs Tiny approaches. Needless to say, it achieved parallelism of a fat executor and best throughputs of a tiny executor!!

--num-executors, --executor-cores and --executor-memory. These three param play a very important role in spark performance as they control the amount of CPU & memory your spark application gets. This makes it very crucial for users to understand the right way to configure them.

Types of Partitioning in Spark:
- Hash Partitioning
- Range Partitioning

1) Hash Partitioning:
- Hash Partitioning attempts to spread the data evenly across various partitions based on the key.
- Object.hashCode method is used to determine the partition in Spark as partition = key.hashCode ( ) % numPartitions.

2) Range Partitioning:
- Some Spark RDDs have keys that follow a particular ordering for such RDDs range partitioning is an efficient partitioning technique.
- In range partitioning method, tuples having keys within the same range will appear on the same machine.
- Keys in a range partitioner are partitioned based on the set of sorted range of keys and ordering of keys.

Note:
- Spark's range partitioning and hash partitioning techniques are ideal for various spark use cases but spark does allow users to fine tune how their RDD is partitioned, by using custom partitioner objects.
- Custom Spark partitioning is available only for pair RDDs i.e. RDDs with key value pairs as the elements can be grouped based on a function of each key.
- Spark does not provide explicit control of which key will go to which worker node but it ensures that a set of keys will appear together on some node.

Data Serialization:

Data serialization is important in distributed environments. Spark provides two serialization libraries – **Java** and **Kyro**.

- By default spark uses java serialization.
- Kyro is significantly faster and more compact than Java serialization.
- The reason Kyro is not the default is because of the custom registration requirement, it does not support all Serializable types and requires registering the classes.

spark.conf.set ("spark.serializer", "org.apache.spark.serializer.KryoSerializer").

Shuffle:

Spark.sql.shuffle.partition: Shuffle partitions are the partitions in spark dataframe, which is created using a grouped or join operation. Number of partitions in this dataframe is different than the original dataframe partitions. Default value is 200.

The challenge is the number of shuffle partitions in spark is static. It doesn't change with different data size. It can be set dynamically by using conf method on the sparkSession

Partition count = Stage input data/Target size
Target size let's say 200 MB
Stage Input size = 10 GB
Partitions = 10000/200 = 50

Note:
If the data volume is not enough to fill all the partitions when there are 200 of them, it would lead to creation of very small files in HDFS, which is not desirable.
However if there are too little partitions, and lots of data to process, each of the executor's memory might not be enough/available to process so much of data at a given time, causing errors like this java.lang.OutOfMemoryError: Java heap space.

Output:

a) Coalesce: Coalesce method reduces the number of partitions in a DataFrame. Coalesce avoids full shuffle, instead of creating new partitions, it shuffles the data using Hash Partitioner (Default), and adjusts into existing partitions, this means it can only decrease the number of partitions.
Coalesce can be done just before write to decide on number of files. It can be also used to downsize no of files.
b) Repartition: The repartition method can be used to either increase or decrease the number of partitions in a DataFrame. Repartition is a full Shuffle operation, whole data is taken out from existing partitions and equally distributed into newly formed partitions. Repartition is very expensive and needs another stage, hence should be avoided.

Data Skewness in Spark:
Data Skewness is uneven distribution of data.

First identify how many partitions are getting created after shuffle operation.
print ("Num Partitions: ", df.rdd.getNumPartitions())

You can configure number of shuffle partitions by setting property:
spark.sql.shuffle.partitions=<Number>

Check count of records per partition:
from **pyspark.sql.funtions** import **spark_partition_id**
df.withColumn ("partitionId",
spark_partition_id()).groupBy("partitionId").count().show(500)

It will print partitionId & count of records in that partition. Analyze the partition. You can use repartition ( ) function to shuffle data based on another column having high cardinality.

Hive Optimization: Top 5 Hive Optimization Techniques:

1) Table Design & File Format:

a) Partitioning:
Works by dividing the data into smaller logical segments.
We finally scan only one partition.
Partitioning can be done on columns with low cardinality.

b) Bucketing:
Works by dividing the data into smaller logical segments.
These segments are created based on system defined hash functions.
Bucketing can be done on columns with high cardinality.

c) Right File Format with Hive:
ORC can reduce data storage by 75% of the original data size. It uses technique like predicate push-down, compression to improve the performance of query.
Snappy provides a fast compression.

2) Optimization of Query:

a) Join Optimization:
Map Side Join, Bucket Map Join, Sort Merge Bucket Join (SMB) these all does minimum shuffling.

```
set hive.enforce.bucketing=true
set hive.enforce.sorting=true
set hive.auto.convert.join=true
set hive.auto.convert.sortmerger.join=true
set hive.optimize.bucektmapjoin=true
set hive.optimize.bucketmapjoin.sortedmerge=true
```

Map Join: Small table is stored in hash table and hash table is written to distributed cache. Join is performed by each local map task. No reduce task is used.

Bucket Map Join: Same as map join. Both tables in this join should be bucketed on same keys.

Sort Merge Bucket Join: If both tables are large then SMB join can be used but need to meet below conditions:
Both tables must be bucketed on same column.
Both tables must be sorted on same column.
No. of bucket in one table should be in multiple of another.

b) Window Functions:
Use of windowing functions for performing complicated queries.

3) Query Execution Engine:
Try using TEZ or Spark enable Hive Engine for better query execution.

1) Vectorization: Limitation: This can be used only with ORC File format.

```
set hive.vectorized.execution=True
set hive.vectorized.execution.enabled=True
```

2) Try avoiding UDF's as they are not much optimized.

3) Cost Based Optimization:

```
set hive.cbo.enable=True
set hive.compute.query.using.stats=True
set hive.stats.fetch.column.stats=True
set hive.stats.fetch.partition.stats=True
```