**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](http://h/)

( 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.

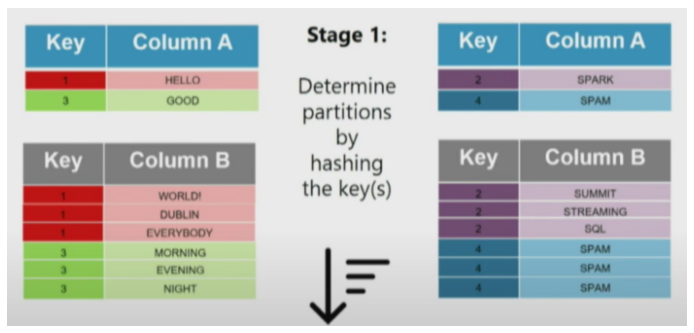
* **Merge** is very quick thing.

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.

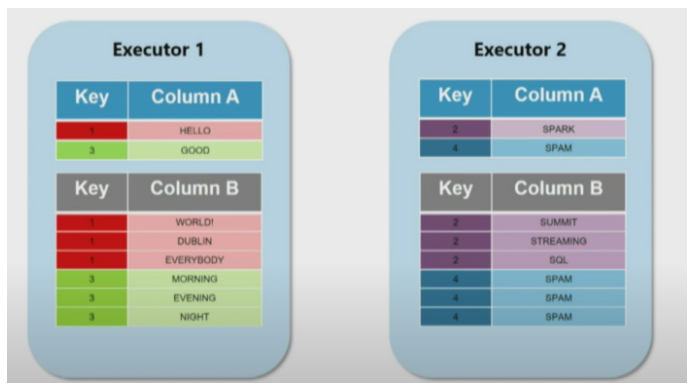


**Sort Phase:**

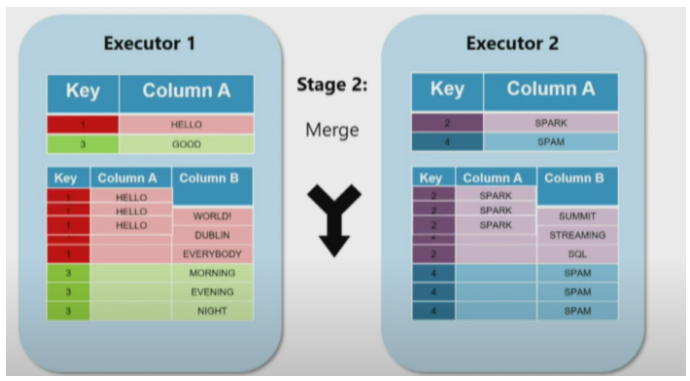
* As you can see, both A and B are **sorted** by **Join** key i.e. **Key** column and **sorted** **data** is split into 2 partitions.
* Each **partition** should have **specific** **key** data.
* There should **not** be any **overlapping** of **Keys** between **partitions** which is **whole** idea of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



**Assign Sorted Partitions to Executors:**



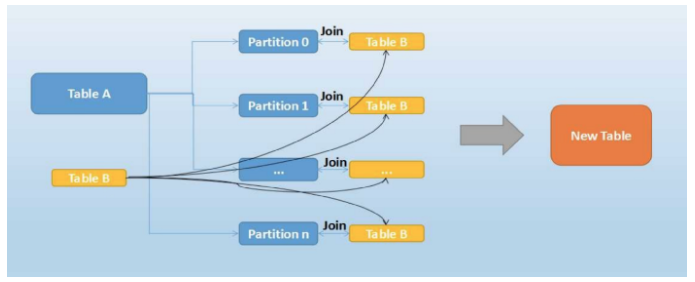
**Merge Phase:** Merging **sorted** data by **keys** is very simple and quickest operation.



**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**.

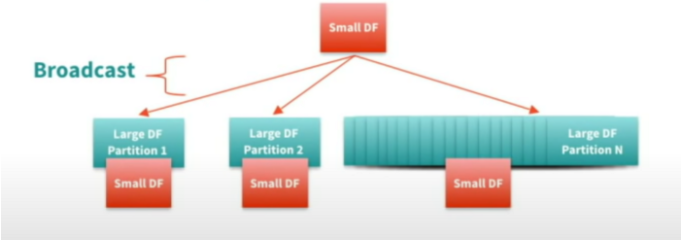


Now **Table B** is **present** on **all** the **nodes** where we have **data** for **Table A**, **no** **more** **data** **shuffling** is **required** and each **partition** of **Table A** can **join** with the required entries of **Table B**.

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**.

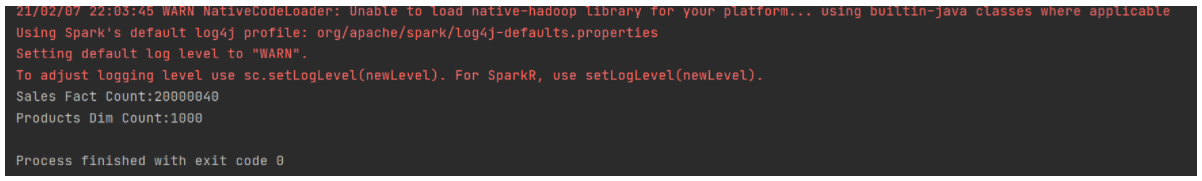


**Sort-Merge vs Broadcast:**

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.

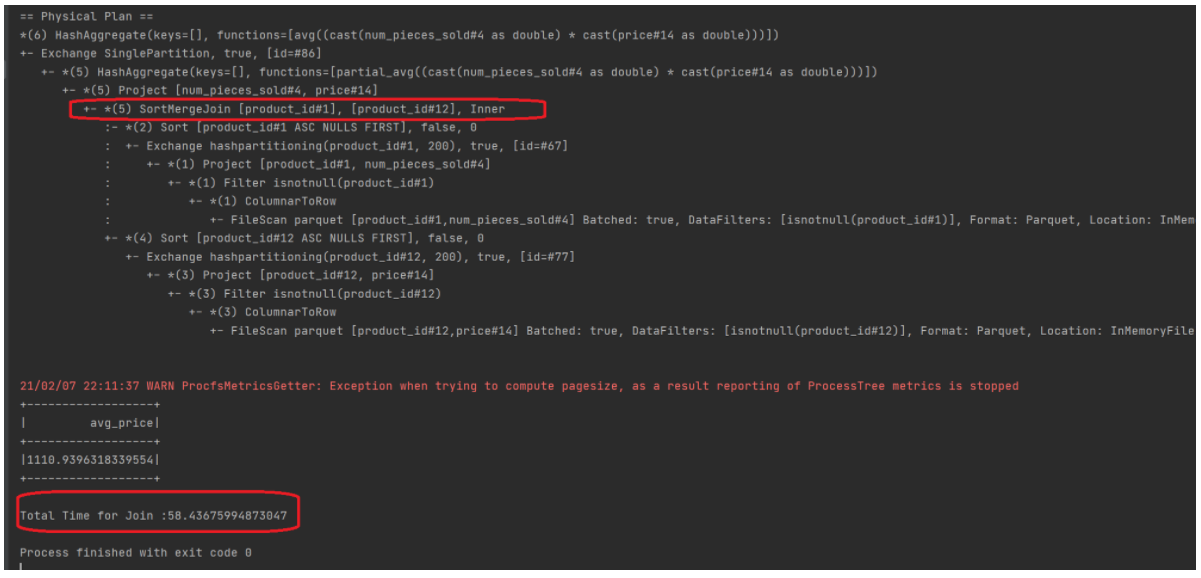
**Verdict**: Broadcast Join is 4 times faster if one of the table is small enough to fit in memory.

Please find below code snippets and results.



**Sort-Merge Join (58 seconds):**

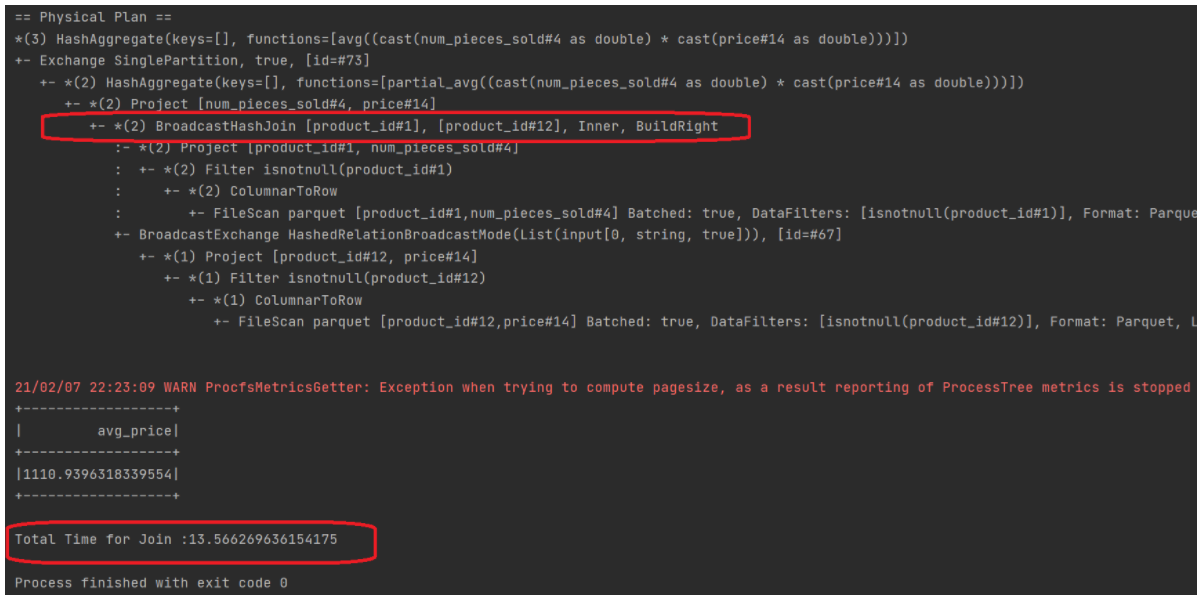




**Broad Cast Join (13 seconds):**

* 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**.





**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**.

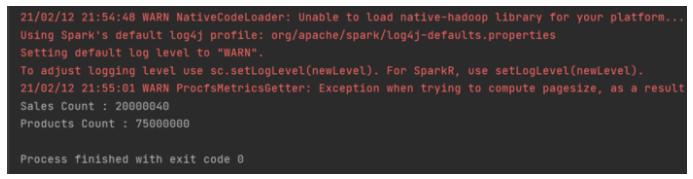
**Part 2: Shuffling**

* 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.

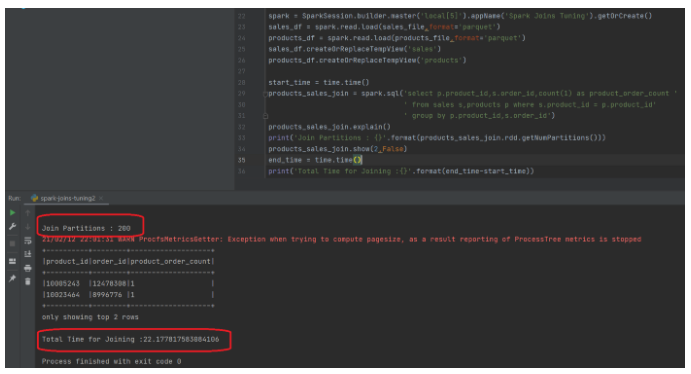
Let’s take an example:

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

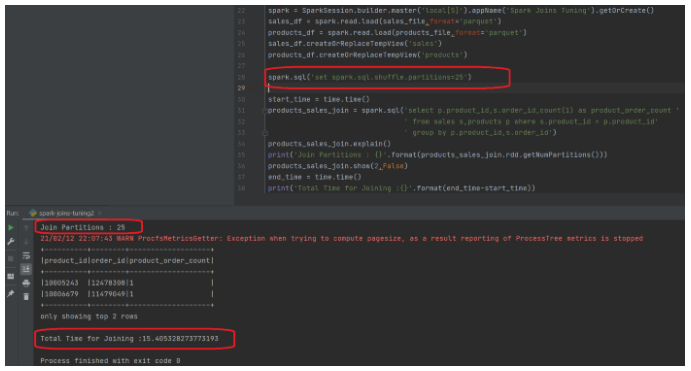
**Verdict:** Changing default shuffle partitions (200) either by increasing or coalescing partitions based on spark configuration and data size would significantly improve join performance.



**Join with Default Shuffle Partitions (200):** 22 Seconds



**Change Default Shuffle partitions:**15 seconds

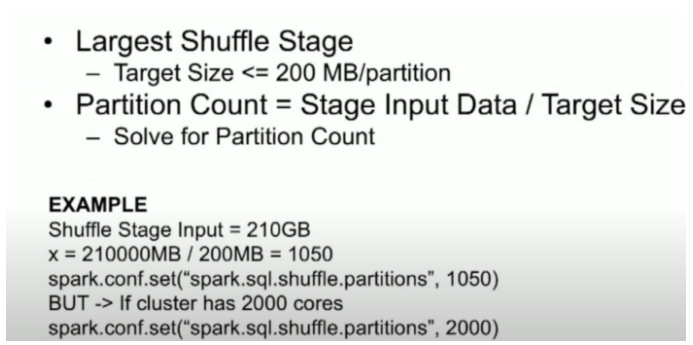


As you can see, having decent compact **25 shuffle partitions** rather than **small** **200 partitions** made a difference in **join** performance. You can see **significant** improvement on big Spark cluster with **correct** **shuffle** **partitions**.

**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:**



**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**.

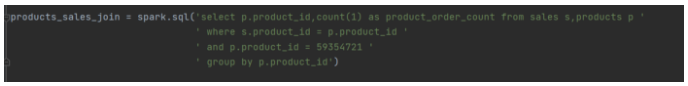
In **Spark 3.0**, the **AQE** **framework** is **shipped** with **three** **features**:

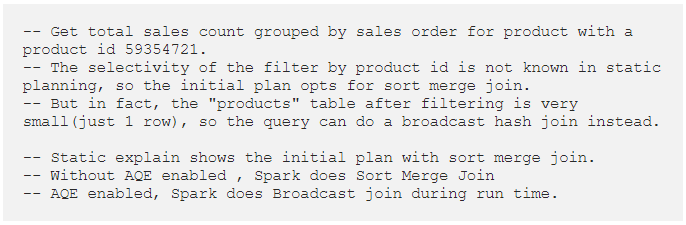
* Dynamically coalescing shuffle partitions
* Dynamically switching join strategies
* Dynamically optimizing skew joins

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

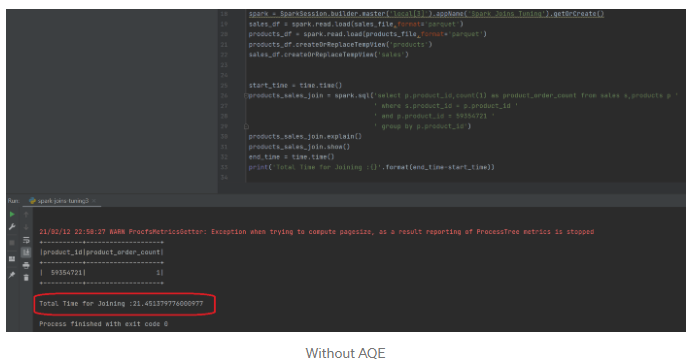
**Dynamically Switching Join:**

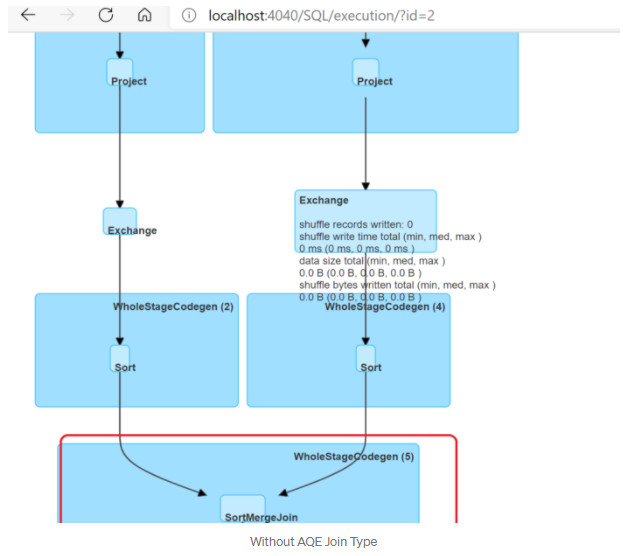
**AQE** converts **sort-merge** join to **broadcast** join when the **runtime** **statistics** of any **join** side is **smaller** than the **broadcast** **join** **threshold**. This is not as efficient as planning a broadcast hash join in the first place, but it’s better than keep doing the sort-merge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



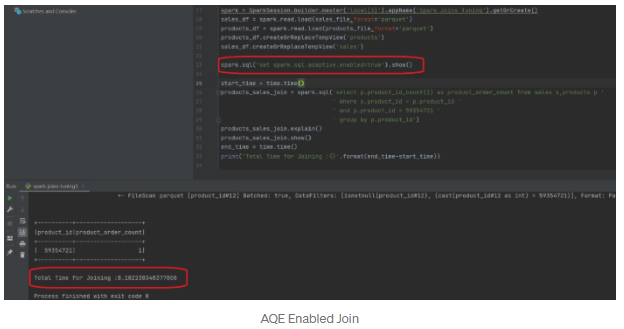


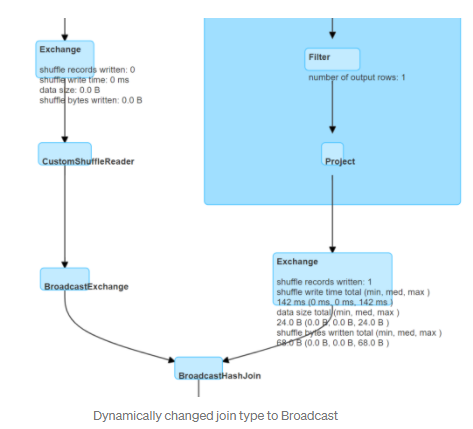
**Without AQE: 21 seconds and Join type is Sort Merge**





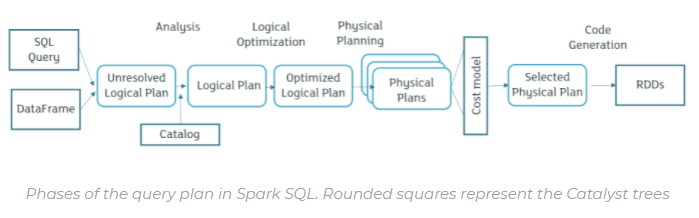
**AQE Enabled: 8 seconds and Join Type Broadcast**





**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.



**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.

**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.