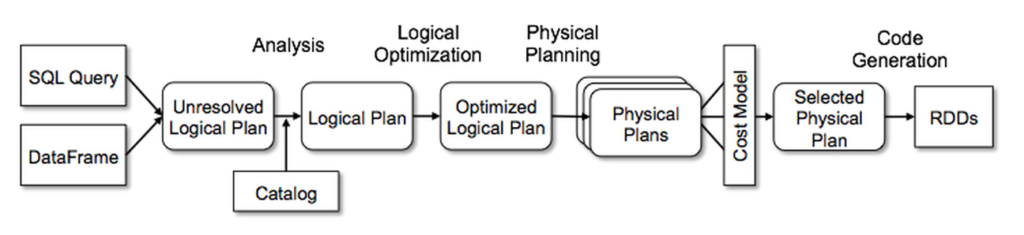
Adaptive Query Execution (AQE) is one of the greatest features of Spark 3.0 which reoptimizes and adjusts query plans based on runtime statistics collected during the execution of the query.

Adaptive Query Execution (AQE) is one of the greatest features of Spark 3.0 which reoptimizes and adjusts query plans based on runtime statistics collected during the execution of the query.

Prior to 3.0, Spark does the single-pass optimization by creating an execution plan (set of rules) before the query starts executing, once execution starts it sticks with the plan and starts executing the rules it created in the plan and doesn’t do any further optimization which is based on the metrics it collects during each stage.

**How Query Executes Prior to Spark 3.0**

Prior to 3.0, Spark does the single-pass optimization by creating an execution plan (set of rules) before the query starts executing, once execution starts it sticks with the plan and starts executing the rules it created in the plan and doesn’t do any further optimization which is based on the metrics it collects during each stage.



1. First, Spark parses the query and creates the Unresolved Logical Plan

Validates the syntax of the query.

Doesn’t validate the semantics meaning column name existence, data types.

1. Analysis: Using the Catalyst, it converts the Unresolved Logical Plan to Resolved Logical Plan a.k.a Logical Plan.

The catalog contains the column names and data types, during this step, it validates the columns mentioned in a query with catalog.

1. Optimization: Converts Logical Plan into Optimized Logical Plan.
2. Planner: Now it creates One or More Physical Plans from an optimized Logical plan.
3. Cost Model: In this phase, calculates the cost for each Physical plan and select the Best Physical Plan.
4. RDD Generation: RDD’s are generated, this is the final phase of query optimization which generates RDD in Java bytecode.

Once RDD’s are generated in Byte code, the Spark execution engine executes the transformations and action.

Adaptive Query Optimization in Spark 3.0, reoptimizes and adjusts query plans based on runtime metrics collected during the execution of the query, this re-optimization of the execution plan happens after each stage of the query as stage gives the right place to do re-optimization.

With each major release of Spark, it’s been introducing a new [optimization](https://sparkbyexamples.com/spark/spark-performance-tuning/) features in order to better execute the query to achieve the greater performance.

Spark 1.x – Introduced Catalyst Optimizer and Tungsten Execution Engine

Spark 2.x – Added Cost-Based Optimizer

Spark 3.0 – Now added Adaptive Query Execution

Adaptive Query Execution is disabled by default. In order to enable set spark.sql.adaptive.enabled configuration property to true. Besides this property, you also need to enable the AQE feature you going to use that are explained later in the section.

**spark.conf.set("spark.sql.adaptive.enabled",true)**

Spark 3.0 comes with three major features in AQE.

1. Reducing Post-shuffle Partitions.
2. Switching Join Strategies to broadcast Join
3. Optimizing Skew Join

Prior to 3.0, the developer needs to know the data as Spark doesn’t provide the optimal partitions after each shuffle operation and the developer needs to [re-partition to increase or coalesce to decrease the partitions](https://sparkbyexamples.com/spark/spark-repartition-vs-coalesce/) based on the total number of records.

With Spark 3.0, after every stage of the job, Spark dynamically determines the optimal number of partitions by looking at the metrics of the completed stage. In order to use this, you need to enable the below configuration.

**spark.conf.set("spark.sql.adaptive.enabled",true)**

**spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled",true)**

Use case :

Without using AQE :

If we have a df and we are using group by to get the count on that, Since groupBy() triggers the wider transformation or [shuffle](https://sparkbyexamples.com/spark/spark-shuffle-partitions/), statement df1.rdd.getNumPartitions results in 200 partitions; This is because spark by default creates **200 partitions** for shuffle operations.

**import spark.implicits.\_**

**val simpleData = Seq(("James","Sales","NY",90000,34,10000),**

**("Michael","Sales","NY",86000,56,20000),**

**("Robert","Sales","CA",81000,30,23000),**

**("Maria","Finance","CA",90000,24,23000),**

**("Raman","Finance","CA",99000,40,24000),**

**("Scott","Finance","NY",83000,36,19000),**

**("Jen","Finance","NY",79000,53,15000),**

**("Jeff","Marketing","CA",80000,25,18000),**

**("Kumar","Marketing","NY",91000,50,21000)**

**)**

**val df = simpleData.toDF("employee\_name","department","state","salary","age","bonus")**

**val df1=df.groupBy("department").count()**

**println(df1.rdd.getNumPartitions)**

we will get 200 partitions here (default number).

But if we enable AQE, we will get less number of partition may be 7-8.

With this feature, developers don’t have to know the size of the data and do the re-partition post shuffle operations base on the data. Spark takes care of this hereafter.

Sometimes we may come across data in partitions that are not evenly distributed, this is called Data Skew. Operations such as join perform very slow on these partitions. By enabling the AQE, Spark checks the stage statistics and determines if there are any Skew joins and optimizes it by splitting the bigger partitions into smaller (matching partition size on other table/dataframe).

**spark.conf.set("spark.sql.adaptive.enabled",true)**

**spark.conf.set("spark.sql.adaptive.skewJoin.enabled",true)**