Optimizer

> -
>
> what is catalyst optimizer.
>
> scala program provided by spark for data frame / SQL API, that automatically finds out most efficient plan to execute data operations specified in the user code.

Scope op optimizer

> Few operation by which optimization can be done.
>
> - Predicate or projection pushdown - help to eliminate data from respecting precondition earlier in the computation.
>   - when we have joining condition, it will move the joining condition such that that data is reduced to smallest so less computation needed.
>   - projection pushdown - out of many column we only need few columns at end only so its better to drop those unwanted column thus only those which are needed at end are considered.
> - Rearrange filter - The optimizes will check which filter will remove most of the records and firstly that filter is executed. Thus with this we have dataset with less record thus now less computation is needed.
> - Replacement of some RegEx expressions by Java's functions start with(string) or contains (string)
>   - This is very costlier operation in spark. If there is RegEx expression than optimizer will look to equivalent function in java.
> - if-else statement - The optimizer should smart enough to choose the right path based on condition.

💡 Optimizer is available for data frame and data set not for RDD.

RDD vs Data frame

> similarity
>
> - Both have immutability, in memory, resilient, distributed computing.
>   - Immutability - The source dataset cannot be modified,
>   - Resilient - They have lazy evaluation model, execution only happen when action is called. This is done by DAG if something wring happen than using DAG we can again generate the dataset.
>
> Difference
>
> - Data frame differ from RDD in maintaining the structure of underlying data in form of schema, Thus data frame is equivalent to tables in database.
> - When data is structured, information about data can be collected and used to derive the best execution plan through RBO (role based optimizer) ,CBO(cost based optimizer) in database.

RBO vs CBO

> -
>
> RBO (Rule based optimizer)
>
> - set of predefined rules is used to design logical plan.
> - when we submit the query than optimizer will refer the predefined rule based on that it will create a logical plan. But it is not efficient for all cases because RBO blindly follow the predefined rule.
>
> CBO (cost based optimizer)
>
> - Based on available statistics collection of underlying data, cost is estimated and best efficient execution approach is decided on cost.

- statistic collection of underlying data
    - e.g. number of records, max min values, distinct values null values we have such info will be kept in statistics which will be used for optimization.

RBO - Rule based optimizer

-

The rule-based optimizer has 15 rules that it uses to determine how to parse a query. Each rule has a rank. The optimizer looks at all the combinations it can find, then chooses the access path with the lowest rank. the 15 rules in order.

Rule-Based Optimizer Rules of Precedence

Rank	Access Path
1	Single row by ROWID
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite key
9	Single-column indexes
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort-merge join
13	Maximum or minimum of indexed column
14	Order by an indexed column
15	Full table scan

CBO - cost based optimizer

-

Cost-Based Optimization (aka **Cost-Based Query Optimization** or **CBO Optimizer**) is an optimization technique in Spark SQL that uses table statistics to determine the most efficient query execution plan of a structured query (given the logical query plan).

Cost-based optimization is disabled by default. Spark SQL uses spark.sql.cbo.enabled configuration property to control whether the CBO should be enabled and used for query optimization or not.

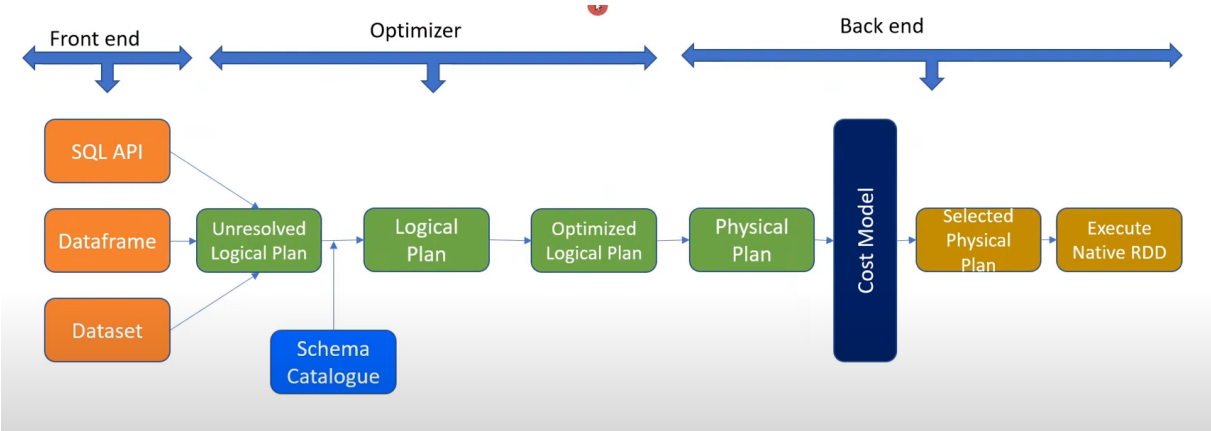
Table Statistics

The table statistics can be computed for tables, partitions and columns and are as follows:

1. **Total size** (in bytes) of a table or table partitions
2. **Row count** of a table or table partitions
3. Column statistics,
i.e. **min**, **max**, **num_nulls**, **distinct_count**, **avg_col_len**, **max_col_len**, **histogram**

catalyst optimizer

It uses both RBO and CBO techniques.



- unresolved logical plan -

    - When we submit the application to spark engine than it will first create unresolved logical plan which is high level plan for the submitted code. At this level syntax correctness is checked.
    - Schematic verification - e.g. we apply certain operation on the employee table but employee table must present before we apply query, or e.g. we are using some object in our query but that object is not created or defined in the environment than this will give error.
    - To validate the name and datatype info is fetched from schema catalogue which contain info about all table and schema.

- logical plan
    - In this stage
- The Catalyst analyzer then performs semantic analysis on the logical plan. It resolves references to columns, checks the validity of expressions, and ensures that the query adheres to the SQL or Data Frame/Dataset API specifications.
- logical plan
    - In Apache Spark's Catalyst optimizer, the logical plan represents the abstract syntax tree (AST) of the query after it has undergone semantic analysis and resolution of references. The logical plan is a high-level, tree-like representation of the user's query, capturing the intended operations without specifying how these operations will be physically executed.
    - Each tree contain nod and each node contain 0 or more child nodes and at each node there will be predefined rules (RBO) these rule can be customized and based on these rule logical plan is created,
- Optimized logical plan
    - Catalyst, the query optimization engine in Spark, applies various rule-based transformations to the logical plan to improve its efficiency without changing the semantics of the query.
    - Grouping relevant operation and create batches. or grouping the relevant operation into batches.

    - These optimizations might include predicate pushdown, constant folding, and other logical transformations aimed at improving the query's performance.
- Physical plan
    - The physical plan is the final execution plan that is generated based on the optimized logical plan. In this phase, Catalyst explores different physical execution strategies to execute the query efficiently on a distributed Spark cluster. This involves decisions about how data will be distributed, which join algorithms to use, and how to leverage parallelism. The physical plan is what is actually executed on the Spark cluster, and it includes information about tasks, stages, and dependencies between tasks.
    - At this point CBO comes into picture. In this cost will be assigned to each task.
    - For cost based optimization there is need of statistical data to optimize and if there is time difference in last collected statistical data and changes in happen in data frame than the optimization might not be so accurate. For this adaptive query execution is made.
- Cost model
    - after the number of physical plan is created, than based on cost the most efficient physical plan is selected.
    - application is converted to java code and than submitted to execution.

Adaptive query execution

Introduced in spark 3,0

```
spark.conf.set("saprk.sql.adaptive.enabled",True)
```

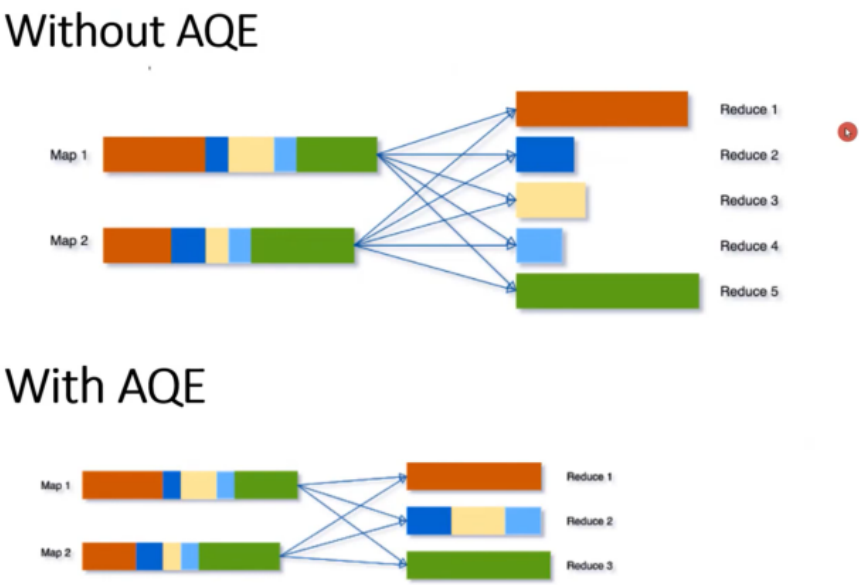
Adaptive Query Optimization in Spark 3.0, reoptimizes and adjusts query plans based on runtime metrics collected during the execution of the query, this re-

optimization of the execution plan happens after each stage of the query as stage gives the right place to do re-optimization.

Adaptive Query Execution Features

Spark 3.0 comes with three major features in AQE.

- Reducing Post-shuffle Partitions.
    - 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 based on the total number of records.
        - coalesce - to repartition the smaller dataset such.
        - without AQE for red and green it will take more time to compute as it has larger dataset and for other it will take less time and because of that some core will stay ideal even job is not done.
        - with AQE - thus to coalesce the smaller data set such that now all core do job approx. at same time so cores don't remain ideal and resource are utilized properly.



    - its always difficult to measure and set the number of partitions.
    - If we choose to have too few partitions, each partition size would be too big to process , and to process these large partitions may need to spill the data into disk thus hitting performance.
    - If we choose too many partition, then the data size of each partition may be very small. it would lead to network i/o operation for data fetches to read the shuffle blocks, which can slow down the query.
        - as with large number of partition with smaller size there is high chance of data shuffle among the node as required data might be in other node and as we know shuffle is a costlier process.
    - AQE chooses the best number of partitions automatically.

  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.

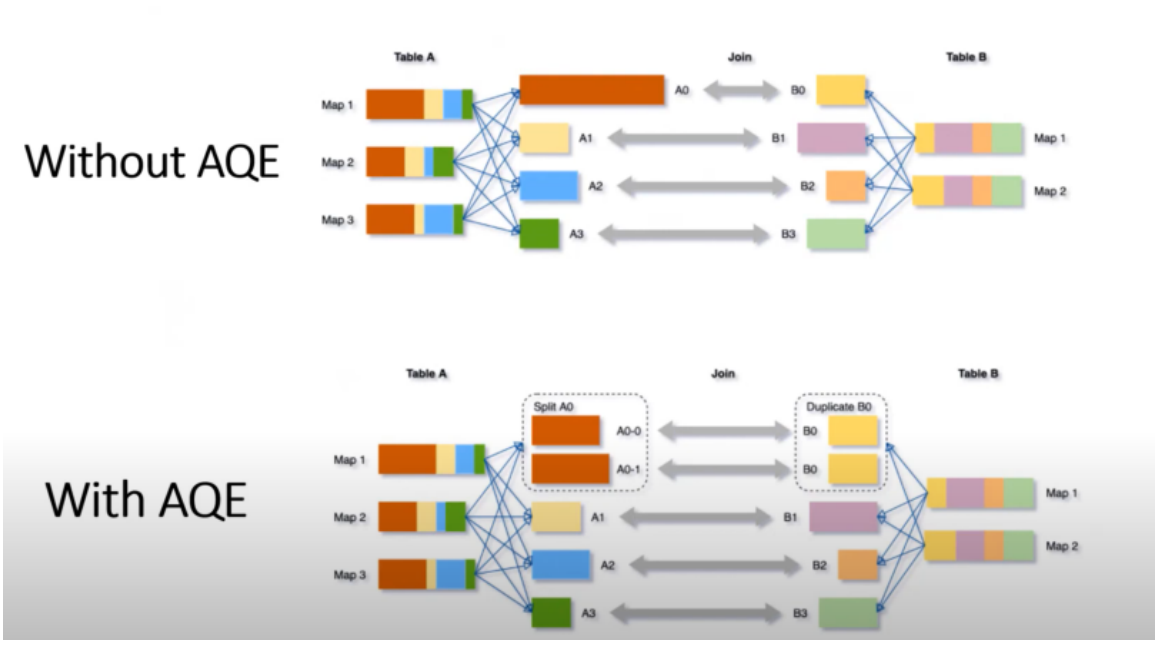
- Switching Join Strategies to broadcast Join
    - Among all different Join strategies available in Spark, broadcast hash join gives a greater performance. This strategy can be used only when one of the joins tables small enough to fit in memory within the broadcast threshold.
    - The broadcast table is loaded to all the nodes and thus less shuffling of data occur during computation for joining as one of the table have all record on each node to which join can occur.

  When one of the join tables could fit in memory before or after filtering data, AQE replans the join strategy at runtime and uses broadcast hash join.

- Optimizing Skew Join
    - 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 this 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/data frame).

- e.g. we have sales data lets say 80% sale coming form one country and remaining 20% from ither different countries if we partition based on country than 80% of data go to one partition go and remaining to other. Thus when one core choose partition having larger dataset it will take more time for it compute this partition than for core which have the partition having less dataset thus the core with small dataset will sit ideal but overall process is not completed and for other core it will take time to complete the computation. It will hit the performance time.
- so its import to not to create data skew there are some algo to remove the skew manually like bucketing and saluting
- with AQE skew join optimization detects such skew automatically from shuffle file statistics. it then splits the skewed partitions into smaller sub partitions, which will be joined to the corresponding partition form the other side.
    - suppose if one partition of very large size than it will internally



Here you can see with AQE the larger dataset are divided into smaller partitions and the copy of dimension table are also created. this will decrease the overall time as now core have to compute less record at a time.

summary

- Introduction of CBO improved performance through statistics collection
- Outdated statistics does not help to improve the performance, also lead to choosing bad execution
- intermediate processed data is written into cluster after completion of each stage and based on that the query plan is changed. so improved statistics can be called in real time data between different stages.
- As per selected physical plan, the job gets triggered for execution. when one of the stage is completed and intermediate data is available, AQE triggers optimized by updating the logical plan through runtime statists. Based on runtime statistics the execution plan is getting re optimized for better performance.