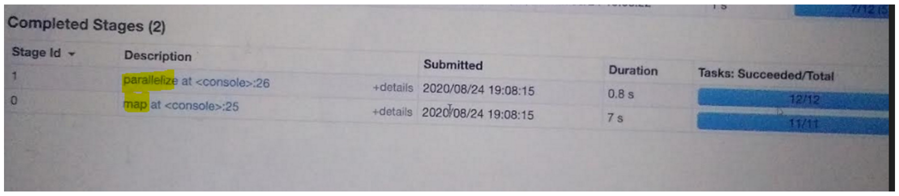
SPARK OPTIMIZATION 2

Thursday, August 26, 2021 10:22 AM

Broadcast join on low level RDD :

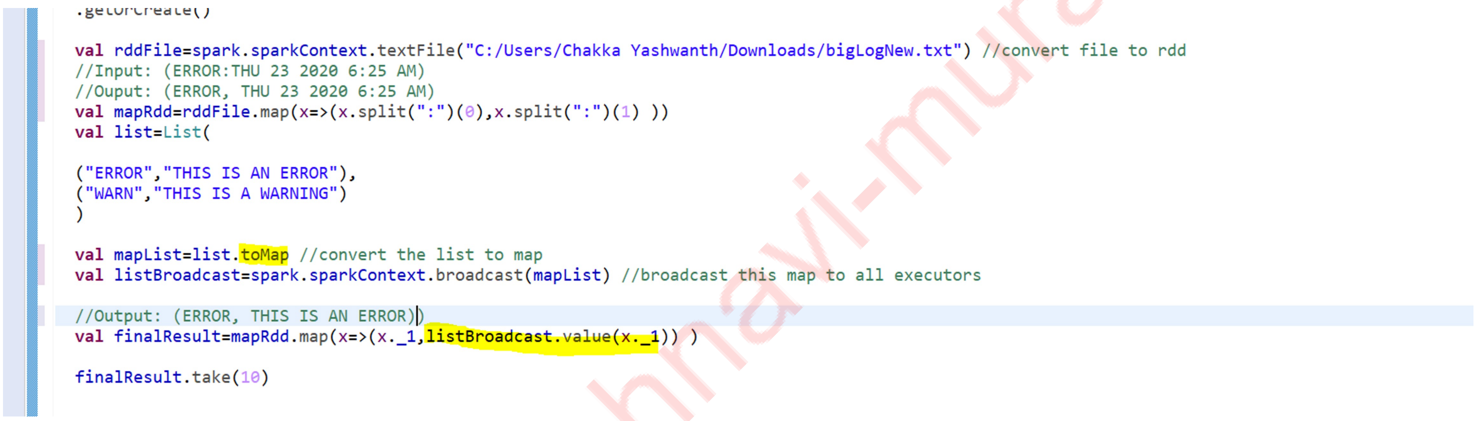
Case 1: when broadcast not used
-->Here, we have 11 partitions for map (1.4 * 1024)/128 = ~ 11 hdfs partitions
-->we have 12 partitions for parallelize function **as it is the default no. of partitions set**
-->Total no. of partitions= max(11, 12) = 12 due to which **we have 12 partitions in total**
-->2 stages in DAG due to shuffling



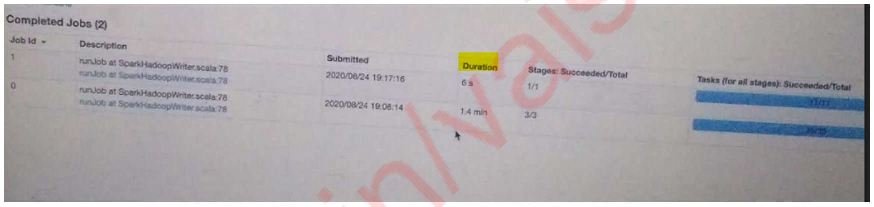
Completed Stages (2)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total
1	parallelize at <console>:26	+details	2020/08/24 19:08:15	0.8 s	12/12
0	map at <console>:25	+details	2020/08/24 19:08:15	7 s	11/11

Case 2: when broadcast used
-->we can clearly see that time consumed is very less (just 6 sec)
-->No. of tasks generated is very less
-->No shuffling of data is caused and only 1 stage in DAG



```
.getOrCreate()

val rddFile=spark.sparkContext.textFile("C:/Users/Chakka Yashwanth/Downloads/bigLogNew.txt") //convert file to rdd
//Input: (ERROR:THU 23 2020 6:25 AM)
//Ouput: (ERROR, THU 23 2020 6:25 AM)
val mapRdd=rddFile.map(x=>(x.split(":")(0),x.split(":")(1) ))
val list=List(

("ERROR","THIS IS AN ERROR"),
("WARN","THIS IS A WARNING")
)

val mapList=list.toMap //convert the list to map
val listBroadcast=spark.sparkContext.broadcast(mapList) //broadcast this map to all executors

//Output: (ERROR, THIS IS AN ERROR))
val finalResult=mapRdd.map(x=>(x._1,listBroadcast.value(x._1)) )

finalResult.take(10)
```



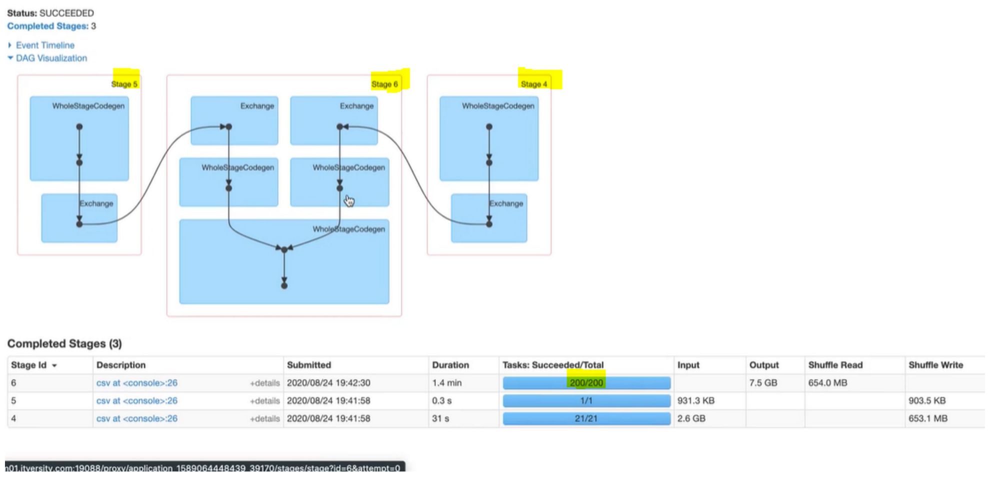
Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2020/08/24 19:17:16	6 s	1/1	[illegible]
0	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2020/08/24 19:08:14	1.4 min	3/3	[illegible]

Using Broadcast join in Data Frame:
-->By default , due to spark internal optimizations if **we try to perform join on 2 tables, it will infer which is smaller table and perform broadcast join on it by default**.

Case 1: when broadcast join is purposedly disable

-->**when shuffling happens in Data frame by default 200 partitions are created**
-->**In RDD, no. of partitions before and after shuffling is same**
-->**Spark performs "shuffle sort merge"** join in this case
-->Total tasks 222 : 200 (shuffling) + 22 (hdfs partitions big table) + 1(hdfs partitions small table)
-->Time taken: 1.4min (shuffling) + 32 sec(to infer schema) ==> **optimizations to be done here**



Status: SUCCEEDED
Completed Stages: 3

- Event Timeline
- DAG Visualization

Completed Stages (3)

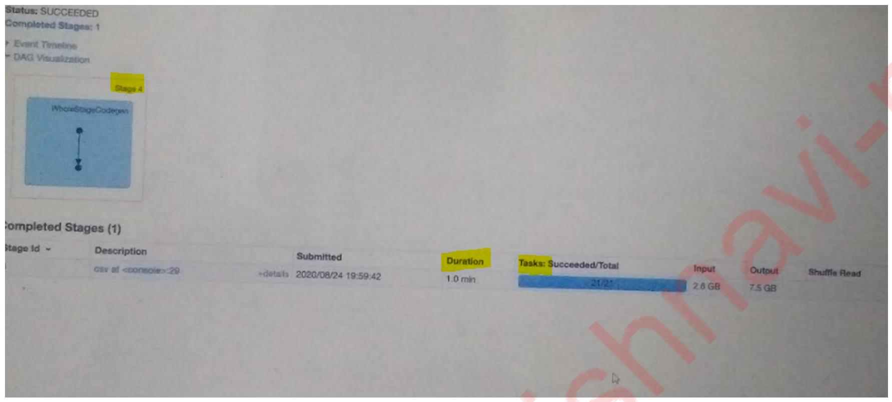
Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
6	csv at <console>:26	+details	2020/08/24 19:42:30	1.4 min	200/200		7.5 GB	654.0 MB	
5	csv at <console>:26	+details	2020/08/24 19:41:58	0.3 s	1/1	931.3 KB			903.5 KB
4	csv at <console>:26	+details	2020/08/24 19:41:58	31 s	21/21	2.6 GB			653.1 MB

n01.itversity.com:19088/proxy/application_1589064448439_39170/stages/stage?id=6&attempt=0

Case 2: when broadcast join is done

1. **Optimizations**: Lessen time to execute and use broadcast join

-->Total tasks : 22 = (21 hdfs partitions big table) + (1 hdfs partition small table)
--> Time consumed = 1min (no infer schema, used broadcast join)
-->Only 1 stage in DAG



Status: SUCCEEDED
Completed Stages: 1

- Event Timeline
- DAG Visualization

ompleted Stages (1)

tage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read
	csv at <console>:29	+details	2020/08/24 19:59:42	1.0 min	21/21	2.6 GB	7.5 GB	

When and why to increase Driver memory?

-->If we have a huge result set (millions of records), and when we use functions like "collect()" , "take() "
-->The driver tries to bring the results and store it in its storage unit.
-->If the result is very high then driver fails to print results on driver machine
-->It is advised not to print huge results in driver machine

Few of the parameters we can set as part of manual allocation:

1. --num-executors : To set the no. of executors
2. --executor-memory : To set memory for each executor
3. --executor-cores : To set the no. of executor cores in each executor
4. --driver-memory: To set the memory for driver

Sample manual allocation:

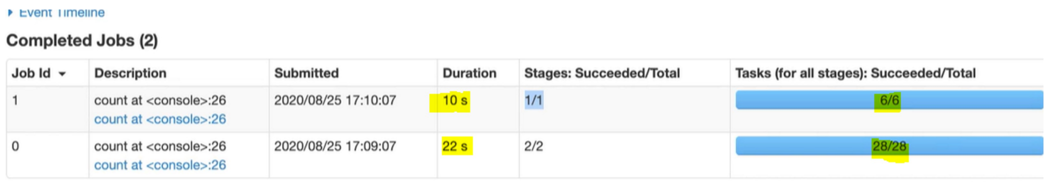
```
spark2-shell --conf spark.dynamicAllocation.enabled=false --master yarn --num-executors 6
--executor-cores 2 --executor-memory 3G --conf spark.ui.port=4063
```

Session 15:

Demo on repartition and coalesce :

Repartition()	**Coalesce** ()
1. To increase/ decrease no. of partitions 2. Not recommended for decreasing partitions	1. To decrease no. of partitions 2. Best for decreasing partitions 3. Shuffling of data mostly avoided 4. All the partitions may not be of

3. Full shuffle of data takes place 4. All the partitions will be of equal size 5. More time consumed 6. Val rdd2=rdd1.repartition(6) 7. Total tasks: 22(hdfs partitions) +6 = 28 8. Time taken: 28sec 9. Time complexity is O(nlogn) For 1000 records, O(1000 *log1000) = 10,000 ie to sort 1000 records it has to perform 10000 operations	equal size 5. Less time consumed and more optimized 6. Val rdd2=rdd1.coalesce(6) 7. Total tasks: 6 8. Time taken: 10sec 9. Time complexity is O(logn) For 1000 records, O(log 1000)= 100 ie to sort 1000 records it needs 100 operations


▸ Event Timeline

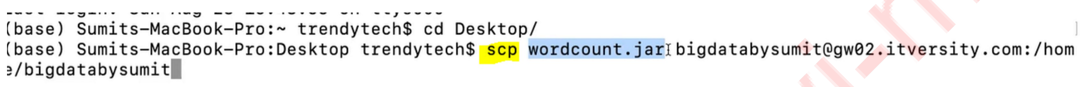
Completed Jobs (2)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	count at <console>:26 count at <console>:26	2020/08/25 17:10:07	10 s	1/1	6/6
0	count at <console>:26 count at <console>:26	2020/08/25 17:09:07	22 s	2/2	28/28

2. To run a spark job on the cluster

-->First, convert the project into jar (your project -> export -> jar)
-->connect to the edge node, where ur jar will run in cluster

To transfer the jar into cluster:
-->scp jar_name login_id /path


```
(base) Sumits-MacBook-Pro:~ trendytech$ cd Desktop/
(base) Sumits-MacBook-Pro:Desktop trendytech$ scp wordcount.jar bigdatabysumit@gw02.itversity.com:/hom
e/bigdatabysumit
```

Command to run the spark jar in cluster:

--spark2-shell \ ==>to run the spark shell
--class rdd_to_DF \ ==> which specific appl(program) to run in the entire project files
--master yarn \ ==> to run it in "yarn"
--deploy-mode cluster \ ==>where driver will also be in cluster (cluster mode), driver will be in one of the worker node
--num-executors 4 \ ==>num of executors to be given (optional, will be taken care by dynamic allocation)
--executor-memory 4G \ ==> memory for each executor (optional, will be taken care by dynamic allocation)
Wordcount.jar bigLog.csv ==> "name of jar", to run the program file that is actually needed

Session 16:

Join optimizations

Main 2 things we need to prioritize while performing join operations:
1. **Try to avoid or minimize shuffling of data**
2. **Increase parallelism so that execution time gets reduced and all tasks work equally**

More Scenarios:

1. Both are small tables : Try to solve using DB Sql rather than big data technologies
2. 1 large table 1 small table : Broadcast join
3. 2 large tables : we need to minimize shuffling in this case and inc parallelism

How to minimize shuffling :

1. **Filter and aggregate the data before shuffling**
   -->so that data to be shuffled gets reduced

2. **How to increase parallelism ?**
   1. suppose we have 250 tasks (50 executors, 5 cores each)
   2. shuffle partitions (200)
   3. we have 100 distinct keys ie at max 100 tasks can run parallely

Min (total cpu cores, total shuffle partitions, total distinct keys)
Ie min (250, 200, 100) = 100 tasks ie (150 tasks are idle (150 cores), 50 tasks idle (shuffle partitions)

3. **Increase no. of distinct keys : Salting (high cardinality)**
2. **Inc no. of shuffle partitions**
3. **Increase resources if required**

What are skew partitions and how to handle them ?
-->**Partitions in which data is unevenly distributed** ie few partitions get huge data to process while others get less data.

Ex. If we have 10 lakhs records where 6 lakh records belong to 100 users
Ie 100 keys (customer_id) will go to 100 tasks which does the max work
-->**Each distinct key goes to one particular partition**
-->**we can handle it well using bucketing, partitioning, salting and sorting**

What is Sort aggregate vs Hash aggregate ?

Sort Aggregate	Hash Aggregate
1. **Data is first sorted based on keys (grouping columns) before aggregations are performed**. 2. It takes more time to perform the aggregation. 3. Sorting of data is time consuming	1. **It creates a hash table for every unique value it comes across. If the record appears again then its corresponding count value is increased in hash table** 2. It takes very less time to perform the aggregation 3. Sorting of data is not required 4. Hash table structure is created in off heap memory

Internals of Sort Aggregate and Hash Aggregate :

Why query1 has been picked up to be sort aggregate and query2 as Hash Aggregate?

Query 01:
-->We are trying to find the total count of orders placed by each customer in every month.
--> we are trying to get the month no. in order to sort it (monthnum, jan 1, feb 2)
-->first(xxxxx) => takes only first value and ignores all the next values
-->here, first (date_format(order_date, 'M') => returns "1", "2" (month no. in strings)

When the physical plan tries to pick up the best cost effective plan to execute it. It finds that "monthnum" is of string and strings are immutable.
-->Due to which it can't append the values to hash table directly
-->Due to this reason system picks up "sort aggregate" to be performed
-->**For immutable data types , system picks sort aggregate to be performed**

```
spark.sql("select order_customer_id, date_format(order_date, 'MMMM') orderdt,  count(1) cnt,
first(date_format(order_date,'M')) monthnum  from orders group by order_customer_id, orderdt
order by cast(monthnum as int)").explain
```

It took 3.9 minutes to complete this query - sort aggregate

```
sort aggregate
===============


customer_id:month      value
1024:january        1  "1"
1024:january        1  "1"
1024:january        1  "1"
1024:january        1  "1"
1025:january        1  "1"
1025:january        1  "!"
```

first the data is sorted based on the grouping columns.

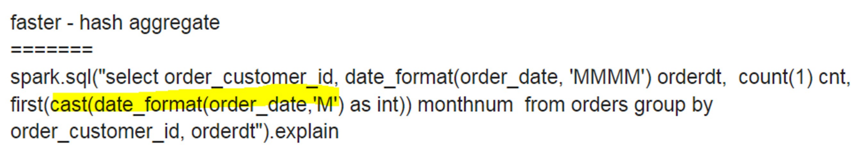
1024:january ,{1,1,1,1,1}

sorting of data takes time..

2000
O(nlogn)
1000 * log(1000)
1000 * 10 = 10000

Why Query 02 is picked up as Hash Aggregate?

-->The "monthnum" which is a string is casted to "integer" due to which system can add them into hash table directly
->For each similar record we encounter we just append the values corresponding to it in hash table
-->Due to this system picks to perform Hash Agrregate on this query
-->**For hash aggregate we need to use mutable data types**



```
faster - hash aggregate
=======
spark.sql("select order_customer_id, date_format(order_date, 'MMMM') orderdt,  count(1) cnt,
first(cast(date_format(order_date,'M') as int)) monthnum  from orders group by
order_customer_id, orderdt").explain
```

Catalyst Optimizer:
-->Structured API's (DF,DS, spark sql) perform better than raw rdd's
-->catalyst optimizer will optimize the execution plan for structured API 's
-->It is rule based optimization where based on rules written optimization takes place

Stages of Query Execution:

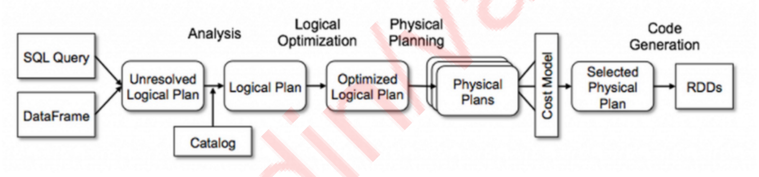
Step 01 : **Parsed Logical Plan (Unresolved logical plan)**
-->Here the query is checked for syntax errors

Step 02: **Analytical Logical Plan (Resolved Logical Plan)**
-->Here, the query is checked for existance of table and column names from the catalog available

Step 03: **Optimized Logical Plan (Catalyst optimization)**
-->Many in-built rules are written on how the optimization has to take place in different scenarios
-->We can write our own optimizations as well
-->Ex. Predicate push down, filtering data

Step 04: **Physical plan**
-->Here, the physical plan (actual execution plan) is selected based on cost effectiveness
-->Ex. Hash/Sort aggregate

Step 05: **Conversion into RDD's**
-->They are finally converted into rdd's which are basic structure of storing data in spark
-->RDD's are the low level programming constructs which are directly sent to executors for processing



Sample User written Optimization Rule:
-->Here, if a number is multiplied by 1, we add the rule saying that this multiplication need not be calculated
As the answer will be the same.

```
import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
import org.apache.spark.sql.catalyst.rules.Rule
import org.apache.spark.sql.catalyst.expressions.Multiply
import org.apache.spark.sql.catalyst.expressions.Literal

object MultiplyOptimizationRule extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transformAllExpressions {
    case Multiply(left,right) if right.isInstanceOf[Literal] &&
      right.asInstanceOf[Literal].value.asInstanceOf[Integer] == 1 =>
      println("optimization of one applied")
      left
  }
 }

spark.experimental.extraOptimizations = Seq(MultiplyOptimizationRule)
```

Sample connection block on converting to DF from sql table:

```
spark-shell --driver-class-path /usr/share/java/mysql-connector-java.jar

val connection_url ="jdbc:mysql://cxln2.c.thelab-240901.internal/retail_db"

val mysql_props = new java.util.Properties

mysql_props.setProperty("user","sqoopuser")

mysql_props.setProperty("password","NHkkP876rp")

val orderDF = spark.read.jdbc(connection_url,"orders",mysql_props)

orderDF.show()
```

KEYWORDS:

1. Broadcast join: Used to send the copies of data to all executors. Used when we have only 1 big table.
2. Optimization on using coalesce() rather than repartition while reducing no. of partitions
3. Join optimizations:

a. To avoid or minimize shuffling of data
b. To increase parallelism

1. How to avoid/minimize shuffling?

a. Filter and aggregate data before shuffling
b. Use optimization methods which require less shuffling (coalesce())

1. How to increase parallelism ?

a. **Min (total cpu cores, total shuffle partitions, total distinct keys)**
b. Use salting to increase no. of distinct keys
c. Increase default no. of shuffle partitions
d. Increase resources to inc total cpu cores

1. Skew partitions : Partitions in which data is unevenly distributed. Bucketing, partitioning, salting can be used to handle it.
2. Sort aggregate: Data is sorted based on keys and then aggregated. More processing time
3. Hash aggregate: Hash table is created and similar keys are added to the same hash value. Less processing time.
4. Stages of execution plan :

a. Parsed logical plan (unresolved logical plan) : To find out syntax errors
b. Analytical logical plan (Resolved logical plan) : Checks for column and table names from the catalog.
c. Optimized logical plan (Catalyst optimization) : Optimization done based on built in rules.
d. Physical plan : Actual execution plan is selected based on cost effective model.
e. Conversion into Rdd : Converted into rdd and sent to executors for processing.

LinkedIn Post:

Sharing below few very generic techniques and configurations that we have tried and will definitely give you head start in your performance optimization :-

1. Add spark configuration not to start the 2nd attempt if there is failure in 1st attempt. It will help you to find job failure in 1st attempt itself-
--conf spark.yarn.maxAppAttempts=1

2. Follow suggestions provided by the unravel.
for example- increase shuffle partition if suggested and advised(--conf spark.sql.shuffle.partitions=1000)

3. Explicitly limit the executors, if not your spark job will take maximum cores available in the cluster and hang there for long time.
-- conf spark.dynamicAllocation.maxExecutors=25

4. Make sure twice that you are not configuring Cores more than 4 at any cost.

5. Try reducing the number of HDFS files getting generated.Try to come up with partition number based on your data.
By default partition number is 200 so if you are writing data in hive, it will create 200 HDFS files. And it results in more time while moving these many HDFS files to table local.

1. Add G1GC garbage collector configuration , if your job is exiting with 143 morning heap issue OR is your GC time is more than Spark task execution time

--conf 'spark.executor.extraJavaOptions=XX:+UseG1GC'

2. Kryo Sterlizer(if anyone is not not using) as it is the best for serde the objects
-- conf spark.serialize=org.apache.spark.serializer.KryoSerializer

3. You can try updating HDFs file write algorithm version configuration as well with "2" if it is configured 1 or not configured as version 2 gives better result while writing files to hdfs

-- conf spark.hadoop.mapreduce.fileoutputcommitter.algorith.version=2

Top 5 Hive Optimization Techniques

1. 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.

for example partitioning can be done on state column.

2. Bucketing

Works by dividing the data into smaller segments.

These segments are created based on system defined hash functions (not logical).

We finally scan only one bucket.

bucketing can be done on columns with high cardinality.

for example bucketing can be done on productid column.

3. Join optimizations techniques

Map side join , Bucket Map Join, Sort Merge Bucket Join also called as SMB join.

All of them try to minimize shuffling.

4. Use Orc file format with a compression like snappy.

Orc can reduce the data storage by 75% of the original.

It uses techniques like predicate push-down, compression, and more to improve the performance of the query.

Snappy provides a fast compression.

5. UDF's are not very optimized.

filter operations are evaluated left-to-right.

For best performance, put UDFs on the right in an ANDed list of expressions in the WHERE clause.

What other hive optimizations have you worked on?

SPARK OPTIMIZATIONS:

We have lot of optimization technique, will see some of them for the next few days.
1. Shuffle partition size :
Based on our dataset size, number of cores, and memory, Spark shuffling can benefit or harm the jobs.
When we are dealing with less amount of data, we need to reduce the shuffle partitions otherwise we will end up with many partitioned files with a fewer number of records in each partition. which results in running many tasks with lesser data to process.
On other hand, when we have too much of data and having less number of partitions results in few longer running tasks and some times we may also get out of memory error.
2. Bucketing :
The Bucketing is commonly used to optimize performance of a join query by avoiding shuffles of tables participating in the join.
df.write.bucketBy(n, “column”).saveAsTable(“bucketed_table”)
The above code will create a table (bucketed table) with row sorted based on the column with “n” — no.of.buckets.
Bucketing can benefit when pre-shuffled bucketed tables are used more than once in the query.
3. Serialization :
Serialization plays an important role in the performance for any distributed application. By default, Spark uses Java serializer.
Spark can also use another serializer called ‘Kryo’ serializer for better performance.
Kryo serializer is in compact binary format and offers processing 10x faster than Java serializer.
To set the serializer properties:
conf.set(“spark.serializer”, “org.apache.spark.serializer.KryoSerializer”)
4. API selection :
Spark introduced three types of API to work upon — RDD, DataFrame, DataSet
* RDD is used for low level operation with less optimization
* DataFrame is best choice in most cases due to its catalyst optimizer and low garbage collection (GC) overhead.
* Dataset is highly type safe and use encoders. It uses Tungsten for serialization in binary format.
5. Avoid UDF’s :
UDF’s are used to extend the functions of the framework and re-use this function on several DataFrame.
UDF’s are once created they can be re-use on several DataFrame’s and SQL expressions.
But UDF’s are a black box to Spark hence it can’t apply optimization and we will lose all the optimization Spark does on Dataframe/Dataset. Whenever possible we should use Spark SQL built-in functions as these functions designed to provide optimization.
6. Use Serialized data format’s :
Most of the Spark jobs run as a pipeline where one Spark job writes data into a File and another Spark jobs read the data, process it, and writes to another file for another Spark job to pick up.
When we have such use cases, prefer writing an intermediate file in Serialized and optimized formats like Avro, Kryo, Parquet. Any transformations on these formats performs better than text, CSV, and JSON.