Running Notes

Week14: Apache Spark Optimization Part2

Week14: Apache Spark - Optimization Part2

Spark Optimization Session - 12
==========

Broadcast join can be used when we have 1 large table and 1 small table.

and we want to join these.

RDD (lower level API's)
==========

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

we will have 2 rdd's

one of them will be large

one of them will be smaller.

ERROR: Thu Jun 04 10:37:51 BST 2015
WARN: Sun Nov 06 10:37:51 GMT 2016
WARN: Mon Aug 29 10:37:51 BST 2016
ERROR: Thu Dec 10 10:37:51 GMT 2015
ERROR: Fri Dec 26 10:37:51 GMT 2014
ERROR: Thu Feb 02 10:37:51 GMT 2017
WARN: Fri Oct 17 10:37:51 BST 2014
ERROR: Wed Jul 01 10:37:51 BST 2015
WARN: Thu Jul 27 10:37:51 BST 2017
WARN: Thu Oct 19 10:37:51 BST 2017

the size of this data is 1.4 GB (Large dataset)

val rdd1 = sc.textFile("bigLogNew.txt")

Input:
ERROR: Thu Jun 04 10:37:51 BST 2015
WARN: Sun Nov 06 10:37:51 GMT 2016
WARN: Mon Aug 29 10:37:51 BST 2016

ERROR: Thu Dec 10 10:37:51 GMT 2015
ERROR: Fri Dec 26 10:37:51 GMT 2014
ERROR: Thu Feb 02 10:37:51 GMT 2017
WARN: Fri Oct 17 10:37:51 BST 2014
ERROR: Wed Jul 01 10:37:51 BST 2015
WARN: Thu Jul 27 10:37:51 BST 2017
WARN: Thu Oct 19 10:37:51 BST 2017

output:
(ERROR,Thu Jun 04 10:37:51 BST 2015)
(WARN, Sun Nov 06 10:37:51 GMT 2016)

MAP TRANFORMATION

```
val rdd2 = rdd1.map(x => (x.split(":")(0),x.split(":")(1))
```

rdd2 will have something like the below:

RDD2 is Large
(ERROR,Thu Jun 04 10:37:51 BST 2015)
(WARN, Sun Nov 06 10:37:51 GMT 2016)

RDD3 IS SMALL
("ERROR",0)
("WARN",1)


```
val rdd4 = rdd2.join(rdd3)
```

(ERROR,(Thu Jun 04 10:37:51 BST 2015,0))
(WARN, (Sun Nov 06 10:37:51 GMT 2016,1))

```
val a = Array(
("ERROR",0),
("WARN",1)
)

val rdd3 = sc.parallelize(a)

val a = Array(
```

```
("ERROR",0),
("WARN",1)
)

val keyMap = a.toMap

val bcast = sc.broadcast(keyMap)

val rdd1 = sc.textFile("bigLogNew.txt")

val rdd2 = rdd1.map(x => (x.split(":")(0),x.split(":")(1)))

val rdd3 = rdd2.map(x => (x._1,x._2,bcast.value(x._1)))

rdd3.saveAsTextFile("joinresults2")
```

Dataframes (Structured API's)

Spark Optimization Session - 13
==========

we will have 2 dataframes.

one of them will be large and other will be small.

orders is 2.6 GB
customers is around 900 kb

customers - customer_id (small)

orders - order_customer_id (large)

we will try to create dataframe for each of these.

```
spark2-shell --conf spark.dynamicAllocation.enabled=false --master yarn --num-executors 21


val customerDF =
spark.read.format("csv").option("header",true).option("inferSchema",true).option("path","custom
ers.csv").load
```

```
val orderDF =
spark.read.format("csv").option("header",true).option("inferSchema",true).option("path","orders.c
sv").load

spark.conf.set("spark.sql.autoBroadcastJoinThreshold",-1)

val joinedDF = customerDF.join(orderDF,customerDF("customer_id") ===
orderDF("order_customer_id"))
```

how many partitions are there in this big DF - 21

and for the small dataframe we have just a single partition.

whenever we do shuffling in case of structured API's we ge 200 partitions by default.

if I do groupBy on a Dataframe then we will get 200 partitions after the shuffling is done.

500 mb file..

rdd - groupBy then

before groupBy we had 4 partitions..

after groupBy the partitions remain the same that means 4 partitions.

Spark Optimization Session - 14
==========

2 things..

1. lets not infer the schema and save time.

2. use a broadcast join.

```
import org.apache.spark.sql.types._

val ordersSchema = StructType(
List(
StructField("order_id",IntegerType,true),
StructField("order_date",TimestampType,true),
StructField("order_customer_id",IntegerType,true),
StructField("order_status",StringType,true)
```

```
)
)

val customerDF =
spark.read.format("csv").option("header",true).option("inferSchema",true).option("path","custom
ers.csv").load

val orderDF =
spark.read.format("csv").schema(ordersSchema).option("header",true).option("path","orders.csv
").load


val joinedDF = customerDF.join(orderDF,customerDF("customer_id") ===
orderDF("order_customer_id"))

joinedDF.take(1000000)
```

increase the --driver-memory when you are collecting more data on driver machine.

otherwise you will get out of memory error.

--num-executors

--driver-memory

--executor-memory

--executor-cores

==========
Spark Optimization Session - 15
==========

1. demo on repartition vs coalesce when we want to decrease the number of partitions. - coalesce

2. using spark-submit we will try to submit a job.

while submitting the jar we will see

a) client

b) cluster

you have a 1 gb file

and you create a rdd on top of it.

8 partitions..

a cluster of 4 nodes..

on each machine there might be 2 partitions..

rdd.coalesce(4)

when we are using coalesce then the resultant partitions can be of unequal sizes..

when we are using repartition then full shuffling is involved which is time consuming but we know that the resultant partitions will be of similar size..

1. write your code in IDE like eclipse/intelliJ

2. bundle your code as a java jar file and export the jar.

3. move this jar file to your edge node/gateway machine.

scp wordcount.jar bigdatabysumit@gw02.itversity.com:/home/bigdatabysumit

wordcount.jar

I want to run LogLevelGrouping

bigLogNew.txt - 1.4 GB (11 partitions)

```
spark2-submit \
--class LogLevelGrouping \
--master yarn \
--deploy-mode cluster \
```

```
--executor-memory 3G \
--num-executors 4 \
wordcount.jar bigLogNew.txt
```

since the deploy mode is cluster mode that means our driver is running on one of the executors residing in the cluster.

driver wn02.itversity.com:37973 (cluster) => recommended
driver gw02.itversity.com:52754 (client)

driver is running on one of the worker node..

```
spark2-submit \
--class LogLevelGrouping \
--master yarn \
--executor-memory 3G \
--num-executors 4 \
wordcount.jar bigLogNew.txt
```

Spark Optimization Session - 16
==========

Join opimizations
==========

we have 2 things which we work for:
1. we always want to avoid or minimize the shuffling.
2. we want to increase the parallelism.

1 large table and 1 small table - broadcast join
no shuffling required.

2 large tables - there is no way to completely avoid the shuffling.
but we can think of minimizing it.

To make sure we shuffle less data
==========

try to filter the data as early as possible before the shuffle. cut down the size before the shuffle phase. and do aggregation before.

filter & aggregation try to do before the shuffle.

Increase the Parallelism
==========

100 node cluster each worker node having 64 GB RAM & 16 CPU Cores.

1. how many cpu cores we are grabbing.

50 executors with 5 cores each - 250 total cpu cores (at the max 250 tasks)

2. whenever shuffling happens in case of structured API's

we get 200 partitions after the shuffle.

spark.sql.shuffle.partitions

200 partitions - (at the max 200 tasks running in parallel)

3. in your data you have 100 distinct keys..

after the shuffle only at max 100 partitions will be full and other will be empty.

whenever the cardinality of data is low then some of the partitions will be empty.

(at the max 100 tasks will be running in parallel)

we grabbed 250 cpu cores (50 executors X 5)

shuffle partitions - 200

we have just 100 distinct keys..

min(Total CPU Cores , Number of Shuffle partitions, Number of Distinct Keys)

min(250, 200, 100)

100

you can try increasing the cardinality - salting.

we can increase the Number of shuffle partitions if required.

we can try grabbing more resources if feasible

Skew Partitions
==========

customers table and orders table..

customers table - customer_id

orders table - order_customer_id

one of the Customer (APPLE) has placed a lot of orders..

10000000 - 10 million total orders

9000000 - 9 million orders are just for one customer(APPLE)

same key always goes to same partition..

200 shuffle partitions..

1 of the partition will be holding majority of data..

which ever task is working on this heavily loaded partition will be very slow.

and other tasks will complete quickly..

your job is dependent on the slowest performing task..

there should not be partition skew, else the job will be delayed.

orders - order_customer_id
customers - customer_id

bucketing & sorting on the datasets on join column

orders into 32 buckets.. - 32 files

customers into 32 buckets.. - 32 files

15 minutes to get the bucketing and sorting done.

when you try to join these 2 tables it will be very quick.

SMB join - shuffling

if you are not doing bucketing and sorting

and you are doing a plain join

30 minutes * 1000 = 30000 minutes

bucketed both the tables on the join column and sorted it.

60 minutes to do bucketing and sorting.

join - 5 minutes.. * 1000 = 5000 minutes

SMB join.

Connecting to External Data Source
==========

in mysql we will have a table and we will try to create a dataframe by directly connecting from that.

mysql-connector-java.jar

```
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()
```

Spark Optimization Session - 17
==========

Sort Aggregate vs Hash Aggregate

orders.csv - 2.6 GB

```
order_id,order_date,order_customer_id,order_status
1,2013-07-25 00:00:00.0,11599,CLOSED
2,2013-07-25 00:00:00.0,256,PENDING_PAYMENT
3,2013-07-25 00:00:00.0,12111,COMPLETE
4,2013-07-25 00:00:00.0,8827,CLOSED
5,2013-07-25 00:00:00.0,11318,COMPLETE
6,2013-07-25 00:00:00.0,7130,COMPLETE
7,2013-07-25 00:00:00.0,4530,COMPLETE
8,2013-07-25 00:00:00.0,2911,PROCESSING
9,2013-07-25 00:00:00.0,5657,PENDING_PAYMENT
```

we want to find out the number of orders which are placed by each customer in each month.

grouping based on order_customer_id and Month - count of number of orders.

```
spark2-shell --conf spark.dynamicAllocation.enabled=false --master yarn --num-executors 11
--conf spark.ui.port=4063
```

```
val orderDF =
spark.read.format("csv").option("inferSchema",true).option("header",true).option("path","orders.c
sv").load

orderDF.createOrReplaceTempView("orders")

spark.sql("select * from orders").show

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)").show
```

It took 3.9 minutes to complete this query

```
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 order by monthnum").show
```

It took 1.2 minutes to complete this query

Spark Optimization Session - 18
==========

```
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

```
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 order by monthnum").explain
```

It took 1.2 minutes to complete this query - hash aggregate

2 questions
==========

1. difference between sort aggregate and hash aggregate

2. why in query1 it used sort aggregate
   and why in query2 it used hash 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

2000 * 11 = 22000

sort aggregate takes lot of time when the data grows..

1024:january 1 "1"
1024:january 1 "1"
1024:january 1 "1"
1025:january 1 "1"
1025:january 1 "!"

Hash Aggregate
==========

hash table

==========

customer_id:month value
1024:january 3 "1"
1025:january 2 "1"

no sorting is required..
additional memory is required to have the hashtable kind of structure.

1000 rows...

sort aggregate = O(nlogn) = 1000 * 10 = 10000 operations
hash aggregate = O(n) = 1000

this hash table kind of structure is not a part of container..
rather this additional memory is grabbed as part of off heap memory..

this memory is not the part of your jvm..

question 2: why in the first query it used sort aggregate and why in second query it used hash aggregate..

slower - sort aggregate
======
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").explain

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

customer_id:month value
1024:january 3 1

month number was string

string is immutable

when we are using hash aggregate we should have mutable types in the values

Spark Optimization Session - 19
==========

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

Rule Based Optimization.

Many rules are already available. Also if we want we can add our own optimization rules.

Students.csv - 60 mb

student_id,exam_center_id,subject,year,quarter,score,grade
1,1,Math,2005,1,41,D
1,1,Spanish,2005,1,51,C
1,1,German,2005,1,39,D
1,1,Physics,2005,1,35,D
1,1,Biology,2005,1,53,C
1,1,Philosophy,2005,1,73,B
1,1,Modern Art,2005,1,32,E
1,1,History,2005,1,43,D
1,1,Geography,2005,1,54,C

```
val df1 =
spark.read.format("csv").option("header",true).option("inferSchema",true).option("path","/Users/t
rendytech/Desktop/students.csv").load

df1.createOrReplaceTempView("students")

spark.sql("select * from students").show
```

1. Parsed Logical Plan - un-resolved

our query is parsed and we get a parsed logical plan - unresolved

it checks for any of the syntax errors.

syntax is correct

2. Resolved/Analysed Logical plan..

it will try to resolve the table name the column names etc..

if the columnname or table name is not available then we will get analysis exception.

if we have referred to the correct columnnames and table name

3. Optimized Logical Plan - catalyst optimizer.

filter push down

combining of filters..

combining of projections

There are many such rules which are already in place.

If we want we can add our own rules in the catalyst optimizer.

consider you are doing a Aggregate.

as per the logical plan lets say it says we have to do Aggregate.

in physical plan..

physical plan1
==========
sortAggregate

physical plan2
==========
HashAggregate

It will select the physical plan which is the most optimized one with minimum cost.

This selected physical plan is converted to Lower Level API's
RDD code.

Spark Optimization Session - 20
==========

student_id,exam_center_id,subject,year,quarter,score,grade
1,1,Math,2005,1,41,D
1,1,Spanish,2005,1,51,C
1,1,German,2005,1,39,D
1,1,Physics,2005,1,35,D
1,1,Biology,2005,1,53,C
1,1,Philosophy,2005,1,73,B
1,1,Modern Art,2005,1,32,E
1,1,History,2005,1,43,D
1,1,Geography,2005,1,54,C

a * b

if b is 1 then return a

Catalyst optimizer

RDD vs Structured API's

Parsed Logical Plan - Unresolved

Analysed/Resolved Logical Plan

Optimized Logical Plan

Physical Plan

```
val df1 = spark.read.format("csv").option("header",
true).option("inferSchema",true).option("path","/Users/trendytech/Desktop/students.csv").load

df1.createOrReplaceTempView("students")

spark.sql("select student_id from (select student_id, exam_center_id from students) where
student_id <5").explain(true)

spark.sql("select student_id,sum(score) from (select student_id, exam_center_id,score from
students where exam_center_id=5) where student_id < 5 group by student_id ").explain(true)
```

http://localhost:4040/

```
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)
```