TRENDY TECH

UPLIFT YOUR CAREER!

Running Notes

Week13: Apache Spark Optimization Part1

Week13: Apache Spark - Optimization Part1

Spark Optimization Session - 1
=================================

Spark Performance Optimizations / Tuning Spark Jobs

There are basically 2 main areas we should focus on..

1. cluster configuration level - resource level optimization

2. Application code level - how we write the code.

Partitioning, bucketing, cache & persist, avoid/minimize shuffling of data, join optimizations, using optimized file formats, using reduceByKey instead of groupByKey

Spark Optimization Session - 2
==================================

resources - memory (RAM) , CPU cores (Compute)

our intention is to make sure our job should get the right amount of resources.

10 Node cluster (10 worker nodes)

16 cpu cores

64 GB RAM

Executor (it is like a container of resources)

1 node can hold more than one executor

in a single worker node we can have multiple executors (multiple containers)

container - cpu cores + memory (RAM)
executor/container/JVM

16 cores , 64 gb ram

there are 2 strategies when creating containers.

1. Thin executor - intention is to create more executors with each executor holding minimum possible resources.

total of 16 executors , with each executor holding

each executor - 1 core, 4 gb ram

Drawback
==========
1. In this scenario we will be losing the benefits of multithreading.

2. A lot of copies of broadcast variable are required..
each executor should receive its own copy.

2. Fat executor - intention is to give maximum resources to each executor.

16 cores, 64 gb ram

you can create a executor which can hold 16 cpu cores and 64 gb ram..

Drawbacks
==========

1. It is obverved that if the executor holds more than 5 cpu cores then the hdfs throughput suffers.

2. if the executor holds very huge amount of memory, then the garbage collection takes a lot of time.

garbage collection means removing unused objects from memory.

Spark Optimization Session - 3
===============================

10 Nodes

16 cores

64 GB Ram

1. Tiny executors
2. Fat executors

16 cores , 64 GB Ram

1 core is given for other background activities
1 gb RAM is given for operating system

in each node we are now left with 15 cores, 63 GB Ram

=> we want multithreading within a executor (> 1 cpu core per executor)
=> we do not want our hdfs throughput to suffer (it suffers when we use more that 5 cores per executor)

5 is the right choice of number of cpu cores in each executor.

15 cores, 63 GB Ram - each machine

we can have 3 executors running on each worker node

each executor will contain 5 cpu cores and 21 GB Ram.

out of this 21 GB RAM some of it will go as part of overhead (off heap memory)

max (384MB , 7% of executor memory)

= 1.5 GB (overhead / off heap memory) - this is not part of containers.

21 - 1.5 = ~19 GB

that mean in each worker node we have 3 executors.

each executor holds - 5 cpu cores, 19 GB RAM

we have a 10 node cluster/ worker nodes

10 * 3 = 30 (executors across the cluster)

30 executors with each executor holding -
5 cpu cores, 19 GB RAM

1 executor out of these 30 will be given for YARN Application Manager

30 - 1 = 29 executors...

Spark Optimization Session - 4
================================

5 worker nodes

8 cpu cores , 32 GB RAM

out of 32 GB usable is 24 GB for yarn containers

executor/container - should have memory in between 1 & 4 GB

in each worker node there are 8 physical cpu cores..

we get 16 VCPU's

out of 32 GB usable is 24 GB for yarn containers

out of 16 we can use 12 cores for yarn containers

24 GB RAM , 12 VCPU's on each worker node

5 such worker nodes..

executor memory - 1 to 4 GB

executor cores - 1 or 2

maximum executors you can create in entire cluster.

12 executors in each worker node

12 * 5 = 60 executors/containers

60 executors with each holding 1 cpu cores 2 GB Ram

30 executors with each holding 2 cpu cores 4 GB Ram

5 worker nodes

24 GB usable Ram

12 VCPU's

in our resource pool which YARN is managing we have

120 GB RAM and 60 CPU Cores.

24 * 5 = 120 GB RAM

12 * 5 = 60 cpu cores

Spark Optimization Session - 5
================================

bigLogNew.txt - 1.46 GB

I want to move this file to my edge node in the cluster..

5 Datanodes

1 edgenode which is alloted to us and we have access to that.

step 1: move this file bigLogNew.txt to the edge node.

step 2: move this file bigLogNew.txt from edge node to the hdfs using put command.

step 3: we will be doing some operations..

to transfer file from my local to a external server (edge node)

scp (secure copy)

```
scp Desktop/bigLogNew.txt.zip bigdatabysumit@gw02.itversity.com:/home/bigdatabysumit
```

```
scp <local_file_path> <username@hostname:<path in server>>
```

```
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
```

we are operating in local mode thats why the executor and driver are same.

1 container is allocated which is on local machine..

master = local[*]

master = yarn

```
spark2-shell --master yarn
```

in yarn mode

we can see 1 driver & 2 executors.

There are 2 ways to allocate the resources

1. allocating the resources manually

2. dynamic resource allocation

initial executors - 2

max executors - 10

min executors - 0

executorIdleTimeout - 120 seconds

executor - cpu + memory

each executor by default will get 1 cpu core and 1 gb RAM based on the default configurations of this cluster.

whenever we allocate memory to the container

300 mb out of that goes for some overheads...

1024 - 300 = 724 mb

1. storage & execution memory.. - 724 * .6 = 434.4

2. additional buffer memory for other purpose - 724 * .4

Spark Optimization Session - 6
================================

our big file is in hdfs

and we will be processing that

we will go with dynamic resource allocation

bigLogLatest.txt

```
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
```

```
spark2-shell --master yarn
```

```
val rdd1 = sc.textFile("bigLogLatest.txt")

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

val rdd3 = rdd2.groupByKey

val rdd4 = rdd3.map(x => (x._1, x._2.size))

rdd4.collect
```

ERROR,{Thu Jun 04 10:37:51 BST 2015 , Thu Dec 10 10:37:51 GMT 2015, Fri Dec 26 10:37:51 GMT 2014}

(ERROR, 3)

WARN, {Sun Nov 06 10:37:51 GMT 2016 , Mon Aug 29 10:37:51 BST 2016}

since we have called one action we can see one job in spark UI

we can see 2 stages because we have one wide transformation.

Tasks - 132

total blocks in hdfs are 66

we have 2 stages..

we have 2 stages each with 66 tasks..

we have 66 partitions in our rdd thats why 66 tasks are created per stage.

Spark Optimization Session - 7
================================

Tasks - 132

2 stages..

rdd has 66 paritions because our hdfs file has 66 blocks..

66 executors..

we can have at the max 10 executors..

10 executors have to execute 66 tasks

some of the executors might have to execute multiple tasks one after the other...

if in executor there is 1 cpu core - then only one task can be performed at a time

if in a executor there are 4 cpu cores - then this executor can parallely run 4 tasks at a time..

number of executors * number of cpu cores each executor hold

10 * 1 = 10

10 tasks can be executed at the same time...

in stage 0 we have 66 tasks and each task did the same amount of work.

each of the partiton was have equal amount of data.

after we used groupByKey shuffling happened..

ERROR
WARN

after shuffling we will get maximum 2 full partitions.

8.5 gb

4.25 gb error 1 gb

4.25 gb warn   7.5 gb

one of the partition will definitely get 4.25 gb or more...

66 new patitions which will be created in stage 2

only 2 of them will hold data..

rest all of them will be empty..

you will have 66 task in stage 2...

entire data will be executed in just 2 tasks..

remaining 64 tasks wont do anything..

1 cpu core and 1 gb RAM - 400 mb only...

4.25 gb data will go to one partition

one executor should handle one partition

executor can handle 400 mb

but you want this executor to handle atleast 4.25 gb

salting..

warn1
warn2
warn3
warn10

error1
error2
error3
error10

20 distinct keys..

20 filled partitions.. - 20 executors...

8 gb..

8 gb / 20

400 mb

Spark Optimization Session - 8
================================

1. static resource allocation

2. dynamic resource allocation

initial executors - 2

max executors - 10

min executors - 0

executorIdleTimeout - 120 seconds

dynamic resource allocation can be really helpful with long running jobs..

10 stages..
=============

50 containers in static mode..

100 tasks ..

20 tasks

10 tasks

10 tasks

10 tasks

10 tasks..

static resource allocation
============================

```
spark2-shell --conf spark.dynamicAllocation.enabled=false --master yarn --num-executors 4
--executor-cores 2 --executor-memory 2G
```

Spark Optimization Session - 9
================================

```
spark2-shell --master yarn --num-executors 5 --executor-cores 2 --executor-memory 2G
```

initial executors - 5

max executors - 10

min executors - 0

executorIdleTimeout - 120 seconds

executor-cores 2

executor-memory 2G

note: if you have a long running job then its better to go with dynamic resource allocation..

```
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
```

8.5 GB

66 blocks in hdfs

our base rdd has 66 partitions..

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

groupByKey

we have a rdd with 66 partitions but only 2 were having data rest 64 were empty..

2 partitions have to hold 8.5 gb in total..

1 gb 7.5 gb

one option is increase the container size so that it can handle 16 gb

Is there a way we can generate a random number between some range lets say (1 to 40)

```
ERROR1
ERROR7
ERROR40
ERROR1

WARN1
WARN2
```

WARN1

ERROR,{TIMESTAMP1 , TIMESTAMP2, ...} ERROR1,3
ERROR, {......} ERROR7,5
ERROR, {......} ERROR40,7

ERROR, 15

ERROR1,3

ERROR,3

ERROR7,5

ERROR,5

WARN1,10

WARN,10

if each executor has to hold more than its capacity it will give out of memory error..

4.5 gb

if we give 4 gb to a container

2.5 gb

it will lead to out of memory error.

error and warn..

salting and added a random number between a particular range.. this is to ensure we get more unique keys..

120 keys..

8.5 gb data in 120 keys...

8000 mb / 120

if we have 66 partitions in stage1 we will have 66 partitions in stage2 after groupByKey

66 partitions
4 containers

```
val random = new scala.util.Random
val start = 1
val end = 60

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

val rdd2 = rdd1.map(x => {
var num = start + random.nextInt( (end - start) + 1 )
(x.split(":")(0) + num,x.split(":")(1))
})

val rdd3 = rdd2.groupByKey

val rdd4 = rdd3.map(x => (x._1 , x._2.size))

rdd4.cache

val rdd5 = rdd4.map(x => {
if(x._1.substring(0,4)=="WARN")
("WARN",x._2)
else
("ERROR",x._2)
})

val rdd6 = rdd5.reduceByKey(_+_)

rdd6.collect.foreach(println)
```

Spark Optimization Session - 10
==================================

Memory usage in spark falls under two broad categories..

1. Execution memory - memory required for computations in shuffle, join, sorts, aggregations..

2. Storage memory - is used of cache.

In spark Execution memory and storage memory share a common region..

When no execution is happening.. then your storage can aquire all the available memory and vice versa..

Execution may evict storage if necessary..

2 gb common unified region

2 gb for storage..

but this eviction can happen until total storage memory usage falls unser a certain threshold...

2 gb if all 2 gb is used for storage..

now if some execution/computations are coming they cannot evict the entire 2 gb..

there is a certain threshold beyond which the execution cannot evict the storage..

execution can evict storage upto a certain threshold.

but storage can not evict execution..

This design ensures several desirable properties:

1. application which do not use caching can use entire space for execution.

2. applications that do not use caching can reserve a minimum storage space..

this makes your data blocks immune from being evicted..

--executor-memory 4g

if you request a container/executor of 4 GB size..

then you are actually requesting

4 GB (heap memory)

+

max (384 mb, 10% of 4 GB) (off heap memory) - overhead

4096 mb (java heap)
+
384 mb (off heap) - VM overheads

out of the 4 GB (total heap memory)

300 mb is again reserved

4 gb - 300 mb = 3.7 gb

so we are now left with 3.7 GB

out of this 3.7 GB 60 % of it goes to the unified (storage + execution memory)

2.3 GB is for unified region (storage + execution)

remaining 40% of 3.7 GB goes to user memory... - 1.4 GB

(to hold user datastructures, storing spark related metadata and safeguarding OOM errors..)

we requested for 4 GB container

4 GB Heap memory

384 mb of off-heap memory

The heap memory is sub divided..

out of 4 gb heap memory 300 mb is reserved.

3.7 gb..

60% is for (storage & execution) - 2.3 GB

out of 2.3 GB 50% is the threshold for storage memory..

this means we can cache data upto 1.15 GB roughly without worrying about eviction by executions/computations.

40% is for user memory..

3 GB for java heap

300 mb is reserved..

2.7 GB

2.7 * .6 (storage & execution unified region)

1.5 GB

Spark Optimization Session - 11
==================================

cache & persist
================

the size of file was 8.2 gb on disk (serialized form it is 8.2 gb)

when we talk about memory data is kept in deserialized form

20 gb in deserialized form in memory

SERIALIZED MEANS IN BYTES FORM - AND IT TAKES A BIT OF TIME BECAUSE IT HAS TO BE DESERIALIZED..

DESERIALIZED IS FAST BUT TAKES MORE SPACE..

cache and persist.

in terms of serializer
=======================

we should prefer kryo serializer over java serializer..

whenever the data is stored on disk or has to be transferred over the disk it has to be in serialized form..

if we use kryo serializer then size will be much lesser than in the case of java serializer.

5 gb

2 gb

kryo is significantly faster and more compact than java serialization (often as much as 10 times..)