SPARK OPTIMIZATION NOTES

SPARK OPTIMIZATIONS

Monday, August 23, 2021 3:21 PM

Spark performance Optimizations / Tuning spark jobs

There are mainly 2 areas of focus:

1. **Cluster configuration** (resource level configuration) : It deals with how the resources were allocated to spark jobs and how to make sure they get enough resources to run and process data(allocate executors, ram , cores)
2. **Application code level**: How the code written can be optimized so that we can use less resources and get max gain out of it.
   −−>Partitioning, bucketing, cachefpersist, join optimizations, optimized file formats, reducefavoid shuffling

How to make sure our spark job gets right amount of resources?

Session 01:
Node/Machine (are the same)
Executor/Container/JVM (RAM+CPU CORES) (are the same)
−−>Let's take a below scenario 10 node cluster with each node holding 16 cores of 64 gb ram
**each executor can have max of 16 cores with 64gb ram (in this example)

What are the different types of executors:

1. **Thin Executor**: To create more no. of executors with each executor holding minimum possible resources.
   More no. of executors and less amt of resources
   Ex. 16 executors => 16 cores, 64gb ram
   Ie 1 executor=> 1 core, 4gb ram

Drawbacks:
−−>**Multi threading not possible** as each executor has only 1 core.
−−>**Many copies of broadcast variables** required (as we have many executors)

2. **Fat executor**: To provide max resources to each executor
   Less no. of exectors and more amt of resources
   −−>each executor has 16 cores with 64gb ram

Drawbacks:
−−>Too much of multi threading makes system slow
−−>Time consuming for getting data frm hsfs for all cores (more than 5 cores for each executor is not advisable)
−−>**Time consuming for garbage collection**

Garbage collection : To remove unused objects from memory

Session 03:
How the resources are distributed among clusters in spark (example) ?

3. **Balanced Executor**: Due to above limitations we have a more balanced executor to process the data in spark(each executor has max of 5 cores)

Scenario: Total 10 nodes, each node having 16 cores and 64 gb ram
−−>Here 1 core and 1gb ram is used for maintaining background processes in each node.
−−>1 node => 3 executors => 15 cores , 63 gb ram
−−>each executor => 5 cores, 21 gb ram
−−>we have 7% ram given to off−heap memory (~1.6gb ram)
−−>ie each executor => 5 cores, 19.4 gb ram
−−>For 10 node cluster => 30(10*3) executors => 150(15*10) cores, ~592(3*10*19.4) gb ram
−−>out of this, 1 executor given to YARN.
Total : 10 nodes => 29 executors => 147 cores, ~583.6 gb ram

What is off heap memory and what is it's use?
Off-heap memory : It is memory block outside containers which is used as raw storage unit especially for optimizations.
−−> **Data(objects) stored here are easy to be cleaned** rather than using garbage collector which takes time
−−>It can be used as **java overheads**

What is an overhead?
In general, it's resources (most often memory and CPU time) that are used, which do not contribute directly to the intended result, but are required by the technology or method that is being used.
Ex.
It's like when you need to go somewhere, you might need a car. But, it would be a lot of overhead to get a car to drive down the street, so you might want to walk. However, the overhead would be worth it if you were going across the country.
In computer science, sometimes we use cars to go down the street because we don't have a better way, or it's not worth our time to "learn how to walk"

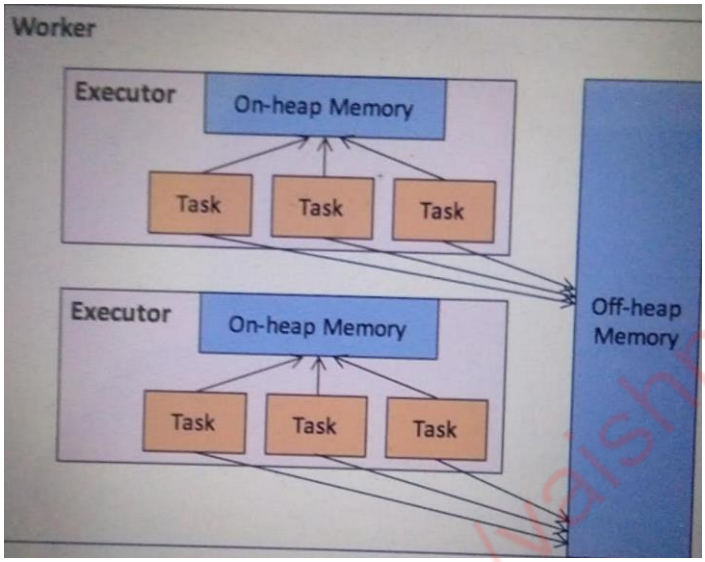
Session 04:

How the resources are distributed in a cluster in spark (example frm ITVersity)?
−−>Total no. of nodes: 5
−−>RAM and VCPU's (cores) in each node : 24gb ram, 12 cores
−−>Total ram : 120 (24*5)
−−>Total cores: 60 (12*5)
−−>Executor memory range: 1 to 4 gb
−−>Executor cores: 1 to 2 cores
−−>Max executors in a node: 12 executors ==> Total 60(12*5) executors, 12(12*1) gb ram, 12(12*1) Vcpu's [Max executors min resources, 1 gb ram, 1 core]
−−>Min executors in a node: 6 executors==>30(6*5) executors, 24(6*4) gb ram, 12(6*2) vcpu's [Min executors max resources , 4gb ram, 2 cores]

Session 06:

We will look at practical in IT Versity Lab on how the resource configuration has been done?
−−>**Few of the properties to look out for :**

1. executorIdleTimeOut: 120 sec. It means that if executor is idle for 120 s it will be remove
2. Spark2−shell −−master yarn : To run the cluster on yarn (executor and driver will be on cluster)
3. Off−heap memory − It occupies ~(10 −30)% of memory allocated to executor
   Ie 1gb ram => 1024 − 300 = 724 (actual memory allocated)
4. Storage & execution memory : (60% of above) = 724*0.6 = 434.4
5. Additional buffer memory : (40% of above) = 724*0.4 = 289.6



What are the different types of resource allocation?
−−>**Static Allocation** : The amount of resources will be fixed and the same till the job ends (executors, ram, cpu cores)
−−>**Dynamic Allocation**: Spark takes care of resources allocated to its jobs and removes them as well once the tasks get finished and they get into idle state

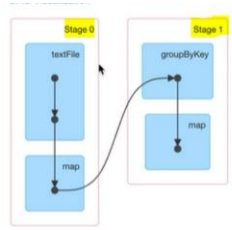
Static Allocation	Dynamic Allocation
1. Resources will be fixed at the start 2. Resources won't be released even if jobs don't need them 3. Resources go un−used mostly	1. Resources will be allocated dynamically based on requirement 2. Resources will be released if they are no longer used 3. Resources don't go un−used

Session 07:
We see an example of using "**groupByKey**" on a **big file** and how resources are allocated for it.
−−>File size : 8.2GB
−−>1 job triggered => 1 action called
−−>2 stages in DAG => 1 wide transformation called
−−>No. of hdfs partitions => (8.2*1024)∫128 = ~67 (each partition is of size 128mb)
−−>No. of total tasks = 67*2 = 134 (as we have 2 stages)

Stage 01:
−−>data is almost equally distributed among the 66 tasks
−−>all the tasks have written the output to disc (serialized format) which is then taken up for stage 2

Stage 02:

−−>From input file, we have to find "count of ERROR and WARN keywords"
−−>so we have at the max 2 tasks to handle this
−−>ie 8.2 GB file => 2 tasks (which is very heavy for them to handle)
−−>Out of 67 tasks we have only 2 tasks will be working
−−>65 tasks won't perform anything
−−>Due to which the executor would fail to handle the tasks and job will not get finished



Active Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	collect at <console>:26 +details (kill)	2020/08/02 19:34:27	1.3 min	64/66 (2 running)			796.1 MB	

Completed Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
0	map at <console>:25 +details	2020/08/02 19:33:15	1.2 min	66/66	8.2 GB			1855.2 MB

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succe
0	collect at <console>:26 collect at <console>:26 (kill)	2020/08/02 19:33:15	3.4 min	1/2	119/132

What is an edge node ?
Edge node is also called as **gateway node**. It's a node which can be accessed by client instead of name node.

How to deal with out of memory / groupByKey problem?
−−>**Salting** : We create more keywords in order to make all the tasks work equally rather than giving the whole work to few tasks.
From our example,
−−>**I/P** : ERROR, WARN **O/P** : ERROR1, ERROR2….ERROR10 , WARN1, WARN2……WARN10
−−> so that we can involve many tasks to work on it

Steps Involved:

1. To generate a random no. from 1 to 60
2. Load the data file
3. To map the input and append a random no. to it
   IfP : (ERROR, T1), (ERROR,T2)
   OfP: (ERROR1, T1), (ERROR10, T2)
4. Grouping all similar keys together
   IfP: (ERROR1, T1), (ERROR10, T2) , (ERROR1, T5)
   OfP : (ERROR1, (T1,T5,T0.....)) , (ERROR10, (T6, T8))
5. To find the size of time stamps which actually give the count of each ERROR AND WARN
   IfP : (ERROR1, (T1,T5,T0.....)) , (ERROR10, (T6, T8))
   OfP : (ERROR1, 25) , (ERROR10, 45)
6. Now, removing the appended random numbers to make final keywords as only ERROR AND WARN
7. Now, adding all the values to get the final count
   OfP: (ERROR, 74932) , (WARN, 7323)

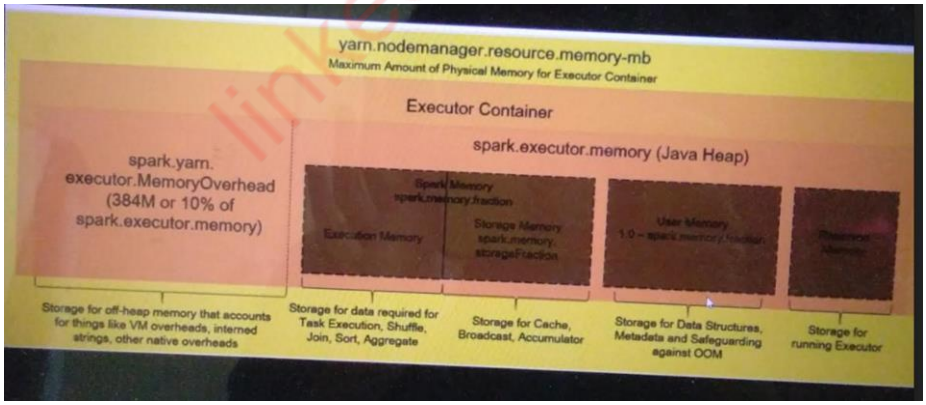
```
val random = new scala.util.Random  1
val start = 1
val end = 60

val rdd1 = sc.textFile("bigLogLatest.txt") 2

val rdd2 = rdd1.map(x => {     3
var num = start + random.nextInt( (end - start) + 1 )
(x.split(":")(0) + num,x.split(":")(1))
})

val rdd3 = rdd2.groupByKey  4

val rdd4 = rdd3.map(x => (x._1 , x._2.size))   5

val rdd5 = rdd4.map(x => {      6
if(x._1.substring(0,4)=="WARN")
("WARN",x._2)
else
("ERROR",x._2)
})

val rdd6 = rdd5.reduceByKey( + )  7
```

Execution Memory	Storage Memory
1. It is used to perform computations like shuffle, sort, join etc **2.** It cannot be evicted by storage memory **3.** It shares the memory with storage memory togethor	**1. It is used to store cache** **2.** It can be evicted by execution memory if it crosses its threshold **3.** If execution memory is not using the memory then it is taken up by the storage

-->**User memory** : It is used to store user data structures, meta data etc
-->**Reserved memory** : It is used to reserve memory to run executors

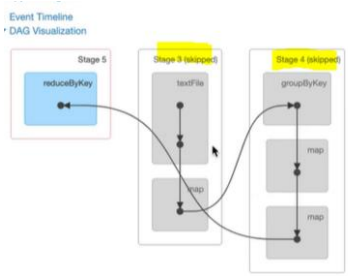
Let's take a practical example:
-->we have request to create a executor of 3GB. Following is how the memory gets divided

1. 3GB RAM => on heap memory
   +
   384 mb or 10% RAM => off heap memory
2. Reserved memory : ~10% (on heap memory) = 300mb
3. Remaining = (3072−300) =~2.7GB (execution memory ,storage memory + user memory)
4. Execution + storage memory = 60% of 2.7GB = ~1.6 GB
5. User memory = 40% of 2.7GB =~0.98 GB

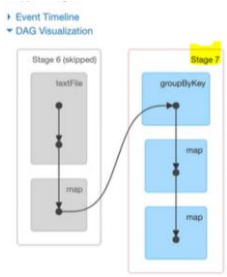


Practical example on cache() and persist() methods:
Spark optimization:

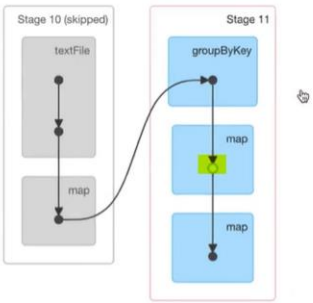
1. When we call an action at the end after the entire prgm is executed due to internal optimization of spark, spark stores the result and prints it directly and skips the execution of previous stages.



2. When we call an action in the middle of a stage, spark has to recompute that stage every time which is time consuming. Stage 7 will be recomputed every time an action is called in the middle of its stage.



3. When we cache an rdd, computation starts from the cache rdd only



Completed Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Outp
11	collect at <console>:26 +details	2020/08/05 18:49:07	25 s	66/66		

Skipped Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output
10	map at <console>:31 +details	Unknown	Unknown	0/66		

−−>cache stored in "storage level" for a program

Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory
4	MapPartitionsRDD	Memory Deserialized 1x Replicated	66	100%	12.9 KB

−−>**when we cache a big data file and store it in-memory**. We observe that only 42% of it is cache. Size of file also increases as it in memory (deserialized manner)
−−>storage level cannot be given entire memory as execution memory also uses it.

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
8	bigLogLatest.txt	Memory Deserialized 1x Replicated	28	42%	10.8 GB	0.0 B

−−>**when we cache a big data file and store it in DISK**

	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
	Disk Serialized 1x Replicated	66	100%	0.0 B	8.2 GB

−−>**when we cache a big data file and store it in DISK and MEMORY**

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
8	bigLogLatest.txt	Disk Serialized 1x Replicated	66	100%	8.3 GB	5.5 GB

Which serializer to choose ?

Kryo Serializer	Java Serializer
1. It occupies less space when data has to be transferred over diskf stored in disk in serialized format. 2. It is said to be 10 times faster then java serializer	1. It occupies more space

****Note**

1. Node = machine
2. Cluster = multiple nodesfmachines
3. Container = Executor = JVM
4. No. of tasks= No. of cores =No. of hdfs partitions
5. Total tasks= Total cores = (No. of hdfs partitions * no. of stages)
6. Order of hierrarchy : Node =>Executor(ram + cpu cores) =>task (hdfs partition)
7. No. of tasks running parallely => (no. of executors)*(no. of cpu cores)
8. Increase parallelism => Increase resource (executor, cpu cores, ram) => reduce process time
9. RAM distribution => off heap memory + on heap memory ((execution memory + storage memory) + user memory + reserved memory)

KEYWORDS:

1. Spark optimization:

a. Cluster Configuration : To configure resources to the cluster so that spark jobs can process well.

b. Code configuration: To apply optimization techniques at code level so that processing will be fast.

1. Thin executor: More no. of executors with less no. of resources. Multithreading not possible, too many broadcast variables required. Ex. 1 executor with each 2 cpu cores, 1 GB ram.
2. Fat executor: Less no. of executors with more amount of resources. System performance drops down, garbage collection takes time. Ex 1 executor 16 cpu cores, 32 GB ram.
3. Garbage collection: To remove unused objects from memory.
4. Off heap memory: Memory stored outside of executorsf jvm. It takes less time to clean objects than garbage collector, used for java overheads (extra memory which directly doesn't add to performance but required by system to carry out its operation)
5. Static allocation: Resources are fixed at first and will remain the same till the job ends.
6. Dynamic Allocation: Resources are allocated dynamically based on the job requirement and released during job stages if they are no longer required.
7. Edge node: It is also called as gateway node which is can be accessed by client to enter into hadoop cluster and access name node.
8. How to increase parallelism :

a. Salting : To increase no. of distinct keys so that work can be distributed across many tasks which in turn increase parallelism.

1. Execution memory : To perform computations like shuffle, sort, join
2. Storage memory : To store the cache
3. User memory : To store user's data structures, meta data etc.
4. Reserved memory : To run the executors
5. Kyro Serializer: Used to store the data in disk in serialized manner which occupies less space.