Systems for Data-Science - Milestone 2
Group 04

Fares Ahmed, Paul Mansat, Tian Xiangyu, Nicolas Zimmermann, and Xiaoyan Zou

School of Computer and Communication Sciences

Spring 2020

Application 1

Problem Identification

App 1 reads a csv file containing 3 attributes, collects and sums the last attribute of same users by using groupBy on the second attribute.
First, App 1 converts the data into an RDD, that is further processed by creating tuples with corresponding values (modified or not). Secondly, the data is collected to the driver and grouped by the first value (i.e key) of the tuple. Values sharing the same key are summed together. Finally, base on the pair (key, new summed values), App 1 returns a map named t2 and print out all its pairs.
The log file indicates that the original code has ExecutorLostFailure error which is due to OutOfMemoryError(OOM): "Exception in thread thread_name: java.lang.OutOfMemoryError: Java heap space" for "failed: collect at App1.scala:23".
Shuffle file is lost and container is killed on request then it causes ExecutorLostFailure. There is not enough space for the executor. Namely, the task is too heavy for an executor in terms of memory required. Since we cannot modify configuration (i.e. increase executor/driver memory). We frist tried to create more partitions in order to reduce the partition size by re-partition.
However, the OOM error was raised again: we are still collecting a large amount of data into the driver that makes it crash. The original codes collect first all processed data and combines data together according to the second attribute of the csv file. This raises error when the data is huge.

Solution

Initially We solved this problem by delaying the collection of the data until the very end of the processing of the data. By doing so, less data is being transferred to the driver. Nevertheless, this still consumes much time. Finally, our solution consists of a more efficient way to accomplish the task. That is to reduce the results elements before calling "collect". We synthesize operations by reduceByKey, which replaces groupBy and map-sum operation. In this manner, data transferred to the driver are largely reduced and the program does not yield to an error.

Application 2

Problem Identification

App 2 is very similar to App 1 except that when codes call textFile method, the optional parameter minPartitions is set to 200.
App 2 is doing exactly the same operation as App 1. For further information refer to the documentation in Application 1.

When one uses the textFile method, the data will be divided into partitions. The default size of a block in HDFS is 128 MB. The large dataset has size around 1.4 GB which is divided into around 12 partitions. By specifying minPartitions in textfile method, we have 200 partitions in App 2. Having more partitions might reduce the runtime with more executors. However, this does not reduce the amount of data transferred over the network since the code uses a groupBy. Exception messages for App 2 are: "java.lang.OutOfMemoryError: GC overhead limit exceeded", "java.lang.OutOfMemoryError: Java heap space" and "BlockManagerMasterEndpoint: No more replicas available for taskresult_xx", where xx is the task number.
The driver is running out of memory due to "collect at App2.scala:23". We collect large amount of data from workers and try to fit it into a single driver node.

Solution

Since we cannot modify the configuration(i.e. increase executor memory), we can reduce the executor workload by reducing the amount of data being transferred over the network. Our solution replaces groupBy and map by reduceByKey in order to reduce the amount of data being transferred to the driver. Since reduceByKey reduces to minimal amount of results, we can "collect" right after. The summation part in the last "map" function is already done by reduceByKey.

Application 3

Problem Identification

Application 2 creates a list of tuple (key: Int, data: Int), computes the modulo of the data from 2 to 100, groups elements by key and flatten the resulting data by summing everything together.
The way the latter operation is done is by creating a huge data-set of (key: Int, Data % N), where N $\in [2, 100]$. For each key and modulo N, an entry in the data-set is created.
One can see that the creation of such a large data-set is a waste of space: in the end the data is summed up.

Solution

The solution is then to avoid the creation of such a large data-set by doing operations (i.e summing and computation of modulo) on-the-fly.
We first group all elements according to their key, sum-up their respective data together and use the foldleft operator to compute the modulo between 2 and 100 of the summed data.

Note that we used the associative property of the modular operator (i.e $(a + b) \mod n = a \mod n + b \mod n$).

Application 4

Problem Identification

App 4 is very similar to App 2 except that App 4 has an additional configuration. The driver memory is set to 10G, which is 1G by default (as in App 1 and App 2).
App 4 is doing exactly the same operations as in App 1 and App 2. For further information, please refer our documentation in Application 1.
In App 2, we did not have enough memory in the driver to get all computed results. Here we have more spaces in the driver by spark.driver.memory configuration. However, this time, we have another error due to the raise of driver memory. The error is "Job aborted due to stage failure: Total size of serialized results of 102 tasks (1026.5 MB) is bigger than spark.driver.maxResultSize (1024.0 MB)", due to collect at App4.scala:24. When we increase the driver memory, we better adapt the corresponding driver.maxResultSize. When the executor tried to send its result to the driver, it surpasses the driver.maxResultSize. The big trunk of result data in executors causes the problem.

Solution

The solution is to reduce the amount of results (so reduce the size of results) then send them to the driver. This is done by replacing groupBy and map by reduceByKey. We combine some elements inside each partition by the key value so that the elements being transferred over the network are largely reduced. Then we sum the corresponding values together. Hence, the size of final results being collected to the driver is minimal.

Application 5

Problem Identification

Application 5 is quiet similar to App 3. It takes as input a data-set with element (key: Int, element: Int). From this data-set, it creates an iterator (key: Int, element % N) for $N \in [2, 50]$. App 5 then groups all elements with respect to their key and sums each element that have same key. As one can see, the intermediate iterator created will be unnecessarily large (w.r.t to the number of entries), when in the end, we use this data-set to group all elements by and sum their associated data. In fact, the size of the latter iterator is the reason why App 5 crashed when ran on the large data-set.

Solution

Naturally, the solution is to reduce the size of the latter described iterator. One might first note that if all elements are grouped by key before any transformation is made to the elements, we will reduce the number of entries of the iterator. Indeed, with an early grouping, we avoid having multiple time the same key in the iterator. Hence, the first thing we do is to group by key the input.
The latter solution is far from being enough: we still create an unnecessarily large iterator when in the end all elements of the iterator will be summed up (for each entry of input we create 48 new entries in the intermediate iterator). Ideally we would like to compute the sum on the fly (i.e while processing the data). This requires to move the summing to the earliest possible location. To do so, we use a the map operator entangled with a fold-left:

- The map covers all elements associated to a key and associates to each key the it sum of modular element
- The second inner fold-left is used to compute the 48 modulus of each elements.

This structure allows us to compute the sum of each elements on the fly and avoid the creation of the latter described large iterator.
We summarized with the help of a small schema how we used the two fold-left on Figure 1.

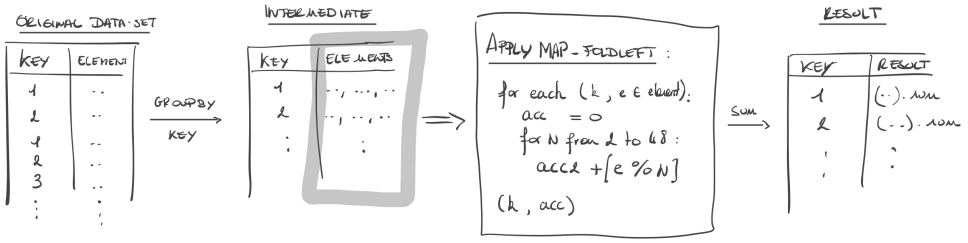

Figure 1: Summary of operations used for App 5

Application 6

Problem Identification

In this app, we are grouping by $(c0, c1)$ and summing $c2$ in each group. However here we first group by $c0$ and then by $c1$, resulting in the second operation done locally on the different executor and not as a RDD operation anymore. Hence the load was not well distributed between the various

executors after the first grouping which leads to a crash (lack of memory in one or more of the executors) with the second grouping.

Solution

Instead we directly group by $(c0, c1)$ (using map to $((c0, c1), c(2))$ and group by key), making the whole operation on the RDD itself. With this way of grouping the data, it is directly well formatted for summing $c(2)$ and printing, allowing for simpler and more readable code.

Application 7

Problem Identification

App7 is coded to join data from two CSV files of different sizes and after that the joined data was filtered and grouped on conditions to achieve required pair RDDs. The original App7 could run over 5 minutes and aborted its job due to stage failure. According to the log file of outputs as shown in the figure below, it can be specified that the executor for groupBy at App7.scala:25 could run a long time and lost because the container is killed by request. As shown in the initialization, the AM container is allocated with 1408 MB memory including 384 MB overhead. It can also be found that the bigger data from the first CSV file contains inputs of 1337.7 MB, which is far larger than the other with the size of only 77 B. It is also indicated by "Shuffle Write" and "Shuffle Read" that the shuffle operation to find same keys for the large data set consumed most of the time and triggered the failure as it needs a large number of bytes and records written to disk to be read by the shuffle ,therefore problems are located on huge demand of shuffle for skewed data sets.

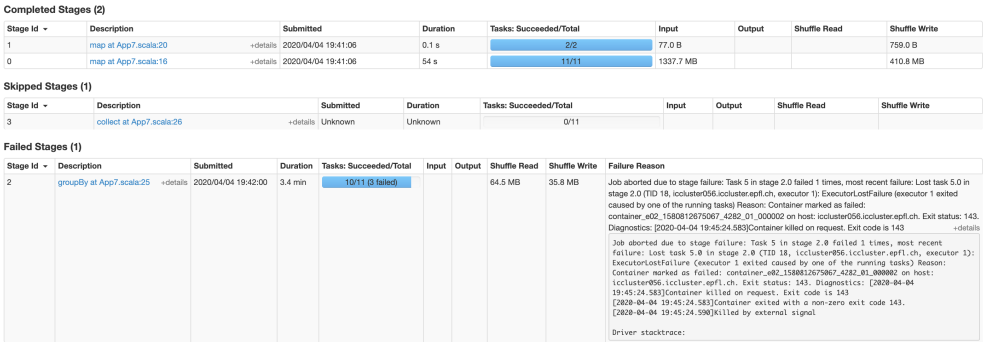

Figure 2: Log file for output of App7 original version

Solution

To avoid overload of shuffle and achieve the same goal of joining efficiently under the condition of unchanged initialization of system capabilities, broadcasting and map-side join can be used. Firstly, the smaller data set, namely rdd2, could be collected and broadcast into drivers, and then the large data set, namely rdd1, is operated and mapped only based on the condition that the same keys can be found in the broadcast data, which will extremely decrease the time consumption on shuffle. Besides, the methods of mapPartition and reduceByKey are used. The mapPartition which replaces the map could operate data on a large scale of partitions. The reduceByKey that replaces groupByKey could combine values based on the same key locally on nodes before, and then transfer pre-aggregated pairs of keys and values for the global combination. By implementing those methods mentioned above, it enables to take roughly 30 seconds for running to obtain outputs, which gains high optimization compared with the original version of App7.

Application 8

Problem Identification

App 8 does the following :

- Extracts (key, value) pairs from a given file
- Group those pairs by keys and sum their values to (key, sum of values) pairs
- Extract the maximum "sum-value" and divide it by two (let's call it **HALF_MAX**)
- Prints out :
  - The maximum value smaller than **HALF_MAX**
  - The minimum value larger than **HALF_MAX**

The problem in this app was that a large majority of the data was shuffled to one of the executors, causing it to run out of memory and forcing it to spill to disk during the first **reduce** operation. This executor would then throw an OutOfMemoryError because it would exceed the Garbage Collection overhead limit as the executor spends most of its time spilling/recovering data from disk than actual computation. This was identified with the following error message from the driver

```
20/04/04 13:39:22 ERROR YarnClusterScheduler: Lost
    executor 2 on iccluster057.iccluster.epfl.ch:
    Container marked as failed:
    container_e02_1580812675067_3981_01_000003 on host:
     iccluster057.iccluster.epfl.ch. Exit status: 52.
    Diagnostics: [2020-04-04 13:39:22.302]Exception
    from container-launch.
Container id:
    container_e02_1580812675067_3981_01_000003
Exit code: 52
```

as well as those messages from that executor, which correspond to the last spill to disk before hitting the GC overhead limit as well as the exception throw

```
20/04/04 13:37:52 INFO ExternalAppendOnlyMap: Thread
    64 spilling in-memory map of 413.2 MB to disk (5
    times so far)

Exception: java.lang.OutOfMemoryError thrown from the
    UncaughtExceptionHandler in thread "SIGTERM handler
    "
20/04/04 13:39:21 ERROR SparkUncaughtExceptionHandler:
     Uncaught exception in thread Thread[Executor task
    launch worker for task 10,5,main]
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

Solution

Since the problem was that the default partitioning strategy sent most of the data to one of the executors, the fix was to define a custom range partitioner to balance the data more evenly between the executors.