Performance of Costly Functions

Linnea May

Lazy Evaluation:

All RDDs and data frames are lazily evaluated. This means that any operation on the data, including reading it in the first place, doesn't occur until the result of all of the operations is requested by the user. Instead, each operation is added to the DAG. This allows Spark to make appropriate decisions all at once and saves time and computation power. The RDD will only be evaluated when we want the resultant RDD and so is only evaluated when we perform an action on the RDD. A list of actions can be found here but common actions include count(), collect(), and take(). In short, the RDD is only evaluated once you ask for the final version of it.

Cacheing:

The ability to cache RDDs can save noticeable amounts of times, and should usually align with actions performed on the RDD or when using iterative algorithms, for example anything used in machine learning. The process of caching an RDD involves simply calling the cache() or persist() method on the RDD. Either method works, but have varying options.

Cacheing is the most common method of storing an object into memory only and does not take any parameters. It is technically identical to calling persist() with the StorageLevel parameter set to MEMORY_ONLY, but speed tests have shown that persisting to memory seems to be faster than its cacheing counterpart.

Perisisting:

The persist method is more flexible in its application as it allows for different locations of storage of the RDD. The following StorageLevel options are available:

- MEMORY_ONLY: The default parameter for persist. It should be equivalent to cacheing, but in practice persisting to memory only seems to be faster. Persisting to memory is undoubtedly the fastest method, assuming each executor has the memory available to do so. If the memory capacities are exceeded, data in the RDD will overflow and be lost to the user.
- MEMORY_AND_DISK: Passing this parameter is your best bet if there's a chance your executors will run out of memory. Any data that would be lost to overflow will instead be stored to disk. It's still faster than re-evaluating the RDD before an action, but the speed of persisting to memory only may not be worth it if there's a risk of losing data.
- DISK_ONLY: This usually isn't a recommended option unless the dataset in use is especially large, because it would usually just be faster to re-read the RDD from disk as opposed to going through the redundancy of also persisting it there.

- Appending _SER to any parameter will serialize the data before it persists. This provides less memory strain, reducing overflow errors, but at the cost of elevated processing time as the data has to be unserialized before evaluated the RDD. That being said, the speed of accessing the serialized data appears to overtake the time it would take to actually unserialize the data, especially using the Kyros serializer, making specifically serliazed to memory the most efficient option.
- Appending a number (such as MEMORY_ONLY_2) increases the number of replicas that will be created on other nodes. This guards against node failure, but does increase the memory or storage demands on each node. Any RDDs lost in memory can be recomputed, but if pressed for time then running tasks on a recovery partition of the RDD can be faster than recomputing it.

The benefits of different storage levels can be summarized as:

Level	Space Usage	CPU Usage	In Memory
MEMORY_ONLY	High	Low	Y
MEMORY_ONLY_SER	Low	High	Y
MEMORY_AND_DISK	High	Medium	Some
MEMORY_AND_DISK_SER	Low	High	Some
DISK_ONLY	Low	High	N

One easy way to improve computation speed when persisting is found in the spark configurations. Setting spark.sql.inMemoryColumnarStorage.compressed=True while launching a spark environment, or more permanent in the spark-defaults.conf file (found in $SPARK_HOME/conf) allows Spark to choose the optimal compression method for each individual column in the dataset. The benefits of this configuration have only been tested on CSV and JSON files. It may be fruitful to test costly functions with this enabled when working with a parquet file due to their columnar nature.

Heartbeat Error:

As we began testing costly functions on the server, the first error we encountered was all of the executors in the cluster timing out and being killed by the driver. The code guilty of causing these errors consisted of creating a new dataframe that joined two others and dropped a couple columns:

```
_yyy = review.join(bus, review.business_id == bus.business_id,
'inner').drop(bus.business_id).drop(bus.stars)
```

Our original configuration when testing on the server:

```
spark = SparkSession \
       .builder \
       .master("yarn") \
       .appName("testing") \
       .config("spark.executor.instances", "70") \
       .config("spark.executor.memory","4g") \
       .config("spark.driver.memory","30g") \
       .config("spark.executor.cores",'1') \
       .config("spark.scheduler.mode","FIFO") \
       .getOrCreate()
```

The procedure we went through to solve this error is logged in five separate files on the server and represents the different methods we used to attempt to solve the problem.

1. Decrease the number of executors and/or partitions.
Our first instinct was to start lowering numbers, and we started with the number of partitions and amount of executors. We decreased the number of executors from 70 to 10, and halved the number of partitions of our data frame. We didn't encounter the heartbeat error afterwards, but in retrospect this solution makes no sense. When we tried simply halving the partitions and keeping 70 executors, the heartbeat error returned, despite the fact that the executors should have less work per task under this new configuration. In light of the solution we ended up finding, our hypothesis is that the problem isn't with the executors, but with the driver not being able to keep up with the amount of executors we have instantiated.

2. Increase executor memory.
Keeping our original configuration, we then tried simply increasing the executor memory. Doing so seemed to help alleviate the timeout error, but didn't appear to be overly effective. Why?

3 Increase the driver memory.
Keeping to our original configuration, the only thing we changed was increasing the driver memory from 30 to 35 GB. This most effectively eliminated the heartbeat error. Increasing the memory increases the size of the JVM, meaning it can handle the increased number of executors without them timing out.

Py4JJavaError

The second main error we ran into was when converting a pyspark RDD into a Pandas RDD using the toPandas() function.

```
_review_business = review_business.toPandas()
```

We know that there's likely an issue with Py4j. Py4j is the connection between the Pyspark and Java, and we can interact with the JVM through the JavaGateway class, which can be found in the spark libs directory. We started to try out different JavaGateway methods to solve this error, namely by explicitly launching the gateway before running the toPandas() method.

```
from pyspark.java_gateway import launch_gateway
```

```
launch_gateway()
```

Explicitly launching the gateway did nothing to solve the problem, but despite this we're still pretty confident it's an issue with communication with the server, whether the program is somehow no longer binding to the correct port or if the costliness of the function is just too great for the JVM.

Sources

- https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=0ahUKEwjIte6x2uvXAhWMON8KHbKrDkMQjhwIBQ&url=http%3A%2F%2Fbigdata-madesimple.com%2Fwhat-is-the-role-of-rdds-in-apache-spark-part-1%2F&psig=AOvVaw1N0bQRGY5cmA719QPHWMWU&ust=1512317103325036
- https://stackoverflow.com/questions/26870537/what-is-the-difference-between-cache-and-persist
- https://stackoverflow.com/questions/28981359/why-do-we-need-to-call-cache-or-persist-on-a-rdd
- https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#resilient-distributed-datasets-rdds
- https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details
-