3. Execution plans, Lazy Evaluation, and caching

October 11, 2018

0.1 Execution Plans, Lazy Evaluation and Caching

0.1.1 Task: calculate the sum of squares :

$$\sum_{i=1}^{n} x_i^2$$

The standard (or **busy**) way to do this is 1. Calculate the square of each element. 2. Sum the squares.

This requires **storing** all intermediate results.

0.1.2 lazy evaluation:

- **postpone** computing the square until result is needed.
- No need to store intermediate results.
- Scan through the data once, rather than twice.

0.1.3 Lazy Evaluation

Unlike a regular python program, map/reduce commands do not always perform any computation when they are executed. Instead, they construct something called an **execution plan**. Only when a result is needed does the computation start. This approach is also called **lazy execution**.

The benefit from lazy execution is in minimizing the the number of memory accesses. Consider for example the following map/reduce commands:

```
A=RDD.map(lambda x:x*x).filter(lambda x: x%2==0)
A.reduce(lambda x,y:x+y)
```

The commands defines the following plan. For each number `x` in the RDD: 1. Compute the square of `x` 2. Filter out `x*x` whose value is odd. 3. Sum the elements that were not filtered out.

A naive execution plan is to square all items in the RDD, store the results in a new RDD, then perform a filtering pass, generating a second RDD, and finally perform the summation. Doing this will require iterating through the RDD three times, and creating 2 interim RDDs. As memory access is the bottleneck in this type of computation, the execution plan is slow.

A better execution plan is to perform all three operations on each element of the RDD in sequence, and then move to the next element. This plan is faster because we iterate through the elements of the RDD only once, and because we don't need to save the intermediate results. We need to maintain only one variable: the partial sum, and as that is a single variable, we can use a CPU register.

For more on RDDs and lazy evaluation see here in the spark manual

0.2 Experimenting with Lazy Evaluation

The `%%time` magic The `%%time` command is a *cell magic* which measures the execution time of the cell. We will mostly be interested in the wall time, which includes the time it takes to move data in the memory hierarchy.

For more on jupyter magics See here

0.2.1 Preparations

In the following cells we create an RDD and define a function which wastes some time and then returns `cos(i)`. We want the function to waste some time so that the time it takes to compute the `map` operation is significant.

```
In [6]: #start the SparkContext
        import findspark
        findspark.init()

        from pyspark import SparkContext
        sc = SparkContext()  #note that we set the number of workers to 3


        ---------------------------------------------------------------------------

        Py4JError                                 Traceback (most recent call last)

        <ipython-input-6-bc4535848905> in <module>()
          4
          5 from pyspark import SparkContext
    ----> 6 sc = SparkContext()  #note that we set the number of workers to 3


        /anaconda3/lib/python3.6/site-packages/pyspark/context.py in __init__(self, master, ap
        116         try:
        117             self._do_init(master, appName, sparkHome, pyFiles, environment, batchS
    --> 118                           conf, jsc, profiler_cls)
        119         except:
        120             # If an error occurs, clean up in order to allow future SparkContext c


        /anaconda3/lib/python3.6/site-packages/pyspark/context.py in _do_init(self, master, ap
        186         self._accumulatorServer = accumulators._start_update_server()
        187         (host, port) = self._accumulatorServer.server_address
    --> 188         self._javaAccumulator = self._jvm.PythonAccumulatorV2(host, port)
        189         self._jsc.sc().register(self._javaAccumulator)
        190


        /anaconda3/lib/python3.6/site-packages/py4j/java_gateway.py in __call__(self, *args)
       1523         answer = self._gateway_client.send_command(command)
```

```
   1524         return_value = get_return_value(
-> 1525             answer, self._gateway_client, None, self._fqn)
   1526 
   1527         for temp_arg in temp_args:


    /anaconda3/lib/python3.6/site-packages/py4j/protocol.py in get_return_value(answer, ga
    330                 raise Py4JError(
    331                     "An error occurred while calling {0}{1}{2}. Trace:\n{3}\n".
--> 332                     format(target_id, ".", name, value))
    333         else:
    334             raise Py4JError(


    Py4JError: An error occurred while calling None.org.apache.spark.api.python.PythonAccu
py4j.Py4JException: Constructor org.apache.spark.api.python.PythonAccumulatorV2([class java
	at py4j.reflection.ReflectionEngine.getConstructor(ReflectionEngine.java:179)
	at py4j.reflection.ReflectionEngine.getConstructor(ReflectionEngine.java:196)
	at py4j.Gateway.invoke(Gateway.java:237)
	at py4j.commands.ConstructorCommand.invokeConstructor(ConstructorCommand.java:80)
	at py4j.commands.ConstructorCommand.execute(ConstructorCommand.java:69)
	at py4j.GatewayConnection.run(GatewayConnection.java:238)
	at java.lang.Thread.run(Thread.java:745)
```

We create an RDD with one million elements to amplify the effects of lazy evaluation and caching.

```
In [2]: %%time
        RDD=sc.parallelize(range(1000000))
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 604 ms
```

It takes about 01.-0.5 sec. to create the RDD.

```
In [3]: print(RDD.toDebugString().decode())
```

```
(4) PythonRDD[1] at RDD at PythonRDD.scala:48 []
 |  ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:175 []
```

0.2.2 Define a computation

The role of the function `taketime` is to consume CPU cycles.

```
In [4]: from math import cos
        def taketime(i):
            [cos(j) for j in range(100)]
            return cos(i)
```

```
In [5]: %%time
        taketime(1)

CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 52 ţs


Out[5]: 0.5403023058681398
```

0.2.3 Time units

- 1 second = 1000 Milli-second (*ms*)
- 1 Millisecond = 1000 Micro-second (μs)
- 1 Microsecond = 1000 Nano-second (*ns*)

0.2.4 Clock Rate

One cycle of a 3GHz cpu takes $\frac{1}{3}ns$
`taketime(1000)` takes about 25 μs = 75,000 clock cycles.

0.2.5 The `map` operation.

```
In [6]: %%time
        Interm=RDD.map(lambda x: taketime(x))

CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 33.4 ţs
```

0.2.6 How come so fast?

- We expect this map operation to take 1,000,000 * 25 μs = 25 Seconds.
- **Why** did the previous cell take just 29 μs?
- Because **no computation was done**
- The cell defined an **execution plan**, but did not execute it yet.

Lazy Execution refers to this type of behaviour. The system delays actual computation until the latest possible moment. Instead of computing the content of the RDD, it adds the RDD to the **execution plan**.

Using Lazy evaluation of a plan has two main advantages relative to immediate execution of each step: 1. A single pass over the data, rather than multiple passes. 2. Smaller memory footprint becase no intermediate results are saved.

0.2.7 Execution Plans

At this point the variable `Interm` does not point to an actual data structure. Instead, it points to an execution plan expressed as a **dependence graph**. The dependence graph defines how the RDDs are computed from each other.

The dependence graph associated with an RDD can be printed out using the method `toDebugString()`.

```
In [7]: print(Interm.toDebugString().decode())
```

```
(4) PythonRDD[2] at RDD at PythonRDD.scala:48 []
 |  ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:175 []
```

Interm= `(4) PythonRDD[2] at RDD at PythonRDD.scala:48 []`
______`(4)` corresponds to the number of partitions
RDD = `|  ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:489 []`
At this point only the two left blocks of the plan have been declared.

0.2.8 Actual execution

The `reduce` command needs to output an actual output, **spark** therefor has to actually execute the `map` and the `reduce`. Some real computation needs to be done, which takes about 1 - 3 seconds (Wall time) depending on the machine used and on it's load.

```
In [8]: %%time
        print('out=',Interm.reduce(lambda x,y:x+y))
```

```
out= -0.2887054679684464
CPU times: user 10 ms, sys: 10 ms, total: 20 ms
Wall time: 25 s
```

0.2.9 How come so fast? (take 2)

- We expect this map operation to take 1,000,000 * 25 μs = 25 Seconds.
- Map+reduce takes only ~4 second.
- Why?

- Because we have 4 workers, rather than one.
- Because the measurement of a single iteration of `taketime` is an overestimate.

0.2.10 Executing a different calculation based on the same plan.

The plan defined by `Interm` might need to be executed more than once.

Example: compute the number of map outputs that are larger than zero.

```
In [9]: %%time
        print('out=',Interm.filter(lambda x:x>0).count())
```

```
out= 500000
CPU times: user 20 ms, sys: 0 ns, total: 20 ms
Wall time: 22.7 s
```

0.2.11 The price of not materializing

- The run-time (3.4 sec) is similar to that of the reduce (4.4 sec).
- Because the intermediate results in `Interm` have not been saved in memory (materialized)
- They need to be recomputed.

The middle block: `Map(Taketime)` is executed twice. Once for each final step.

0.2.12 Caching intermediate results

- We sometimes want to keep the intermediate results in memory so that we can reuse them later without recalculating. * This will reduce the running time, at the cost of requiring more memory.
- The method `cache()` indicates that the RDD generates in this plan should be stored in memory. Note that this is a **plan to cache**. The actual caching will be done only when the final result is needed.

```
In [10]: %%time
         Interm=RDD.map(lambda x: taketime(x)).cache()
```

```
CPU times: user 10 ms, sys: 0 ns, total: 10 ms
Wall time: 47.1 ms
```

By adding the Cache after `Map(Taketime)`, we save the results of the map for the second computation.

0.2.13 Plan to cache

The definition of `Interm` is almost the same as before. However, the *plan* corresponding to `Interm` is more elaborate and contains information about how the intermediate results will be cached and replicated.

Note that `PythonRDD[4]` is now [Memory Serialized 1x Replicated]

```
In [11]: print(Interm.toDebugString().decode())
```

```
(4) PythonRDD[5] at RDD at PythonRDD.scala:48 [Memory Serialized 1x Replicated]
 |  ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:489 [Memory Serialized 1x Repli
```

Comparing plans with and without cache Plan with Cache

```
(4) PythonRDD[33] at RDD at PythonRDD.scala:48 [Memory Serialized 1x Replicated]
 |  ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:489 [Memory Serialized 1x Repli
```

Plan without Cache

```
(4) PythonRDD[2] at RDD at PythonRDD.scala:48 []
 |  ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:489 []
```

The difference is that the plan for both RDDs includes **[Memory Serialized 1x Replicated]** which is the plan to materialize both RDDs when they are computed.

0.2.14 Creating the cache

The following command executes the first map-reduce command **and** caches the result of the `map` command in memory.

```
In [12]: %%time
         print('out=',Interm.reduce(lambda x,y:x+y))
```

```
out= -0.2887054679684655
CPU times: user 5.43 ms, sys: 3.79 ms, total: 9.22 ms
Wall time: 3.59 s
```

0.2.15 Using the cache

This time `Interm` is cached. Therefor the second use of `Interm` is much faster than when we did not use `cache`: 0.25 second instead of 1.9 second. (your milage may vary depending on the computer you are running this on).

```
In [13]: %%time
         print('out=',Interm.filter(lambda x:x>0).count())
```

```
out= 500000
CPU times: user 5.31 ms, sys: 2.97 ms, total: 8.28 ms
Wall time: 121 ms
```

0.3 Summary

- Spark uses **Lazy Evaluation** to save time and space.
- When the same RDD is needed as input for several computations, it can be better to keep it in memory, also called `cache()`.
- Next Video, Partitioning and Gloming

0.4 Partitioning and Gloming

- When an RDD is created, you can specify the number of partitions.
- The default is the number of workers defined when you set up `SparkContext`

```
In [14]: A=sc.parallelize(range(1000000))
         print(A.getNumPartitions())
```

```
4
```

We can repartition `A` into a different number of partitions.

```
In [15]: D=A.repartition(10)
         print(D.getNumPartitions())
```

```
10
```

We can also define the number of partitions when creating the RDD.

```
In [16]: A=sc.parallelize(range(1000000),numSlices=10)
         print(A.getNumPartitions())
```

```
10
```

0.4.1 Why is the #Partitions important?

- They define the unit the executor works on.
- You should have at least as pany partitions as workers.
- Smaller partitions can allow more parallelization.

0.5 Repartitioning for Load Balancing

Suppose we start with 10 partitions, all with exactly the same number of elements

```
In [17]: A=sc.parallelize(range(1000000))\
             .map(lambda x:(x,x)).partitionBy(10)
         print(A.glom().map(len).collect())
```

```
[100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000]
```

- Suppose we want to use `filter()` to select some of the elements in `A`.
- Some partitions might have more elements remaining than others.

```
In [18]: #select 10% of the entries
         B=A.filter(lambda pair: pair[0]%5==0)
         # get no. of partitions
         print(B.glom().map(len).collect())
```

```
[100000, 0, 0, 0, 0, 100000, 0, 0, 0, 0]
```

- Future operations on `B` will use only two workers.
- The other workers will do nothing,
  because their partitions are empty.

- To fix the situation we need to repartition the RDD.

- One way to do that is to repartition using a new key.

- The method `.partitionBy(k)` expects to get a `(key,value)` RDD where keys are integers.

- Partitions the RDD into `k` partitions.
- The element `(key,value)` is placed into partition no. `key % k`

```
In [19]: C=B.map(lambda pair:(pair[1]/10,pair[1])).partitionBy(10)
         print(C.glom().map(len).collect())
```

```
[20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000]
```

Another approach is to use random partitioning using `repartition(k)` * An **advantage** of random partitioning is that it does not require defining a key. * A **disadvantage** of random partitioning is that you have no control on the partitioning.

```
In [20]: C=B.repartition(10)
         print(C.glom().map(len).collect())
```

```
[20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000]
```

0.5.1 Glom()

- In general, spark does not allow the worker to refer to specific elements of the RDD.
- Keeps the language clean, but can be a major limitation.

- **glom()** transforms each partition into a tuple (immutabe list) of elements.
- Creates an RDD of tules. One tuple per partition.
- workers can refer to elements of the partition by index.
- but you cannot assign values to the elements, the RDD is still immutable.

- Now we can understand the command used above to count the number of elements in each partition.
- We use `glom()` to make each partition into a tuple.
- We use `len` on each partition to get the length of the tuple - size of the partition.
- We `collect` the results to print them out.

```
In [21]: print(C.glom().map(len).collect())
```

```
[20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000]
```

0.5.2 A more elaborate example

There are many things that you can do using `glom()`.

Below is an example, can you figure out what it does?

```
In [22]: def getPartitionInfo(G):
             d=0
             if len(G)>1:
                 for i in range(len(G)-1):
                     d+=abs(G[i+1][1]-G[i][1]) # access the glomed RDD that is now a  list
                 return (G[0][0],len(G),d)
             else:
                 return(None)

         output=B.glom().map(lambda B: getPartitionInfo(B)).collect()
         print(output)
```

```
[(0, 100000, 999990), None, None, None, None, (5, 100000, 999990), None, None, None, None]
```

0.6 Summary

- We learned why partitions are important and how to control them.
- We Learned how `glom()` can be used to allow workers to access their partitions as lists.