Lab 6.1: Caching

Overview

In this lab, we will execute a transformation with caching and without caching, to see how it can help with performance. We'll work with both a DataFrame and an RDD to see the (very large) di!erence in memory usage.

We will use a large (500 MB) data file we supply ***spark-labs/data/twinkle/500M.data***.

Builds on

None

Run time

15-20 minutes

Load a File into a DataFrame Without Caching

Tasks

- In the Spark shell, load the ***spark-labs/data/twinkle/500M.data*** file into a dataframe.
    - Count the number of rows 3 times.
        - So we can get a stable time - there are non-Spark mechanisms in play, e.g. OS caching of files. Look
    - at the Web UI Jobs tab
    - See how long each count took, how much data was input, and how much storage is used.
        - You can see the total time on the **main jobs tab**.
        - You can see how much data was input by **drilling down through a particular job**.
    - In our runs, read times leveled out at about 0.7s (first couple of reads were 2s-4s).
    - Input size was 500+MB for each count.

Scala

```
> val twinkleDF = spark.read.text("spark-labs/data/twinkle/500M.data")
```

Python

```
> twinkleDF = spark.read.text("spark-labs/data/twinkle/500M.data")
```

Scala and Python

```
> twinkleDF.count()
> twinkleDF.count()
> twinkleDF.count()
```

Cache the File, then Count Again

Tasks

- Cache the dataframe, then try counting three times again.

```
> twinkleDF.cache()
>
> twinkleDF.count()
> twinkleDF.count()
> twinkleDF.count()
```

- Look at the Web UI Jobs tab
    - See how long each count took, how much data was input, and how much storage was used.
        - You can see the storage usage on the **Storage Tab**.
    - In our runs, job times took from 86ms to 5s.
        - First read was 5 sec
        - After the first read, which caused the data to be cached, the time needed become much smaller than with the uncached data. It leveled out at under 0.1s.
    - The storage size after the first cached run was about 38MB. Why so small?
        - It's because Tungsten uses a very, very e"cient storage mechanismn, and can compress the data very e"ciently.
        - It does not need to cache complete Java objects, and does not need to decompress it for this operation (can access the data in compressed format).

    - Input size was 500+MB for the first run after caching (since the data was not yet cached), after that it was about 38MB, as the data was read from cache.

Do the Same thing with an RDD

Tasks

- Let's go through the same sequence with an RDD, as illustrated below.
    - As with the dataframe, look at the Web UI for all the transformations, including job time, input size, and storage size.

Scala

```
// Scala
> val twinkleRDD = sc.textFile("spark-labs/data/twinkle/500M.data")
```

Python

```
> twinkleRDD = sc.textFile("spark-labs/data/twinkle/500M.data")
```

Scala and Python

```
> twinkleRDD.count()
// Count three times
// Look at the Web UI for the info we want
```

```
> twinkleRDD.cache()
> twinkleRDD.count()
// Count three times
// Look at the Web UI for the info we want
```

- **Uncached** results on our system (Scala):
    - Job times were about 1 sec (and steady) The
    - input size was 500MB
    - No storage was used yet since it wasn't cached.
- **Cached** results on our system:
    - First job time was 9 sec. Subsequent job times were 0.1-0.2 sec. The
    - storage used was **1.8 GB**. Why?
        - Because an RDD is saved as a Java serialized object - clearly an

ine"cient storage format.

- The input size was 500MB for the first run.
- The input size was **1.8 GB** for subsequent runs. It was reading the Java serialized data from cache. Faster than reading from a file, but much more data than with a dataframe.

Summary

You can see that caching can have significan impacts on how your transformations run. With more expensive transformations, this can make an even bigger di!erence.

It comes at a cost, memory used for storage. It should only be used when you're going to use the results of the transformation repeatedly.

DataFrames are much more e"cient than RDDs, due to the advances in data storage that come with Tungsten. Another reason to use Spark SQL / DataFrames.