

02 Spark Caching Overview -KirkYagami



Link to the original article, [here]([Best practices for caching in Spark SQL | by David Vrba | Towards Data Science](#))

Best practices for caching in Spark SQL

Deep dive into data persistence in Spark.

In Spark SQL caching is a common technique for reusing some computation. It has the potential to speedup other queries that are using the same data, but there are some caveats that are good to keep in mind if we want to achieve good performance. In this article, we will take a look under the hood to see how caching works internally and we will try to demystify Spark's behavior related to data persistence.

Using DataFrame API

In DataFrame API, there are two functions that can be used to cache a DataFrame, `cache()` and `persist()`:

```
df.cache()
```

```
df.persist()
```

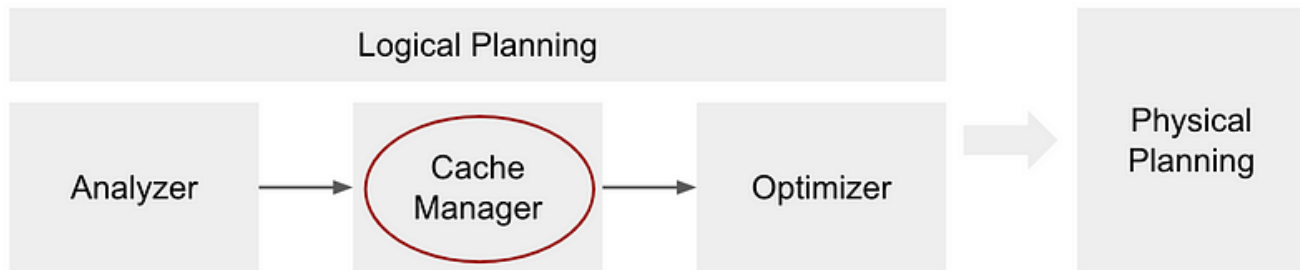
They are almost equivalent, the difference is that `persist` can take an optional argument `storageLevel` by which we can specify where the data will be persisted. The default value of the `storageLevel` for both functions is `MEMORY AND DISK` which means that the data will be stored in memory if there is space for it, otherwise, it will be stored on disk. [Here](#) you can see the (PySpark) documentation for other possible storage levels.

Caching is a lazy transformation, so immediately after calling the function nothing happens with the data but the query plan is updated by the `Cache Manager` by adding a new operator — `InMemoryRelation`. So this is just some information that will be used during the query execution later on when some action is called. Spark will look for the data in the caching layer and read it from there if it is available. If it doesn't find the data in the caching layer (which happens for sure the first time the query runs), it will become responsible for getting the data there and it will use it immediately afterward.

Cache Manager

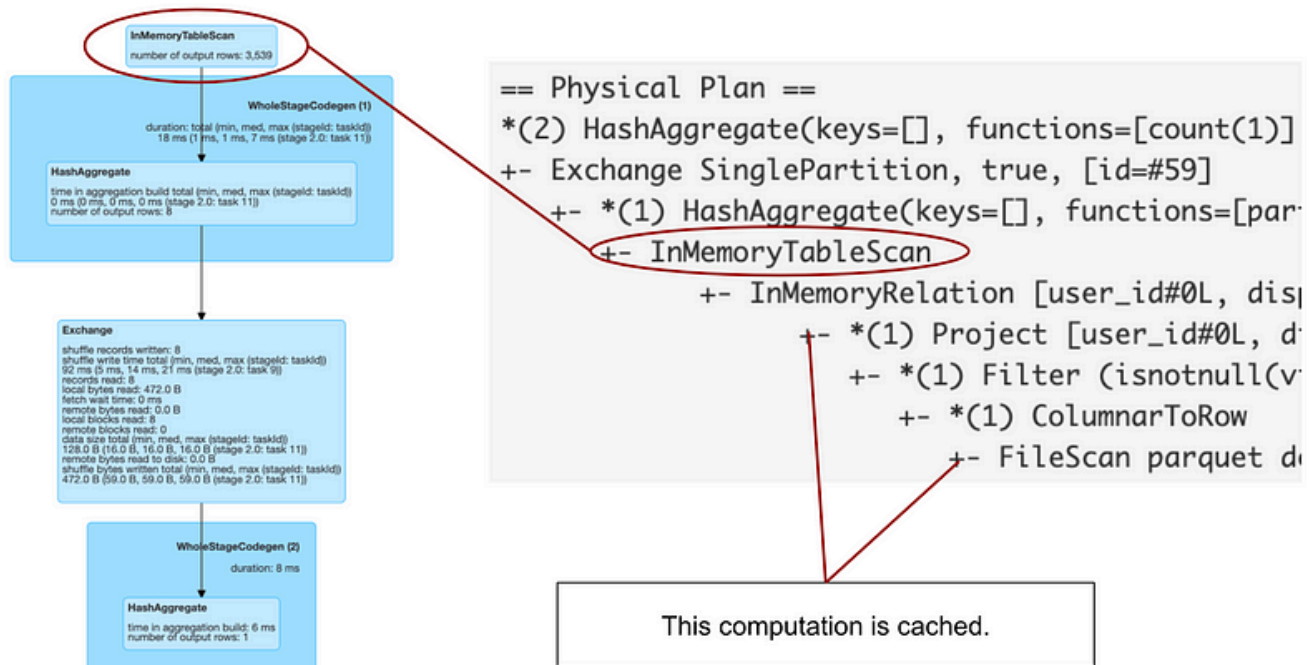
The `Cache Manager` is responsible to keep track of what computation has already been cached in terms of the query plan. When the caching function is called, the `Cache Manager` is invoked directly under the hood and it pulls out the `analyzed logical plan` of the DataFrame on which the caching function is called and stores that plan in an indexed sequence called `cachedData`.

The phase of the `Cache Manager` is part of logical planning and it takes place after the analyzer and before the optimizer:



When you run a query with an action, the query plan will be processed and transformed. In the step of the **Cache Manager** (just before the optimizer) Spark will check for each subtree of the analyzed plan if it is stored in the **cachedData** sequence. If it finds a match it means that the same plan (the same computation) has already been cached (perhaps in some previous query) and so Spark can use that and thus it adds that information to the query plan using the **InMemoryRelation** operator which will carry information about this cached plan. This **InMemoryRelation** is then used in the phase of physical planning to create a physical operator— **InMemoryTableScan**.

```
df = spark.table("users").filter(col(col`name) > x).cache()
df.count() # now check the query plan in Spark UI
```



Here in the above picture you can see graphical and string representation of a query which was using caching. To see what transformations were cached you need to look into the string representation of the plan because the graphical representation doesn't show this information.

Basic example

Let's see a simple example to understand better how the **Cache Manager** works:

```
df = spark.read.parquet(data`path)
```

```
df.select(col1, col2).filter(col2 > 0).cache()
```

Consider the following three queries. Which one of them will leverage the cached data?

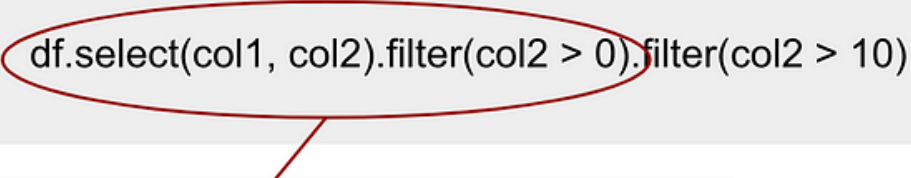
1. `df.filter(col2 > 0).select(col1, col2)`
2. `df.select(col1, col2).filter(col2 > 10)`
3. `df.select(col1).filter(col2 > 0)`

The decisive factor is the **analyzed logical plan**. If it is the same as the analyzed plan of the cached query, then the cache will be leveraged. For query number 1 you might be tempted to say that it has the same plan because the filter will be pushed by the optimizer in both cases anyway. But this is actually not entirely accurate. The important thing to understand is that the phase of the **Cache Manager** takes place **before** the optimizer. What would be the same are the optimized plans but not analyzed plans. So query n. 1 will not leverage the cache simply because the analyzed plans are different.

For query n. 2 you might be again tempted to assume that it will use the cached data because the filter is more restrictive than the filter in the cached query. We can logically see that the queried data is in the cache, but Spark will not read it from there because of the same reason as before — the analyzed plans are different — this time the filtering condition is not the same. To use the cached data we can, however, fix the second query just by adding the filter there:

```
df.select(col1, col2).filter(col2 > 0).filter(col2 > 10)
```

At first sight, the filter `col2 > 0` seems to be useless here, but it is not because now part of the analyzed logical plan will be identical with the cached plan and the **Cache Manager** will be able to find it and use the **InMemoryRelation** in the query plan.



```
df.select(col1, col2).filter(col2 > 0).filter(col2 > 10)
```

This has exactly the same analyzed plan as the cached query so it can be replaced by **InMemoryRelation**

Query number 3 is tricky, at first sight, it appears that it also will have a different analyzed plan because the query is different — we select only `col1`. The filtering condition is, however, using `col2`, which is not present in the previous projection and so the analyzer will invoke a rule **ResolveMissingReferences** and it will add `col2` to the projection and the analyzed plan will actually become identical with the cached plan. This time the **Cache Manager** will find it and use it.

So the final answer is that query n. 3 will leverage the cached data.

Best practices

Let's list a couple of rules of thumb related to caching:

- ◆ When you cache a DataFrame create a new variable for it `cachedDF = df.cache()`. This will allow you to bypass the problems that we were solving in our example, that sometimes it is not clear what is the analyzed plan and what was actually cached. Here whenever you call `cachedDF.select(...)` it will leverage the cached data.
- ◆ Unpersist the DataFrame after it is no longer needed using `cachedDF.unpersist()`. If the caching layer becomes full, Spark will start evicting the data from memory using the LRU (least recently used) strategy. So it is good practice to use `unpersist` to stay more in control about what should be evicted. Also, the more space you have in memory the more can Spark use for execution, for instance, for building hash maps and so on.
- ◆ Before you cache, make sure you are caching only what you will need in your queries. For example, if one query will use `(col1, col2, col3)` and the second query will use `(col2, col3, col4)`, select a superset of these columns: `cachedDF = df.select(col1, col2, col3, col4).cache()`. It is not very useful to call `cachedDF = df.cache()` if the `df` contains lots of columns and only a small subset will be needed in follow-up queries.
- ◆ Use the caching only if it makes sense. That is if the cached computation will be used multiple times. It is good to understand that putting the data to memory is also related to some overhead, so in some cases, it might be even faster to simply run the computation again if it is fast enough and not use caching at all as you can see in the next paragraph.

Faster than caching

There are situations where caching doesn't help at all and on the contrary slows down the execution. This is related for instance to queries based on large datasets stored in a columnar file format that supports column pruning and predicate pushdown such as parquet. Let's consider the following example, in which we will cache the entire dataset and then run some queries on top of it. We will use the following dataset and cluster properties:

dataset size: 14.3GB in compressed parquet sitting on S3

cluster size: 2 workers c5.4xlarge (32 cores together)

platform: Databricks (runtime 6.6 with Spark 2.4.5)

First, let's measure the execution times for the queries where caching is not used:

```
df = spark.table(table`name`)
df.count() # runs 7.9s
df.filter(col("id") > xxx).count() # runs 18.2s
```

Now run the same queries with caching (the entire dataset doesn't fit in memory and about 30% is cached on disk):

```
df = spark.table(table`name`).cache()
```

this count will take long because it is putting data to memory

```
df.count() # runs 1.28min
df.count() # runs 14s
df.filter(col("id") > xxx).count() # runs 20.6s
```

No wonder that the first count takes 1.3min, there is the overhead related to putting data to memory. However, as you can see, also the second count and the query with the filter take longer for the cached dataset as compared to reading directly from parquet. It is a combination of two major reasons. The first one is the properties of the parquet file format — queries based on top of parquet are fast on its own. In the case of reading from parquet, Spark will read only the metadata to get the count so it doesn't need to scan the entire dataset. For the filtering query, it will use column pruning and scan only the `id` column. On the other hand, when reading the data from the cache, Spark will read the entire dataset. This can be seen from Spark UI, where you can check the input size for the first stage (see the picture below).

The second reason is that the dataset is large and doesn't fit entirely in ram. Part of the data is stored on disk as well and reading from the disk is much slower than reading from ram.

Storage tab that shows how much data is cached in ram and on disk:

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
80	FileScan parquet	Disk Serialized 1x Replicated	200	100%	20.8 GB	10.5 GB

Input to the first stage when reading directly from parquet:

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
17	3427354326394563716	df.filter(col("id") > "1000").count() count at NativeMethodAccessorImpl.java:0	2020/07/19 05:51:27	37 ms	1/1			10.4 KB	
16	3427354326394563716	df.filter(col("id") > "1000").count() count at NativeMethodAccessorImpl.java:0	2020/07/19 05:51:09	18 s	181/181	619.7 MB			10.4 KB

Input to the first stage when reading from cache:

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
23	3427354326394563716	df.filter(col("id") > "1000").count() count at NativeMethodAccessorImpl.java:0	2020/07/19 05:53:38	40 ms	1/1			11.5 KB	
22	3427354326394563716	df.filter(col("id") > "1000").count() count at NativeMethodAccessorImpl.java:0	2020/07/19 05:53:18	20 s	200/200	31.2 GB			11.5 KB

Caching in SQL

If you prefer using directly SQL instead of DataFrame DSL, you can still use caching, there are some differences, however.

```
spark.sql("cache table table`name")
```

The main difference is that using SQL the caching is eager by default, so a job will run immediately and will put the data to the caching layer. To make it lazy as it is in the DataFrame DSL we can use the `lazy` keyword explicitly:

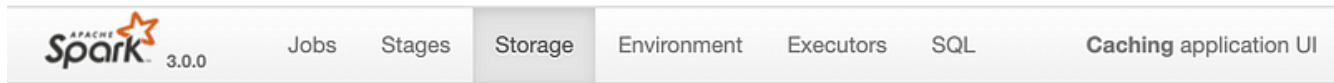
```
spark.sql("cache lazy table table`name")
```

To remove the data from the cache, just call:

```
spark.sql("uncache table table`name")
```

See the cached data

Sometimes you may wonder what data is already cached. One possibility is to check Spark UI which provides some basic information about data that is already cached on the cluster.



Storage

▼ RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
3	In-memory table `default`.`users`	Disk Memory Deserialized 1x Replicated	8	100%	15.5 MiB	0.0 B

Here for each cached dataset, you can see how much space it takes in memory or on disk. You can even zoom more and click on the record in the table which will take you to another page with details about each partition.

To check whether the entire table is cached we can use Catalog API:

```
spark.catalog.isCached("table`name")
```

The Catalog API can also be used to remove all data from the cache as follows:

```
spark.catalog.clearCache()
```

In Scala API you can also use the internal API of the **Cache Manager** which provides some functions, for instance, you can ask whether the **Cache Manager** is empty:

// In Scala API:

```
val cm = spark.sharedState.cacheManager  
cm.isEmpty
```

Other possibilities for data persistence

Caching is one of more techniques that can be used for reusing some computation. Apart from caching there is also checkpointing and exchange-reuse.

The checkpointing is useful for instance in situations where we need to break the query plan because it is too large. A large query plan may become a bottleneck in the driver where it is processed because the processing of a very large plan will take too long. The checkpoint will however break the plan and materialize the query. For

the next transformations, Spark will start building a new plan. The checkpointing is related to two functions `checkpoint` [↗](#) and `localCheckpoint` [↗](#) which differ by the storage used for the data.

The exchange-reuse where Spark persists the output of a shuffle on disk, is, on the other hand, a technique that can not be controlled directly by some API function, but instead, it is an internal feature that Spark handles on its own. In some special situations, it can be controlled indirectly by rewriting the query trying to achieve identical exchange branches. To read more about exchange-reuse, you can check my other [article](#) [↗](#), where I describe it more in detail.

Conclusion

In this article, we tried to demystify Spark's behavior related to caching. We have seen how it works under the hood and what are the differences between using DSL vs SQL. We discussed some best practices on how to make caching as efficient as possible. On one example we showed that for big datasets that do not fit in memory, it might be faster to avoid caching especially if the data is stored in columnar file format. We also mentioned some alternatives to caching such as checkpointing or reused exchange that can be useful for data persistence in some situations.