Spark performance tuning from the trenches

Yann Moisan Follow
May 29, 2018 · 9 min read

Spark is the core component of Teads's Machine Learning stack. We use it for many ML applications, from ad performance predictions to user Look-alike Modeling. We also use Spark for processing intensive jobs like cross-device segment extension or Parquet to SSTables transformation for loading data into Cassandra.

Working with Spark we regularly reach the limits of our clusters' resources in terms of memory, disk or CPU. A scale-out only pushes back the issue so **we have to get our hands dirty**.

Here is a collection of **best practices and optimization tips for Spark 2.2.0** to achieve better performance and cleaner Spark code, covering:

- How to leverage Tungsten,

- Execution plan analysis,
- Data management (caching, broadcasting),
- Cloud-related optimizations (including S3).

***Update 07/12/2018**, see also the second part covering troubleshooting tricks and external data source management.*

1- Use the power of Tungsten

It's common sense, but the best way to improve code performance is to **embrace Spark's strengths**. One of them is Tungsten.

Standard since version 1.5, Tungsten is a Spark SQL component that provides increased performance by rewriting Spark operations in bytecode, at runtime. Tungsten suppresses virtual functions and leverages close to bare metal performance by focusing on jobs CPU and memory efficiency.

To make the most out of Tungsten we pay attention to the following:

Use Dataset structures rather than DataFrames

To make sure our code will benefit as much as possible from Tungsten optimizations **we use the default *Dataset* API** with Scala (instead of *RDD*).

Dataset brings the **best of both worlds** with a mix of relational (*DataFrame*) and functional (*RDD*) transformations. This API is the most up to date and adds type-safety along with better error handling and far more readable unit tests.

However, it comes with **a tradeoff** as *map* and *filter* functions perform poorer with this API. *Frameless* is a promising solution to tackle this limitation.

Avoid User-Defined Functions (UDFs) as much as possible

Using a UDF implies deserialization to process the data in classic Scala and then reserialize it. UDFs can be replaced by **Spark SQL functions**, there are already a lot of them and new ones are regularly added.

Avoiding UDFs **might not generate instant improvements** but at least it will prevent future performance issues, should the code change. Also, by using built-in Spark SQL functions we cut down our testing effort as everything is performed on Spark's side. These functions are designed by JVM experts so UDFs are not likely to achieve better performance.

For example the following code can be replaced by the built-in *coalesce* function:

```
def currency = udf(
(currencySub: String, currencyParent: String) ⇒
   Option(currencyParent) match {
      case Some(curr) ⇒ curr
      case _ ⇒ currencySub
   }
)
```

When there is no built-in replacement, it is still possible to **implement and extend Catalyst's** (Spark's SQL optimizer) *expression* class. It will play well with code generation. For more details, Chris Fregly talked about it here (see slide 56). By doing this we directly access Tungsten format, it solves the serialization problem and bumps performance.

Avoid User-Defined Aggregate Functions (UDAFs)

A UDAF generates *SortAggregate* operations which are significantly slower than *HashAggregate*. For example, what we do instead of writing a UDAF that compute a median is using a built-in equivalent (quantile 0,5):

```
df.stat.approxQuantile("value", Array(0.5), 0)
```

The *approxQuantile* function uses a variation of the *Greenwald-Khanna* algorithm. In our case, **it ended up being 10 times faster** than the equivalent UDAF.

Avoid UDFs or UDAFs that perform more than one thing

Software Craftsmanship principles obviously apply when writing big data stuff (do one thing and do it well). By splitting UDFs **we are able to use built-in** functions for one part of the resulting code. It also greatly simplify testing.

2- Look under the hood

Analysing Spark's **execution plan** is an easy way to spot potential improvements. This plan is composed of stages, which are the physical units of execution in Spark. When we refactor our code, the first thing we look for is an **abnormal number of stages**. A suspicious plan can be one requiring 10 stages instead of 2–3 for a basic *join* operation between two *DataFrames*.

In Spark and more generally in distributed computing, sending data over the network (a.k.a. *Shuffle* in Spark) is the most expensive action. ***Shuffles* are expensive** since they involve disk I/O, data serialization and network I/O. They are needed for operations like *Join* or *groupBy* and happen between stages.

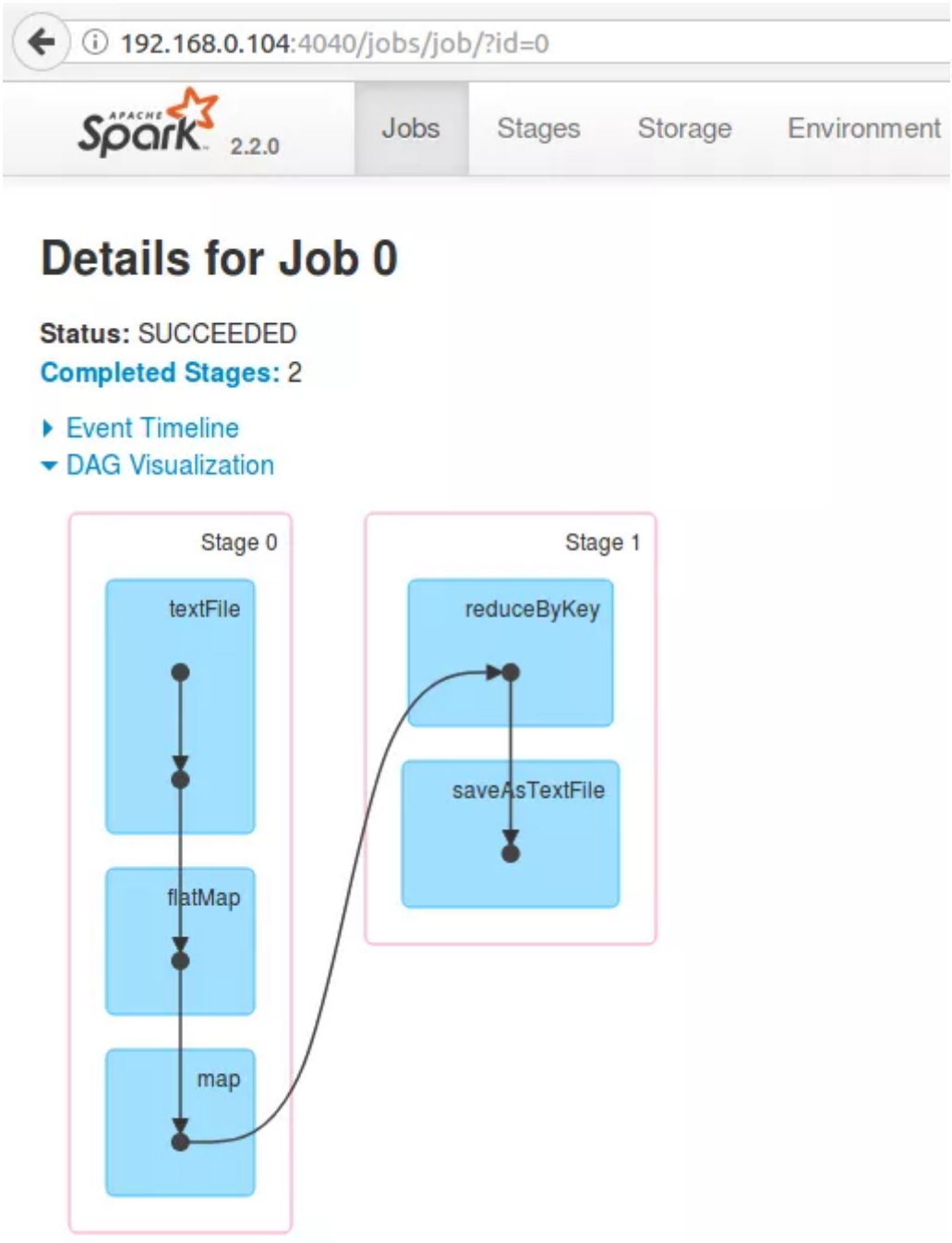
Considering this, **reducing the number of stages is a obvious** way to optimize a job. We use the *.explain(true)* command to show the execution plan detailing all the steps (stages) involved for a job. Here is an example:

```
scala> df.filter("c1!=0").explain
== Physical Plan ==
*Filter (isnotnull(c1#13) && NOT (cast(c1#13 as double) = 0.0))
+- Scan ExistingRDD[c1#13,c2#14,c3#15]

scala> df.filter("c1!=0").explain(true)
== Parsed Logical Plan ==
'Filter NOT ('c1 = 0)
+- LogicalRDD [c1#13, c2#14, c3#15]

== Analyzed Logical Plan ==
c1: string, c2: string, c3: string
Filter NOT (cast(c1#13 as double) = cast(0 as double))
+- LogicalRDD [c1#13, c2#14, c3#15]

== Optimized Logical Plan ==
Filter (isnotnull(c1#13) && NOT (cast(c1#13 as double) = 0.0))
+- LogicalRDD [c1#13, c2#14, c3#15]

== Physical Plan ==
```

```
*Filter (isnotnull(c1#13) && NOT (cast(c1#13 as double) = 0.0))
+- Scan ExistingRDD[c1#13,c2#14,c3#15]
```

Simple execution plan example

The Directed Acyclic Graph (DAG) in Spark UI can also be used to visualize the task repartition in each stage.



A very simple DAG example — Image credits

Optimization relies a lot on both our **knowledge of the data and its processing** (incl. business logic). One of the limits of Spark SQL optimization with Catalyst is that it uses “mechanic” rules to optimize the execution plan (in 2.2.0).

Like many others, we were waiting for a cost-based optimization engine beyond broadcast join selection. It now seems available in 2.3.0, we will have to look at that.

3- Know your data and manage it efficiently

We’ve seen how to improve job performance by looking into the execution plan but there are also plenty of possible enhancements on the data side.

Highly imbalanced datasets

To quickly check if everything is ok we review the execution duration of each task and look for **heterogeneous process time**. If one of the tasks is significantly slower than the others it will extend the overall job duration and waste the resources of the fastest executors.

It’s fairly easy to check min, max and median duration in Spark UI. Here is a balanced example:

Stages tab example on Spark UI

Inappropriate use of caching

There is no universal answer when choosing what should be cached. Caching an intermediate result can **dramatically improve performance** and it's tempting to cache a lot of things. However, due to Spark's caching strategy (in-memory then swap to disk) the cache can end up in a slightly slower storage. Also, using that storage space for caching purposes means that it's not available for processing. In the end, caching might cost more than simply reading the *DataFrame*.

In the *Storage* tab of the UI we verify the *Fraction Cached* and also look at the *Size in Memory* and *Size on Disk* distribution.

Storage tab example on Spark UI

Broadcasting

We regularly use small *DataFrames*, for example when we want to cross a billion auctions with a website list we choose to broadcast the latter to all the executors and **avoid a shuffle**.

```
auction
.join(broadcast(website) as “w”, $”w.id” === $”website_id”)
```

The *broadcast* keyword allows to mark a *DataFrame* that is small enough to be used in broadcast joins.

Broadcast allows to send a read-only variable cached on each node once, rather than sending a copy for all tasks. We try to systematically broadcast small datasets coming from static databases. It's a **quick win** as it's only a single line of code to modify.

Spark is supposed to automatically identify *DataFrames* that should be broadcasted. Be careful though, it's **not as automatic as it appears in the documentation**. Once again, the truth is in the code and the current implementation supposes that Spark knows the *DataFrame's* metadata, which is not effective by default and requires to use Hive and its metastore.

4- Cloud related optimizations

Our Spark clusters run on AWS EMR. EMR provides a managed Hadoop framework on EC2 with YARN to centrally manage cluster resources. Until now we have been using r3.xlarge instances (30Gio, 4 vCPU). We decided to only use one kind of instance so that sizing is simpler.

Here is a configuration that generally gives us good results:

```
-- driver-memory 1g
-- driver-cores 1
-- executor-memory 20g = executor heap size
-- executor-cores 4
-- num-executors $executorCount
```

We revamped our global workflow to **separate the workloads** depending on the use cases. We have three different modes:

- A permanent cluster executing important hourly jobs, essentially performing data preparation for our training jobs,
- Hourly training jobs that spawn their own ephemeral clusters,
- Daily jobs (~2 hours duration) that also spawn their own ephemeral clusters,

We invested time to build a deploy Stack with Jenkins that knows where and when to spawn each job (*spark submit* script). **Ephemeral clusters are killed** once processing is over. It generates great savings, especially since AWS started to bill by the second.

We also **leverage spot instances** for all non permanent clusters. Prices are relatively stable for the instance family we use (previous generation).

Of course, there is always room for improvement. We do not fully use YARN features as we only have Spark applications. In fact, we pay EMR's overhead only for simplicity reasons and could try tools like Flintrock to directly run our clusters on EC2. Here is an article by Heather Miller covering how to use it.

A few precautions using S3

We use S3 for persistent storage but **S3 is not a filesystem**. It's an object store and it means that simple operations are not supported. For example, a simple renaming actually needs to copy and then delete the original file.

The first workaround when using Spark with S3 as an output it to use this specific configuration:

```
spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version 2
spark.speculation false
```

By doing this, files are written progressively instead of being written as a whole at the end of the job. For jobs that read, perform simple transformation and then write the result, we observed an overall improvement of the **execution time by a factor of 2**.

Another solution is the Hadoop output committers for S3, open sourced by Netflix, but we haven't tested it.

We store our files on S3 using Parquet, a columnar storage format. Parquet allows to limit the amount of data read from S3 (only the needed columns are read). It's also worth mentioning that Spark **supports predicate pushdown with Parquet** (i.e. pushing down the filtering closer to the data). It prevents from loading unnecessary parts of the data in-memory and reduce network usage.

However, predicate pushdown should be used with extra care. Even if it appears in the execution plan, **it will not always be effective**. For example, a filter push-down does not work on String and Decimal data types (cf PARQUET-281).

. . .

***See also**, have a look at the other articles of our Spark series covering troubleshooting tricks and external data source management:*

Spark troubleshooting from the trenches

Part II — Troubleshooting tricks and external data source management with JDBC

medium.com

Lessons learned while optimizing Spark aggregation jobs

Spark from the trenches — Part III

medium.com

Spark UDAF could be an option!

Calculate average on sparse arrays — Part IV

medium.com

That's it for now

We hope this selection will be helpful and we thank Spark's vibrant community for sharing such great resources (see references below)!

Performance optimization is a **never-ending topic**, especially with rapidly evolving technologies like Spark. The long awaited 2.3.0 version brings major features like Kubernetes support and a streaming execution engine catching up with Flink and promising *"sub-millisecond end-to-end latency"*.

These advances open exciting new possibilities. If you are interested in Spark and data processing, have a look at the job opportunities at Teads!

By Yann Moisan and Benjamin DAVY

Bibliography

- Advanced Apache Spark Meetup Project Tungsten, by Chris Fregly — 12/11/2015
- Working with UDFs in Apache Spark, by Curtis Howard — 03/02/2017
- Spark Execution Model, Cloudera
- Cassandra write tuning parameters, DataStax
- Apache Spark and Amazon s3 gotchas and best practices, by Subhojit Banerjee — 18/11/2016
- A Spark 2.0.0 Cluster Takes a Longer Time to Append Data, databricks
- How to tune your Apache Spark jobs Part 1 & Part 2, by Sandy Riza — 03/2015
- Spark performance tuning checklist, by Taraneh Khazaei — 08/09/2017
- Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop, by Sameer Agarwal et al. — 23/05/2016
- Diving into Spark and Parquet Workloads, by Example, by Luca Canali — 29/06/2017
- Running Spark on YARN, by Mahmoud Hanafy — 29/02/2015
- How to optimize Apache Spark apps, by Henri Hu — 08/09/2017
- Running Spark on a Cluster: The Basics, by Heather Miller — 09/04/2018

Thanks to Benjamin DAVY and Alban Perillat-Merceroz.

Big Data Apache Spark Optimization Tips And Tricks Best Practices


Download on the
App Store

GET IT ON
Google Play