# **Spark performance tuning from the trenches**

*Spark is the core component of Teads’s*[*Machine Learning stack*](https://medium.com/teads-engineering/teads-machine-learning-stack-tooling-and-community-of-practice-9a20a980fdf)*. 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*](https://medium.com/teads-engineering/spark-from-the-trenches-part-2-f2ff9ab67ea1)*covering troubleshooting tricks and external data source management.*

[Spark from the trenches — Part 2 — Yann Moisan — Medium  
Troubleshooting tricks and external data source management with JDBCmedium.com](https://medium.com/teads-engineering/spark-from-the-trenches-part-2-f2ff9ab67ea1)

**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](https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html).

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***](https://spark.apache.org/docs/latest/sql-programming-guide.html)**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*](https://typelevel.org/frameless/TypedDatasetVsSparkDataset.html) 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**](https://spark.apache.org/docs/latest/sql-programming-guide.html#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**](https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html)**’s** (Spark’s SQL optimizer) *expression* class. It will play well with code generation. For more details, [Chris Fregly](https://twitter.com/cfregly) talked about it [here](https://fr.slideshare.net/cfregly/advanced-apache-spark-meetup-project-tungsten-nov-12-2015) (see slide [56](https://image.slidesharecdn.com/advancedapachesparkmeetup-projecttungsten-nov122015-151113030443-lva1-app6891/95/advanced-apache-spark-meetup-project-tungsten-nov-12-2015-56-638.jpg?cb=1447384119)). 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](http://spark.apache.org/docs/2.0.0/api/R/approxQuantile.html" \t "_blank)* 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**

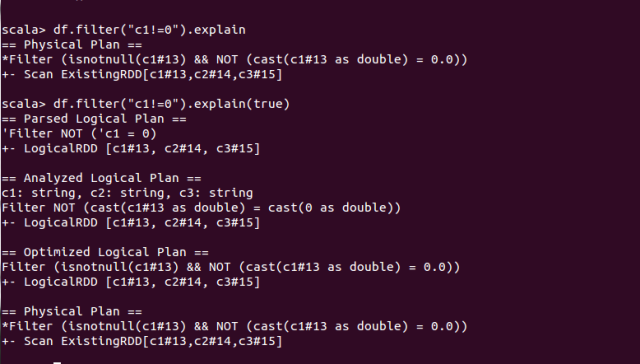
Software Craftsmanship principles obviously apply when writing big data stuff (do one thing and do it well). By splitting UDFs **we are able to use built-in** functions for one part of the resulting code. It also greatly simplify testing.

**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*.

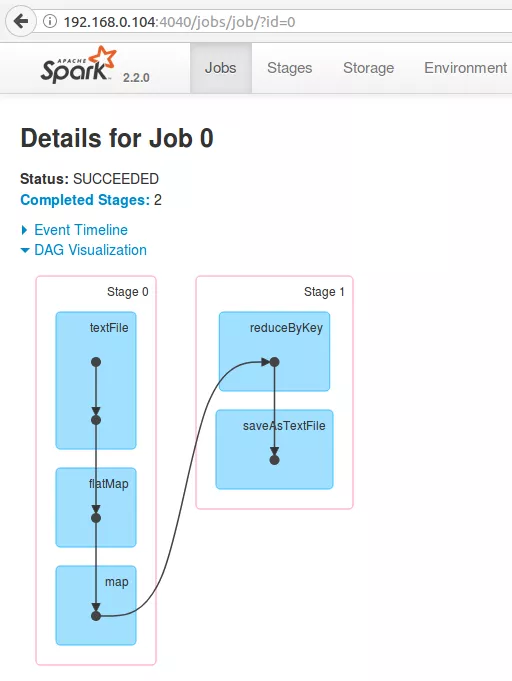
In Spark and more generally in distributed computing, sending data over the network (a.k.a. *Shuffle* in Spark) is the most expensive action. ***Shuffles* are expensive** since they involve disk I/O, data serialization and network I/O. They are needed for operations like *Join* or *groupBy* and happen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



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](https://www.tutorialkart.com/apache-spark/dag-and-physical-execution-plan/)

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](https://issues.apache.org/jira/browse/SPARK-16026)beyond broadcast join selection. It now seems available in 2.3.0, we will have to look at that.

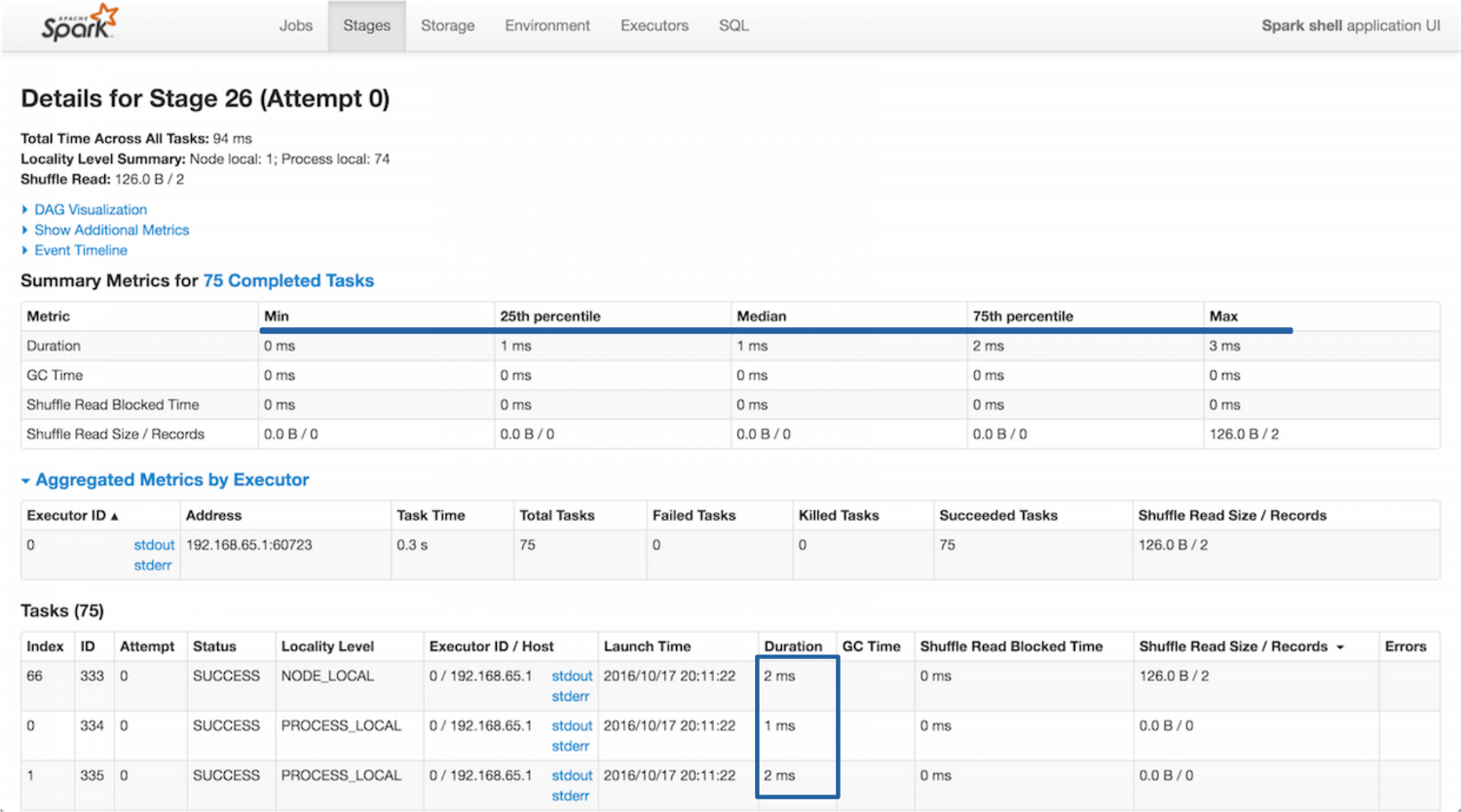
**3- Know your data and manage it efficiently**

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:

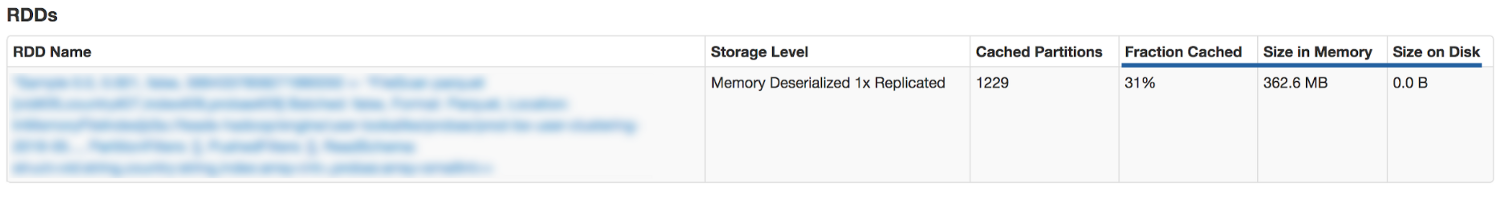


Stages tab example on Spark UI

**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](http://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](https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/internal/SQLConf.scala#L171) 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](https://aws.amazon.com/fr/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](https://medium.com/teads-engineering/real-life-aws-cost-optimization-strategy-at-teads-135268b0860f), 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](https://github.com/nchammas/flintrock" \t "_blank) to directly run our clusters on EC2. Here is an [article](http://heather.miller.am/blog/launching-a-spark-cluster-part-1.html) by [Heather Miller](https://twitter.com/heathercmiller) 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](https://medium.com/@subhojit20_27731/apache-spark-and-amazon-s3-gotchas-and-best-practices-a767242f3d98) 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](https://github.com/rdblue/s3committer), open sourced by Netflix, but we haven’t tested it.

We store our files on S3 using Parquet, a[columnar storage](http://en.wikipedia.org/wiki/Column-oriented_DBMS) 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](https://issues.apache.org/jira/browse/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 JDBCmedium.com](https://medium.com/teads-engineering/spark-from-the-trenches-part-2-f2ff9ab67ea1)

[Lessons learned while optimizing Spark aggregation jobs  
Spark from the trenches — Part IIImedium.com](https://medium.com/teads-engineering/lessons-learned-while-optimizing-spark-aggregation-jobs-f93107f7867f)

**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](https://spark.apache.org/releases/spark-release-2-3-0.html) version brings major features like Kubernetes support and a streaming execution engine catching up with [Flink](https://flink.apache.org/" \t "_blank) 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](https://teads.tv/teads-jobs/) at Teads!

*By*[*Yann Moisan*](https://medium.com/@yamo93)*and*[*Benjamin DAVY*](https://medium.com/@benjamin.davy)

**Bibliography**

* [Advanced Apache Spark Meetup Project Tungsten](https://fr.slideshare.net/cfregly/advanced-apache-spark-meetup-project-tungsten-nov-12-2015), by Chris Fregly — 12/11/2015
* [Working with UDFs in Apache Spark](https://blog.cloudera.com/blog/2017/02/working-with-udfs-in-apache-spark/), by Curtis Howard — 03/02/2017
* [Spark Execution Model](https://www.cloudera.com/documentation/enterprise/5-9-x/topics/cdh_ig_spark_apps.html#spark_exec_model), Cloudera
* [Cassandra write tuning parameters](https://github.com/datastax/spark-cassandra-connector/blob/master/doc/reference.md#write-tuning-parameters), DataStax
* [Apache Spark and Amazon s3 gotchas and best practices](https://medium.com/@subhojit20_27731/apache-spark-and-amazon-s3-gotchas-and-best-practices-a767242f3d98), by Subhojit Banerjee — 18/11/2016
* [A Spark 2.0.0 Cluster Takes a Longer Time to Append Data](https://docs.databricks.com/spark/latest/faq/append-slow-with-spark-2.0.0.html), databricks
* How to tune your Apache Spark jobs [Part 1](http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/) & [Part 2](http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/), by Sandy Riza — 03/2015
* [Spark performance tuning checklist](https://zerogravitylabs.ca/spark-performance-tuning-checklist/), by Taraneh Khazaei — 08/09/2017
* [Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop](https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html), by Sameer Agarwal et al. — 23/05/2016
* [Diving into Spark and Parquet Workloads, by Example](https://db-blog.web.cern.ch/blog/luca-canali/2017-06-diving-spark-and-parquet-workloads-example), by Luca Canali — 29/06/2017
* [Running Spark on YARN](https://badrit.com/blog/2015/2/29/running-spark-on-yarn), by Mahmoud Hanafy — 29/02/2015
* [How to optimize Apache Spark apps](http://rea.tech/how-we-optimize-apache-spark-apps/), by Henri Hu — 08/09/2017
* [Running Spark on a Cluster: The Basics](http://heather.miller.am/blog/launching-a-spark-cluster-part-1.html), by Heather Miller — 09/04/2018

# **5 Key Factors to keep in mind while Optimizing Apache Spark in AWS(Part 1)**

This article aims to help experienced developers with some of the bottlenecks faced while dealing with extreme volume of data with limited resources. It is not about fundamentals and theoretical optimization techniques which are frequently discussed. Suggested solutions( or optimization tricks) are based on inferences drawn from the practical problems faced while optimizing Apache Spark.

#### Long Lineage

Lazy evaluation in spark means, actual execution does not happen until an action is triggered. The types of commands available in spark can be divided into 2 types.

* Actions ( eg. head(), show(), write(), count())
* Transformations (eg. map(), filter(), groupBy(), select())

Every transformation command run on spark gets added to the lineage(explained below) after the syntax check, actual execution happens only when an action based command is run.

**Optimization Trick**: It is not advisable to chain lot of transformation in a lineage, especially when you would like to process huge volume of data with minimum resources. Rather, break the lineage by writing intermediate results into HDFS( preferable HDFS if you have storage available, as writing S3 could be slower)

#### File System Preferences

The types of files we deal with can be divided into two types

* Splittable ( eg. LZO, Bzip2)
* Non- Splittable ( eg. Gzip, Zip)

For the purpose of the discussion, Splittable files means they are parallely processsable in a distributed fashion rather in one machine( non-Splittable).

**Optimization Trick**: If you have a huge file (10gb and zipped) and you try to load into spark, it might just get processed using one node( or executor) if it is not splittable which could be a bottleneck. If you come across such cases, it is a good idea to use s3cmd and move the file from s3 into HDFS and unzip it(If the big file you are referring is in s3). If it is in HDFS, you could unzip it before you load into spark.

Note : We will discuss the columnar file formats in PPD section below.

#### Writing Queries and/or Transformations

The biggest mistake people make in big data systems is, try to “optimize queries” in fact it should be “optimize data”. “Simplicity is the Key”, This is applied to all distributed systems including spark. To apply this in real life, it is advised not to write complex queries in spark, rather try to break it down as much simpler steps as you can. People have a misunderstanding that, more number of steps could increase the processing but, actually not. Spark might internally combine some of the steps and perform at once.

**Optimization Trick**: Always try to break your queries (or transformations) into granular steps instead of writing one big query. Operations chained in spark are different steps ( not a single big query or transformation)

#### Predicate Push Down(PPD)

PPD in simple terms, is a process of only selecting the required data for processing when querying a huge table. eg: If you have a table of 100 columns and you are querying only 10 columns, in PPD data for only those 10 columns are selected for further processing. Another example could be, if there is a filter clause(eg. where clause) in any query, the filter will be applied first to reduce the number of records picked for processing. This significantly improves the performance by reducing the number of records read/write resulting reduction in input/output operation.

Columnar file formats give us a great way of using the power of PPD as it inherently enabled to do so. Some of the examples of Columnar file formats are Parquet, RC or Row-Column, ORC or Optimized Row-Column etc.

**Optimization Trick**:There are two important notes to make here.

* Use Parquet format wherever feasible for reading and writing files into HDFS or s3 as parquet seems to be performing very well along with Spark. Especially, All the intermediate steps that you would like to write data into HDFS so as to break the lineage( As mentioned under optimization trick in Lazy Evaluation)
* Always try to identify the “filters” and try to move it up as early as you can for all your data processing pipeline.

#### Data Skew Checks

Performance of the distributed systems are highly dependent on how much distributed the data is. One way to ensure distribution is to check the number of partitions of a RDD or a DataFrame.

**Optimization Trick**: Do check the number of paritions of the dataframes or RDDs just before you carry out any complex operation. In case you find the number of partitions are too low, it is a good idea to repartition them to increase the number of partitions. you could use the below line of code for checking the number of partitions in pyspark.

df.rdd.getNumPartitions()

#### Conclusions

In Bigdata systems it is advisable to optimize data first before we think about optimizing quries.

The second part of the story is available on the below link. Kindly, give a read and share your feedback.

<https://medium.com/@brajendragouda/5-key-factors-to-keep-in-mind-while-optimising-apache-spark-in-aws-part-2-c0197276623c>

# **Spark troubleshooting from the trenches**

We will continue to dig into some real-world situations that we have dealt with and focus on two topics:

* First, we will see some **operation tricks we actively use for troubleshooting**. At Teads, we embrace the following motto: *You build it, you run it*. We had to make sure that we have the right tools to look at our system’s health and understand what’s going on.
* Then, we will talk about **best practices to use external data sources** in your workflows with JDBC.

**1- Operation tricks**

**Monitoring Spark applications**

Spark includes a configurable metrics system based on the *[dropwizard.metrics](http://metrics.dropwizard.io/" \t "_blank)*library. It is set up via the Spark configuration. As we already are heavy users of **Graphite** and **Grafana**, we use the provided [Graphite sink](https://www.hammerlab.org/2015/02/27/monitoring-spark-with-graphite-and-grafana/).

The Graphite sink needs to be **used with caution**. This is due to the fact that, for each metric, Graphite creates a fixed-size database to store data points. These zeroed-out [*Whisper*](https://graphite.readthedocs.io/en/latest/whisper.html) files consume quite a lot of disk space.

By default, the application id is added to the namespace, which means that every *spark-submit* operation creates a new metric. Thanks to [SPARK-5847](https://issues.apache.org/jira/browse/SPARK-5847) it is now possible to **mitigate the *Whisper* behavior** and remove the *spark.app.id*from the namespace.

spark.metrics.namespace=$name

Going further, we have built an **in-house deploy stack** that will automatically call *spark-submit* using this configuration. As developers, we have nothing to do to enable the monitoring and we are able to manage dozens of jobs.

On our Grafana dashboard (see below), we can choose the job we want to inspect. We especially observe **memory consumption and task-related metrics**.

With one quick look at the number of active tasks per executors, we are able to check if the cluster resources are efficiently used. Here, the visible drop on the graph is due to a *shuffle* and can help to identify bottlenecks.



Task monitoring

We monitor memory issues using two metrics: *PS Scavenge* (Minor GC) and *PS MarkSweep*(full GC). When we observe a simultaneous decrease of active tasks with an increase of full GCs we know that we need to review the sizing of our cluster.



Garbage Collector monitoring

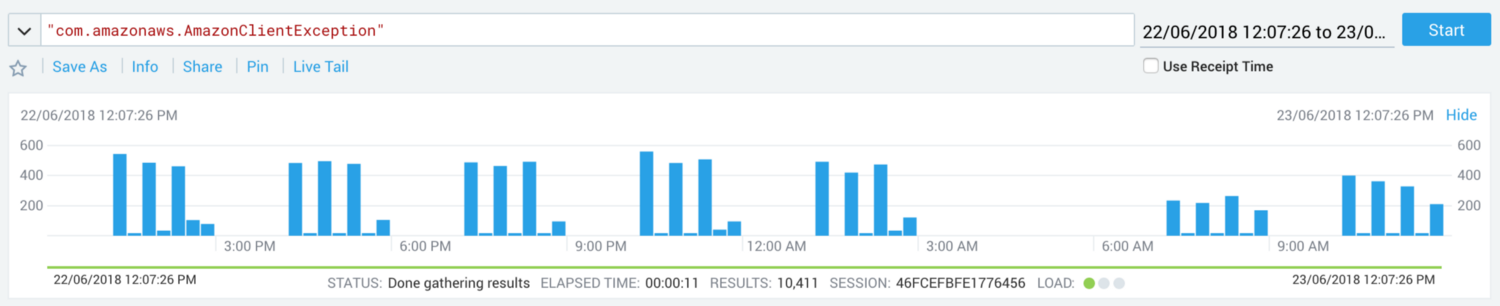
This first set of dashboards is useful to detect memory issues. However, on the CPU side, we could go further by **using flame-graphs** to observe the executors. This could help detect hotspots and faulty instructions. Tools like [Babar](https://github.com/criteo/babar) (open sourced by Criteo) can be used to aggregate Spark flame-graphs.

**Log management**

At Teads, we use [Sumologic](https://www.sumologic.com/" \t "_blank), a cloud-based solution, to manage our logs. With our deploy stack, a [log4j](https://logging.apache.org/log4j/) configuration is generated for all our jobs using the *SyslogAppender*. So once again, there is nothing to do when a developer creates a new Spark job.

A log configuration is enforced for all our jobs to send data to our log collector. When an exception occurs, **we can quickly know how many jobs are impacted**, how many executions of these jobs are running, etc.

In the example below, we can see a read timeout on Amazon S3 that affects a job retrying continuously.



Read timeout on Amazon S3

**Troubleshooting with logs**

Logs are the best troubleshooting material there is but it can be a pain to investigate them.

Let’s dig into a real case study. In this example, one of our jobs was failing. After a quick look in the driver’s log, we can see some errors:

[WARN ] 2018–08–02 22:03:23,672 org.apache.spark.deploy.yarn.YarnAllocator — Container marked as failed: container\_1532943674849\_0820\_01\_000012 on host: ip-10–20–109–19.eu-west-1.compute.internal. Exit status: -100. Diagnostics: Container released on a \*lost\* node  
[WARN ] 2018–08–02 22:03:23,674 org.apache.spark.scheduler.cluster.YarnSchedulerBackend$YarnSchedulerEndpoint — Requesting driver to remove executor 5 for reason Container marked as failed: container\_1532943674849\_0820\_01\_000012 on host: ip-10–20–109–19.eu-west-1.compute.internal. Exit status: -100. Diagnostics: Container released on a \*lost\* node

Apparently, some spark executors died (*Container released on a \*lost\* node*), however, it remains to be explained …

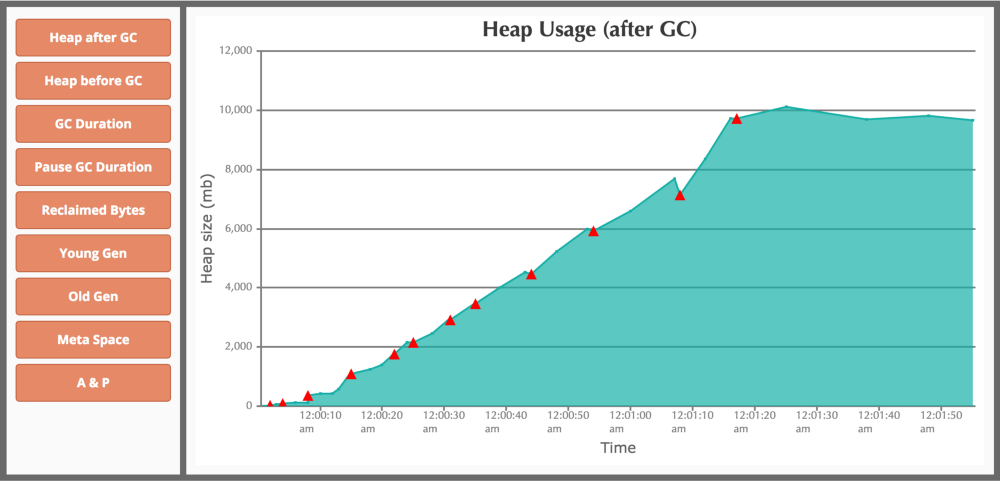
The usual suspect is the memory, **let’s have a look at the GC logs**.

With our deploy stack, [GC logs are activated](https://spark.apache.org/docs/latest/tuning.html#garbage-collection-tuning) on all production jobs as it doesn’t have an impact on performance. Also, our clusters are configured to [archive log files to S3](https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-debugging.html).

aws s3 cp — recursive “s3://teads-hadoop/engine/archive-logs/j-2FUFXCE83DD2B/containers/application\_1532943674849\_0820/container\_1532943674849\_0820\_01\_000038/” ../

Unfortunately, GC logs are stored in what you could call a**cryptic text file**. These are actually pretty hard to analyze without the appropriate tooling.

I have found a very useful service named **[GCeasy](http://gceasy.io/" \t "_blank)** that helps me a lot for this task. GCeasy creates graphs and meaningful aggregates based on your logs. As this data (GC logs) isn’t sensitive for us I find it quite handy.



One of the graphs generated by GCeasy, it also offers the option to export a nice PDF report.

As we can see on the graph, there is a plateau from *12:01:20*, meaning that we fail to free enough memory.

The solution here was to**increase the number of partitions** so that each task has to deal with less memory.

**2- Using external data sources with JDBC**

It’s sometimes handy to be able to work with external data sources. Spark has a built-in support for [JDBC data sources](https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html). At Teads, we use MySQL to store data of reasonable size like:

* Reference data (e.g. currency exchange rates),
* Ad campaign parameters and so on …

Let’s see some read and write use cases.

**JDBC Read**

There are two cases in which we want to read a table

* To join it with an existing DataFrame (so we need to have a DataFrame),
* To use it in a User Defined Function (UDF). In this case, we need to have a plain object, for example, to transform a column (e.g. convert money into a reference currency).

A classic use case is to load a tiny reference table, like exchange rates, in order to use it in a UDF.

Spark  
 .read  
 .jdbc(url, table, props)  
 .collect()

By default, the reading operation happens on a single executor and then a collect operation sends data over the network to the driver.

In that case, it’s faster to directly load the table on the driver with a JDBC library. Coding a Spark job does not mean that we must use Spark to do everything.

**JDBC Read subquery**

Spark SQL has a useful feature that isn’t well-known, allowing to load data with a subquery.

From the Spark documentation we can read:

The JDBC table that should be read. Note that anything that is valid in a FROM clause of a SQL query can be used. For example, instead of a full table you could also **use a subquery** in parentheses.

Here is a working example that we previously used to load the last known exchange rates for a given date.

val table =  
s”””(SELECT dest\_cur, rate  
 |FROM currency\_exchange\_rates  
 |WHERE time = (  
 | SELECT MAX(time) FROM currency\_exchange\_rates WHERE time < ‘${dtf.format(dt)}’  
 |)) AS currency\_exchange\_rates”””.stripMargin

spark.read.jdbc(url, table, props)

***Note****: with Spark 2.4.0, it is even easier, cf*[*SPARK-24423*](https://issues.apache.org/jira/browse/SPARK-24423)

**JDBC Write**

In Spark, there are 4 [save modes](https://spark.apache.org/docs/1.4.0/api/java/org/apache/spark/sql/SaveMode.html): *Append, Overwrite, ErrorIfExists* and *Ignore*.

Let’s consider the following use case: a job needs to replace the entire content of a table. Some tables are read by online services like the buying-engine, one of our services that handles bid requests.

These two modes are interesting for us :

* *Append*: for adding new lines in a table,
* *Overwrite*: to overwrite the **full** table (this could be misleading because existing rows that are not in the DataFrame will be deleted).

***Note****: Updating only a subset of rows is not supported.*

**JDBC Write — Naive approach**

dataset  
.write  
 .mode(SaveMode.Overwrite)  
 .jdbc(…)

Using the overwrite mode, Spark will drop and recreate the table. So all metadata information will be lost (like index or foreign keys). For that reason, we cannot use this mode.

**JDBC Write — A slightly better way**

This approach consists in deleting everything using a standard JDBC client (like *[scalikejdbc](http://scalikejdbc.org/" \t "_blank)*) and then let Spark insert data using a built-in feature.

db.autoCommit { implicit session =>  
sql”””DELETE FROM my\_table”””.update.apply()  
}  
dataset  
 .write  
 .mode(“append”)  
 .jdbc(…)

The main drawback is that**it’s not atomic**. If something bad happens in-between, the table will stay empty.

**JDBC Write — A better way**

Spark has an option to truncate a table.

dataset  
.write  
 .mode(SaveMode.Overwrite)  
 .option(“truncate”, “true”)  
 .jdbc(…)

A little warning, as the truncate operation **requires to have**[**DROP**](https://dev.mysql.com/doc/refman/5.7/en/privileges-provided.html#priv_drop)**privileges**some database users might not be authorized to perform it.

Also, truncate is still not an ideal solution. It looks like a single operation but it’s actually performed in three steps, in the following order:

1. The table is dropped,
2. The dataset is computed,
3. The table is filled up.

Between step 1 and 3, the table is empty. This is a critical issue for online services that need to read these data.

**JDBC Write — Preferred solution**

All examples below are based on the tiny and convenient *scalikejdbc* library.

If the volume of data is reasonable, it’s possible to **collect the whole dataset on the driver** and use a classic JDBC client to perform the job. Hence, all operations will be atomic.

As we can see, it does not even use Spark:

db.localTx { implicit session =>  
 sql”DELETE FROM my\_table”.update().apply()  
 sql”””INSERT INTO my\_table (…) VALUES (…)”””.batch(batchParams:\_\*).apply()  
}

It shows that when using a high-level tool like Spark, we still need to understand what happens under the hood.

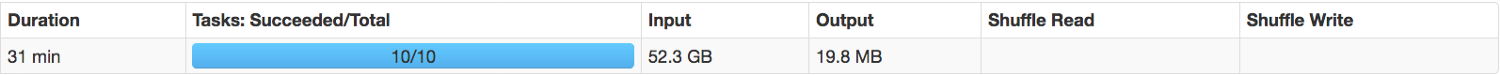
**Coalesce vs repartition**

In the literature, it’s often mentioned that *coalesce* should be preferred over *repartition* to reduce the number of partitions because it avoids a shuffle step in some cases.

But***coalesce* has some limitations** (outside the scope of this article): it cannot increase the number of partitions and may generate skew partitions.

Here is one case where a ***repartition* should be preferred**. In this case, we filter most of a dataset.

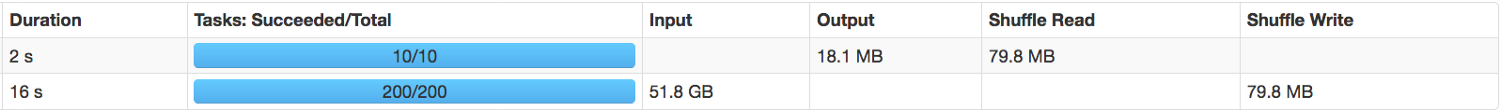
df.doSth().coalesce(10).write(…)



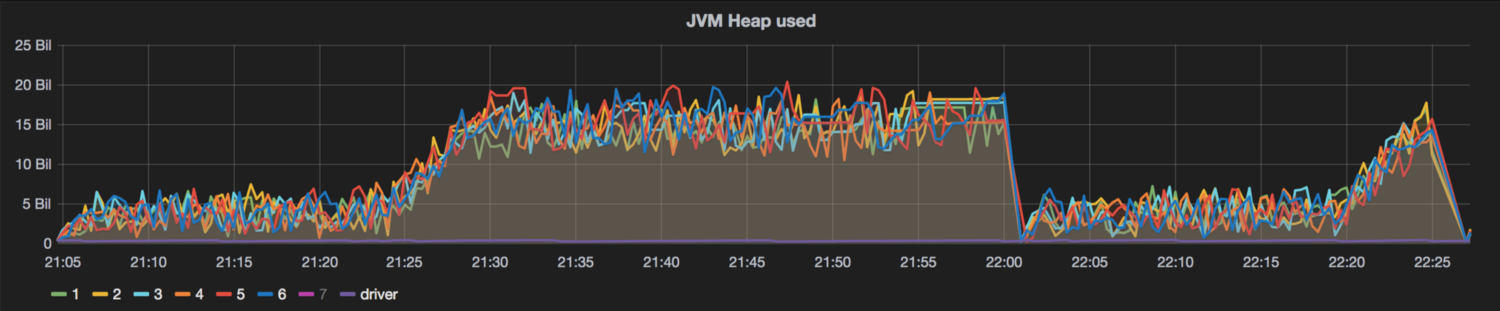
The good point about *coalesce* is that it avoids a shuffle. However, all the computation is done by only 10 tasks.

This is due to the fact that **the number of tasks depends on the number of partitions of the output of the stage**, each one computing a big bunch of data. So a maximum of 10 nodes will perform the computation.

df.doSth().repartition(10).write(…)



Using *repartition* we can see that the total duration is way shorter (a few seconds instead of 31 minutes). The filtering is done by 200 tasks, each one working on a small subset of data. It’s also way smoother from a memory point a view, as we can see in the graph below.



JVM Heap monitoring

**Takeaways**

We have seen how important it is to take into account operation and build tools that ensure all jobs are **started with the right parameters for debugging**.

On external data source management, Spark brings a lot of magic but that **shouldn’t always be taken for granted**. As usual, we can gain more reliability by getting a better understanding of the underlying mechanisms.

*By*[*Yann Moisan*](https://medium.com/@yamo93)*and*[*Benjamin DAVY*](https://medium.com/@benjamin.davy)

* [Tips for using JDBC in Apache Spark SQL](https://medium.com/@radek.strnad/tips-for-using-jdbc-in-apache-spark-sql-396ea7b2e3d3) by Radek Strnad — 08/10/2017
* [Monitoring Spark with Graphite and Grafana](https://www.hammerlab.org/2015/02/27/monitoring-spark-with-graphite-and-grafana/) by HammerLab — 27/02/2015
* Apache Spark Doc — [Monitoring and Instrumentation (Metrics)](https://spark.apache.org/docs/latest/monitoring.html#metrics)
* Apache Spark Doc — [Spark SQL, DataFrames and Datasets Guide](https://spark.apache.org/docs/latest/sql-programming-guide.html#jdbc-to-other-databases)
* [GCeasy](http://gceasy.io/) — GC Log Analyzer

# **Lessons learned while optimizing Spark aggregation jobs**

*In this third article of our Apache Spark series (see*[*Part I*](https://medium.com/teads-engineering/spark-performance-tuning-from-the-trenches-7cbde521cf60)*and*[*Part II*](https://medium.com/teads-engineering/spark-from-the-trenches-part-2-f2ff9ab67ea1)*), we focus on a real-life use case, where we tried several implementations of an aggregation job.*

**Business Context**

At [Teads](https://www.teads.com/" \t "_blank), we distribute ads to over 1.5bn people every month within professionally-produced content.

One of the main components of our platform is responsible for handling *bid requests* (an opportunity to display an ad) and for sending back a *bid response*(the ad to display and the associated price).

An advertising campaign can be set up with delivery constraints :

* target specific users, depending on their geolocation,
* target specific devices, OS, browsers,
* etc.

As a result, ads are filtered according to these requirements.

In order to analyse the ad delivery, we generate a log for each bid request, containing the reasons why an ad was filtered (we also use this log to train our [prediction models](https://medium.com/teads-engineering/ml1-d84f645132c9)). For example, a filtering reason can be geolocation, if an ad targets users from a specific country. For the sake of simplicity, we will use four filtering reasons: from A to D.

This article will focus on **how to implement a reporting job that counts these filtering reasons** for each ad. Such a job can be used to build troubleshooting tools.

**Specification of the job**

**Input data**

Here is the schema of the log.

root  
|-- filtering\_reasons: map (nullable = true)  
| |-- key: string  
| |-- value: array (valueContainsNull = true)  
| | |-- element: string (containsNull = true)

The field filtering\_reasons is a *map* with ad identifiers as keys and arrays of filtering reasons as values.

**Expected output**

Let’s take an example with 2 bid requests.

*Request 1*

* ad1 is filtered by reasons A and B
* ad2 is filtered by reasons A and C

*Request 2*

* ad1 is filtered by reasons A and D
* ad3 is filtered by reason C

scala> df.show(false)  
+---------------------------------------------------------+  
|filtering\_reasons |  
+---------------------------------------------------------+  
|Map(ad1 -> WrappedArray(A, B), ad2 -> WrappedArray(A, C))|  
|Map(ad1 -> WrappedArray(A, D), ad3 -> WrappedArray(C)) |  
+---------------------------------------------------------+

So, ad1 is filtered two times with reason A, one time for reason B (in request 1), never for reason C, and one time for reason D (in request 2).

If we do the same exercise for ad2 and ad3, we see that the expected result is:

+-----+---+---+---+---+  
|ad\_id|A |B |C |D |  
+-----+---+---+---+---+  
|ad1 |2 |1 |0 |1 |  
|ad2 |1 |0 |1 |0 |  
|ad3 |0 |0 |1 |0 |  
+-----+---+---+---+---+

**Technical Solution**

**Preliminary notes**

A variable named allFilteringReasons contains all possible filtering reasons.

**Implementation 1**

The idea is to explode the map to have an ad column and then group filtering reasons by ad with an aggregate function. The desired aggregate function doesn’t exist in Spark, so we have to write a custom one.

Let’s have a closer look, step-by-step.

**Explode —**An explode [method](https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/functions.html#explode-org.apache.spark.sql.Column-) returns two columns when applied on a *MapType* column : one for the key and one for the value. There is also an asfunction made for this specific case, that takes a sequence of aliases.

Here is the resulting DataFrame :

+-----+-----------------+  
|ad\_id|filtering\_reasons|  
+-----+-----------------+  
|ad1 |[A, B] |  
|ad2 |[A, C] |  
|ad1 |[A, D] |  
|ad3 |[C] |  
+-----+-----------------+

**na.fill —**An ad may never have a given filtering reason, resulting in null values. The na.fill replaces all null values by 0.

**Custom UDAF —**For the record, here is the UDAF code (that’s not the aim of the article). The basic idea is to accumulate each filtering reason in a Map whose key is the reason and value is the counter.

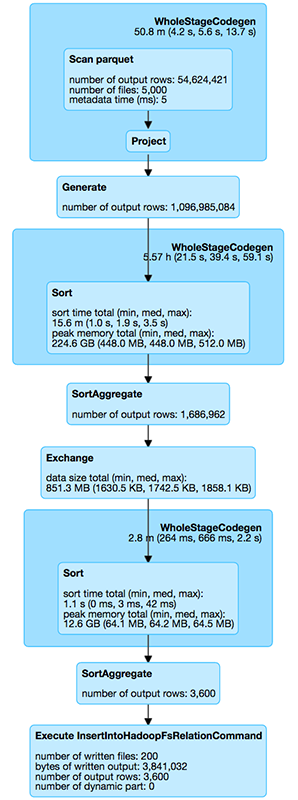
**SQL Plan**

*A note on Catalyst:* When using the DataFrame/Dataset API, a query optimizer called [Catalyst](https://databricks.com/glossary/catalyst-optimizer) is used. This optimizer rewrites the query based on predefined rules.

For performance reasons, Spark tries to group together multiple operators inside a Whole-Stage CodeGen. In order to do that, Spark generates Java code on the fly and compiles it with [Janino](https://janino-compiler.github.io/janino/" \t "_blank) (see [here](https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html) for further details).

On the Web UI, in the SQL tab, we can click on a query to see a graphical representation of the physical plan.

Here is the plan for this implementation:



* The Generate operator corresponds to the explode method in the implementation,
* Generate breaks the WholeStageCodegen in two,
* Exchange corresponds to a shuffle operation between 2 stages,
* SortAggregate appends twice (before and after the shuffle) because Spark performs the aggregation locally on each mapper before sending results to a reducer.

**Implementation 2**

Can we do better?

In this second iteration, the idea is to **avoid using a UDAF**, by transforming each row with a UDF and then use built-in aggregate functions (that could be better optimized by Spark).

If we look at this step-by-step we first have the same explode as in implementation 1.

**User defined function —**For each row of the dataset, filtering reasons are transformed into a Map. Hence, built-in aggregate functions sum can be used.

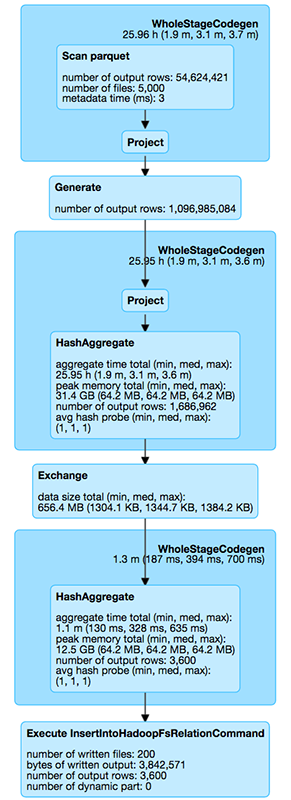
**Aggregation —**All values for a given key must be aggregated together.

* There is a **not so well known** syntax to access a given key on a MapType column $"map.key".
* Another possible way is $"map".getItem(key). *Note:*if the key doesn’t exist in the map, null is returned.

Hence, we can do: sum($"map.key")

**SQL Plan**

The only difference with implementation 1 is that the SortAggregate is replaced by a HashAggregate.



**Implementation 3**

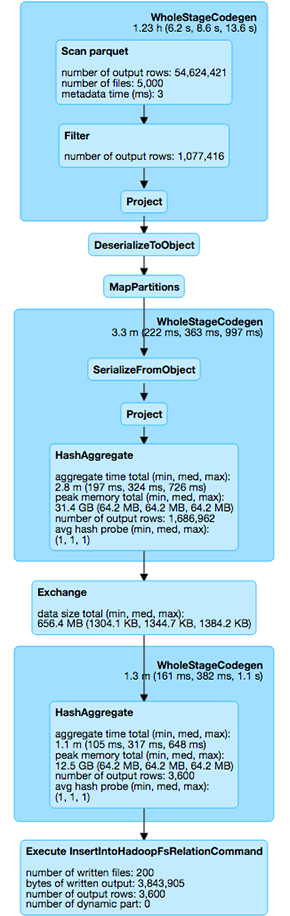
This time we want to **avoid the use of the explode** method by mapping directly on existing partitions.

Here, the aggregation by ad is made manually, inside the loop.

A given ad may appear on multiple partitions, so we need an additional step to aggregate each result on the ad\_id.

**SQL Plan**

The plan is quite different from implementation 1 and 2, with the apparition of new operators :DeserializeToObject & SerializeFromObject



* DeserializeToObject: with the DataFrame API, a lot of operators work on *InternalRow*, the optimized representation of a row into memory. But the MapPartitions operator works on a standard Scala object. We need to use an operator, DeserializeToObject, to convert data from an *InternalRow* to a Scala object.
* SerializeFromObject: this operator is used to go back to the DataFrame world.

**Implementation 4**

We can also embrace a more functional programming style.

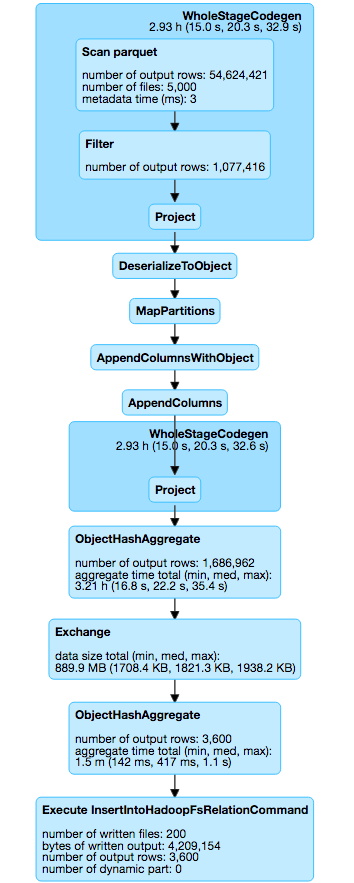
Here, we chain together the following methods :

* flapMap: to replace theexplode
* groupByKey followed by mapValues
* reduceGroups to aggregate values

**SQL Plan**

Similar to implementation 3 with some additional operators.

Interestingly enough, even if we use groupByKey and reduce (*reduceByKey*does not exist), a partial aggregation is done on the reduce side. Cf. [SPARK-16391](https://issues.apache.org/jira/browse/SPARK-16391)



**Other implementations — Array based**

In all previous implementations, counters are stored in a Map. But Map is not a very efficient data structure. It uses a lot of memory, it is not cache friendly and it generates boxing/unboxing in Scala to store Long values.

And given that all keys are known in advance, we can use an array instead. The reason A is at position 0, the reason B at position 1, and so on …

This doesn’t change our previous implementations a lot. For instance, here is the array-based version of our 3rd implementation:

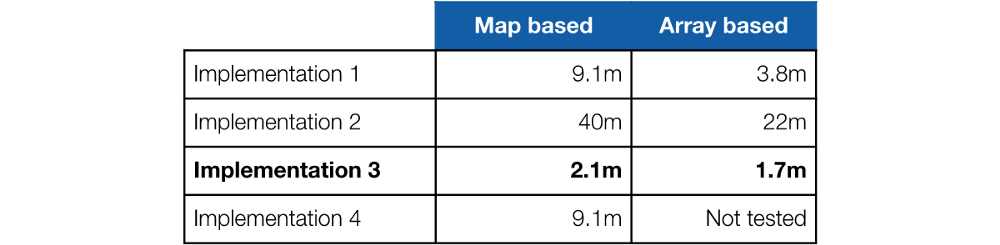
We have precomputed a cache (reasonToIndex) between filtering reason and index.

Hence, lookups in the hashmap are replaced by direct accesses in the array : counter(reasonToIndex(reason))+=1

**Execution time comparison**

The application used for this benchmark processes one hour of our real production log with each implementation. As we can see in the SQL plan, the log contains 50M of input rows and the job produces a 3600 rows output.

This benchmark runs on an EMR cluster of 10 nodes (r3.xlarge), with 10 executors, 4 cores per executor and 20GB of RAM.



All possible implementations need at least one shuffle phase, to put all data for a given ad on the same node. In this case, we have only one shuffle, so our implementations are optimized regarding network exchange.

*HashAggregate* is more efficient than *SortAggregate* because we do not need to sort the data. Nevertheless, implementation 1 (with a *SortAggregate*) is more performant than implementation 2 (with a *HashAggregate*). It’s related to the complexity of both algorithms:

* In impl. 1, only filtering reasons for a given ad are considered (2 or 3 on average), so the map or the array are sparse.
* In impl. 2, all filtering reasons are considered, so this triggers a lot more lookups.

In this use case, using the explode method costs a lot because the explode factor is around 20 (number of output rows in generate / number of output rows in scan, as seen in the SQL plan). This is due to the generation of multiple InternalRow.

Although **implementation 3** needs some deserialisation/serialisation (from [Tungsten](https://databricks.com/glossary/tungsten) to standard objects) it **remains the fastest** (as seen in the SQL plan)

**Conclusions**

1. As seen previously, there are **multiple ways to implement** a given algorithm with Spark.
2. Knowing and **choosing the appropriate data structure** is one of the most important skills for a software engineer and especially in a big data context.
3. In order to improve the performance of your jobs, two important things are required : **knowing Spark execution model** and the **shape of your data**. And it’s worthy because this can lead to significant improvements.

*Thanks to Wassim Almaaoui, Han Ju, Amine Er-Raqabi, Benjamin Davy, Gaël Rico and Robert Dupuy for their help in writing this article.*

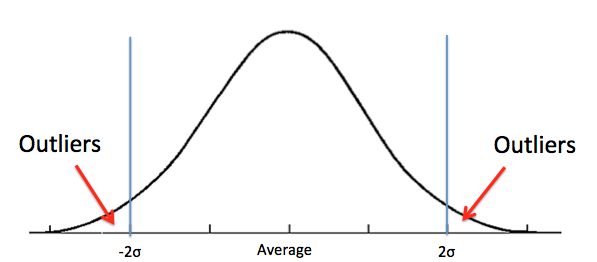
# **Hey SparkSQL, What’s the Average Date?**

[Apache Spark](http://spark.apache.org/) is one of the leading open source analytics frameworks, but it can’t do everything. In this blog post, we’ll look at a few different approaches to computing the *average of date-typed data*, which isn’t natively supported in Spark (as of version 2.3.0). Luckily though, Spark is highly customizable, allowing new analytic functions to be added quite easily.



You may be asking why you’d need to find the average of date values, especially since an *average* (or *arithmetic mean*) is typically an operation on numbers. For example, the average of ***January 7th, 1987***, ***June 23rd, 1994***, and ***December 10th, 1992***, gives you the central date value of ***June 24th, 1991,*** but how is that useful?

One such case is *anomaly detection*, where it’s useful to identify date values falling outside the range of what’s considered “normal”. That is, if we determine the *average* of a series dates, as well as the *standard deviation* of those dates, we can then identify the *outliers* that fall beyond two standard deviations from the average.



Identifying the outliers gives us an opportunity to remove invalid data from our data set, or perhaps to start investigating the root cause of the anomaly. In this scenario, computing the average date from a large data set is important.

In this blog post, we’ll look at two different approaches to computing an average date using the Spark framework. We’ll also discuss some accuracy and performance implications.

**The Problem**

If Spark (particularly SparkSQL) already supported this functionally, we’d be able to compute the average of a series of dates by reading them into a Spark DataFrame (a table with rows and columns), then invoking the avg function on the appropriate column.

In this example, we’ll read the dates from the single-column CSV filedates.csv:

2017-01-02  
2017-03-04  
2017-05-06  
2017-08-01  
...  
2017-10-12

To read this file into a Spark DataFrame, ensuring that the single column of data is interpreted as a Date-typed value, we use the following code:

**import** org.apache.spark.sql.types.\_

// The single column ("datecol") must be interpreted as a Date  
**val** schema = StructType(  
 Seq(  
 StructField("datecol", DateType)  
 )  
)

// Define a new DataFrame, based off the content of the CSV file  
**val** df = spark.read.schema(schema).csv("dates.csv")

// Compute the average of the datecol column  
df.agg(avg(**'**datecol)))

Unfortunately, this simple solution fails with the following error message:

org.apache.spark.sql.AnalysisException: cannot resolve ‘avg(`datecol`)’ due to data type mismatch: function average requires numeric types, not DateType;

It’s clear that Spark’s built-in avg function isn’t designed to support DateTypecolumns, so a workaround is required.

**Approach 1 — Convert to Number, and Back Again**

Given that the avg function is intended to operate on numeric values, our first approach is to translate all the Date values into corresponding Intvalues, representing the number of days since a fixed point in time. We then perform the avg operation, and convert the result back to aDate value.

Unix-based systems use ***January 1st, 1970*** as the point at which everything started (known as the “Epoch”), so we’ll do the same by converting all Datevalues to the number of days since the Epoch. For example, ***January 2nd, 2017*** equates to ***17168,***and ***March 4th, 1812***equates to ***-57646.***

This numeric conversion is possible using SparkSQL’s [datediff](https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/sql/functions.html" \l "datediff-org.apache.spark.sql.Column-org.apache.spark.sql.Column-" \t "_blank)[function](https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/sql/functions.html#datediff-org.apache.spark.sql.Column-org.apache.spark.sql.Column-).

**import** java.sql.Date

**//** Dates are first converted to number of days since this date **val** baseDate = lit(Date.valueOf("1970-01-01"))

// Compute a DataFrame containing the average number of days  
**val** avgDayDataFrame = df.agg(  
 avg(  
 datediff('datecol, baseDate)  
 )  
)

This approach works well, but gives an Int as a return value, rather than a Date.

scala> avgDayDataFrame.show

+-----------------------------------------+  
|avg(datediff(datecol, DATE '1970-01-01'))|  
+-----------------------------------------+  
| 17303.8|  
+-----------------------------------------+

As it turns out, there’s no native SparkSQL function that does the opposite of datediff, to obtain a Date value from our numeric average. The [date\_add](https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/sql/functions.html" \l "date_add-org.apache.spark.sql.Column-int-" \t "_blank)[function](https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/sql/functions.html#date_add-org.apache.spark.sql.Column-int-) looked like it might work, but instead requires a constant Intnumber of days, rather than taking the Int result from a DataFrame.

Our solution is to extract the average value from the first column of the first row of the DataFrame, and then explicitly create a new Date object (which expects the number of milliseconds since the Epoch). Note that we use Longarithmetic here, to avoid exceeding the 2³¹ limit of Int values.

// Trigger query (by running collect), and extract a native Long  
// from the 0th column and 0th row of the DataFrame.  
**val** avgDay = avgDayDataFrame.collect()(0).getDouble(0).toLong  
  
// Now convert the number of days back to a Date type. The Date  
// constructor requires the number of milliseconds since 1970-01-01.  
**val** avgDate = **new** Date(avgDay \* 24 \* 60 \* 60 \* 1000)

This works, but isn’t very elegant, particularly since the final conversion to Date is done outside the context of Spark DataFrames. We therefore can’t do additional DataFrame processing in the same Spark query. The solution is to encapsulate those last few lines in a Spark UDF (User Defined Function).

**import** org.apache.spark.sql.expressions.UserDefinedFunction

// define a function that takes an Int, and returns the Date  
**val** daysToDate: Int => Date = { days =>  
 **new** Date(days \* 24 \* 60 \* 60 \* 1000)  
}

// convert this to a UDF-based function that uses Spark's Column  
// data type as the input and output.  
**val** daysToDateUDF: UserDefinedFunction = udf(daysToDate)

A Spark UDF is essentially a function that accepts a Spark SQL Column-typed value as input, and returns a Column-typed value as output. This allows the function to be used entirely within a Spark query.

The new Spark query, returning a Date value is now:

**val** avgDayDataFrame = df.agg(  
 daysToDateUDF(  
 avg(  
 datediff('datecol, baseDate)  
 )  
 )  
)  
**val** avgDay = avgDayDataFrame.collect()(0).getDate(0)

This gives us exactly what we need. Note that Spark UDFs are [often reported to be inefficient](https://medium.com/@mrpowers/spark-user-defined-functions-udfs-6c849e39443b), particularly because the Spark SQL optimizer is unable to understand them, and therefore unable to optimize them. In our case, we’re only running the UDF once per DataFrame (not once per row), so the performance impact should be minimal.

**Approach 2 — User Defined Aggregate Functions**

An alternative approach is to define a [User Defined Aggregation Function](https://docs.databricks.com/spark/latest/spark-sql/udaf-scala.html)(UDAF). Whereas a regular UDF acts on a single table cell, a UDAF operates on a full column to produce a single aggregated value.

Here’s how an avgdate function would be used in a Spark query:

**val** avgdate = **new** AvgDateUDF

**val** avgDayDataFrame = df.agg(avgdate('datecol))  
**val** avgDay = avgDayDataFrame.collect()(0).getDate(0)

This syntax is much more readable than the previous example, given that you’re calling the avgdate function on a Date column, and getting back a resultingDate value.

To define a Spark UDAF, we must extend the UserDefinedAggregateFunctionclass and override the class members. The key members to be overridden are:

* inputSchema — Defines the type of values that UDAF can operate on (that is, Date values).
* bufferSchema — Defines the intermediate counters used during the aggregation. In this example, we track the count of the number of date values we’ve seen, as well as the runningtotal of the dates.
* dataType — Defines the type of the output data, in this case DateType.
* initialize() — A method setting the counters to their initial values.
* update() — A method called to add each new Date value to our intermediate counter values. Note the special handling for null field values.
* merge() — Given that Spark is a distributed analytics framework, this method joins together the counters from different Spark partitions that were potentially executed on different compute nodes.
* evaluate() — Converts the intermediate counters into a final Date value. This is done by simply dividing the total by the count, and then converting to a Date type.

**class** AvgDateUDF **extends** UserDefinedAggregateFunction {  
  
 **val** *BaseDate* = Date.*valueOf*(**"1970-01-01"**)  
  
 *// each value being aggregated has this type* **override def** inputSchema: StructType =  
 *StructType*(*StructField*(**"dateValue"**, DateType) :: *Nil*)  
  
 *// intermediate values used during aggregation* **override def** bufferSchema: StructType = *StructType*(  
 *StructField*(**"count"**, LongType) ::  
 *StructField*(**"total"**, LongType) :: *Nil* )  
  
 *// output type of the aggregation* **override def** dataType: DataType = DateType  
  
 *// This aggregation always returns a consistent output,   
 // given a consistent input* **override def** deterministic: Boolean = **true** *// Initialize our internal counters.* **override   
 def** initialize(buffer: MutableAggregationBuffer): Unit = {  
 buffer(0) = 0L  
 buffer(1) = 0L  
 }  
  
 *// Update our counters with a new data value.* **override   
 def** update(buffer: MutableAggregationBuffer, input: Row): Unit = {   
 **val** thisDate = input.getAs[Date](0)  
 **if** (thisDate != **null**) {  
 buffer(0) = buffer.getAs[Long](0) + 1  
 buffer(1) = buffer.getAs[Long](1) +   
 thisDate.toLocalDate.toEpochDay  
 }  
 }  
  
 *// merge counters from two different Spark partitions* **override   
 def** merge(buff1: MutableAggregationBuffer, buff2: Row): Unit = {  
 buff1(0) = buff1.getAs[Long](0) + buff2.getAs[Long](0)  
 buff1(1) = buff1.getAs[Long](1) + buff2.getAs[Long](1)  
 }  
  
 *// Return the final value, as a Date* **override def** evaluate(buffer: Row): Any = {  
 **val** avgDays = buffer.getAs[Long](1) / buffer.getAs[Long](0)  
 java.sql.Date.*valueOf*(LocalDate.*ofEpochDay*(avgDays))  
 }  
}

Note that for performance reasons, we access the intermediate counter variables as buffer(0) and buffer(1), rather than using their symbolic "count" and "total" names.

**Arithmetic Overflow**

Even if you’re not a Scala expert, you can hopefully get the gist of the previous code. That is, initialize a counter to 0, and a sum to 0, and then for every new date value, add the number of days (since the base date) to the sum, and increment the counter. Finally, divide the total sum by the count of items seen.

One limitation of this approach is Arithmetic Overflow. That is, the **total**variable has type **Long**, implying it has a maximum value of 2⁶³-1 (or 9,223,372,036,854,775,807). That’s a pretty large number, but it’s still possible to overflow that data type and have it wrap around to zero. If that was to happen, we’d get a totally incorrect result.

In reality though, this is unlikely to happen with the **Long** data types (it would definitely be a problem with **Int**). Given that we’ll likely be dealing with *recent*dates (that is, near to the year 2018), most of the numbers we add to will be around 17,000 (days since 1970). We’d therefore need to find the average of 500 trillion date values before overflow would happen. It’s probably not worth worrying about this case.

In fact, looking at Spark’s avg function, it uses either Double data type which can reach 10³⁰⁸, or the BigDecimal data type, which can be arbitrarily large (depending on your RAM). Clearly this is not a problem for most Spark users.

If we wanted to be really paranoid, there are ways to avoid overflow by either dividing the data set into equal-sized data sets, and then averaging the averages. Or perhaps use an approach of [iteratively refining the average](http://www.heikohoffmann.de/htmlthesis/node134.html). We’ll leave those solutions for another day.

**Performance**

Finally, let’s get a rough indication of the performance of these different approaches. It’s interesting to measure performance, since User Defined Functions are [reportedly slower](https://issues.apache.org/jira/browse/SPARK-14083) than using native Spark functions which are handled better by Spark’s Catalyst Optimizer. Although our avgdate function is easier to use in queries, it might just be slower.

In our tests, we used Amazon EMR-5.13.0 (with Spark 2.3.0) with one master and two core nodes of type m4.2xlarge (8 CPU and 32GB of RAM). The input data set was 10M rows of randomly-generated data in CSV format, with each row having 100 columns. The test case involved computing the average date for a particular date-typed column.

For each test case, the result was computed six times, with the data from the first test run being discarded (to ignore the impact of cold caches). The reported result is the average duration (in milliseconds) of the remaining five test runs.

* Base Case (675ms — StdDev 41ms) — The standard Spark **min** function was used as an indication of how fast native Spark functions could read data, therefore defining a base case scenario to compare against.
* Approach 1 (947ms — StdDev 13ms) — This is our first approach of computing theavg of datediff.
* Approach 2 (1390ms — StdDev 34ms) —Our second approach, using the avgdate user defined aggregation function.

To eliminate the cost of reading the CSV file into memory (the source data resides in Amazon S3), the .cache() directive was used to pin the data into RAM. Therefore, the duration measurements are purely the time required to scan the date column and perform the averaging operation.

As you can see, our second approach takes almost 50% longer to compute the average date, compared to the first approach. It also takes twice as long as our base case of computing the minimum date value. Although some amount of code optimization is likely possible, our user-defined aggregation function is clearly not the best approach, even though it makes the code more readable.

**Conclusion**

Apache Spark is an excellent general-purpose framework for performing data analytics. Even though it comes with a variety of built-in analytic functions, it’s sometimes necessary to implement your own functions. Spark SQL provides a convenient mechanism for defining both cell-based UDFs and column-based UDAFs, making Spark queries easier to construct. Initial tests indicate that user-defined functions are less performant than native Spark functions.