# Intro To Tez Job Optimization: Making Pig Fly

Joshua Martell and Lucy Yang

**Abstract**

Hadoop grid utilization is up. More users, more jobs, and more data mean it is imperative your project's jobs are running efficiently in terms of memory, CPU, storage, namespace, and runtime. This guide will clarify concepts and discuss tools to help users understand and improve Pig jobs running on Tez. It explains the resources Doppler measures comprising total cost of ownership (TCO) and the accompanying Doppler reporting tools. It discusses memory and CPU metrics and the settings as a way to optimize cost. Runtime is also important, affecting SLA and response time to incoming events. The guide discusses identifying and improving job skew and the tradeoffs available when tuning runtime. Finally, the case study reviews a key Pig job from the Cardinal One Mobile ad pipeline. These tools and setting heuristics are applied to this job in several steps, making changes that have a significant impact on the TCO (45% of original) and runtime (24% of original).

## 1. Introduction

Perhaps you have noticed your job queues are always over capacity, or your product manager has asked for new features that will slow down the pipeline. Maybe, your input data has grown faster than your budget, or your hardware is still on order. Whatever the reasons, tuned jobs have a shorter runtime and require fewer resources meaning less queue contention, fewer or easier approvals at CAR and better SLAs. But how does one get started? Hadoop and ETL tools like Pig have many settings and counters, and finding the key options to address performance issues can be difficult. We hope this tutorial to the key Hadoop resources and settings, along with the Pig case study, give the reader some confidence to review their jobs and experiment. The **mapreduce.*** settings are applicable for MapReduce, Pig, and Hive jobs. Some Pig specific (**pig.***) settings commonly used by the authors are also introduced. Generally, these have Hive equivalents, but are not discussed.

## 2. Background

There are several types of resources in Hadoop. The most important ones are inputs to the so called Total Cost of Ownership computation. TCO is an estimated dollar cost for a job or project on Hadoop based on the resources used. It has four components: memory in GB-Hrs and CPU in vCore-Hrs which are consumed and reported by running jobs and GB storage and NameNode which are related to file storage. All of these have quotas associated with projects, some with hard limits (storage and namespace). We will briefly introduce all of them here.

Hadoop's Distributed File System (HDFS) has two major components, storage nodes which keep blocks of data and the NameNode which tracks file names and corresponding block locations. Storage quota comes from this pool of storage space across the servers. The NameNode has a complete map of the HDFS file system in memory, including files and directories and thus a limited number of entries will fit. As more files are created, memory is used tracking the names and block locations. Large files are preferred since fewer file entries will need to be stored. HDFS is full when either the storage is out of space for blocks or the NameNode is full of entries, whichever comes first. These two metrics, storage and namespace (count of files and directories), are tracked for projects and when exhausted, more must be requested and purchased through Doppler and CAR before additional files can be written.

The next two metrics are related to running jobs on the Hadoop cluster. CPU and memory are consumed together by running jobs, but tracked separately.

Memory is a measure of the time reserved memory was held on a node. For example, a job used 10GB of memory for 2 hours. This would be said to use 10GB * 2 Hr = 20GB-Hrs of memory. If you request 10GB, but only use 6GB, your cost is still for 10GB because that memory was reserved for your job alone. It's interesting to note this metric value is the same if you used 20GB for 1 hour, or 1GB for 20 hours.

The final metric is CPU or vCore-Hrs. A vCore is 10% of a server CPU. If you used 20 vCores for 2 hours, you would have used 20 vCores * 2 Hrs = 40 vCore-Hrs. Similar to memory, we can also use 160 vCores for 15 minutes for the same cost. We will discuss leveraging this to adjust the runtime of jobs later in this paper.

Doppler has tools and dashboards for tracking these resources at the macro level. Usage for all four resources are reported in your project page[1] along with the TCO cost per month. Doppler reports average, 2 standard deviations, and max measured values for memory and CPU to help you understand the variance of your usage. Storage and Namespace have hard limits and creation of new files will be prevented if either are exhausted. Memory and CPU resources are more forgiving and bursts are allowed based on fairness to other users and availability on the cluster.

Reporting for individual jobs can be viewed in the Hadoop Apps[2] dashboard. The dashboard shows memory and CPU utilization in GB-Hrs, vCore-Hrs, and TCO dollars as well as duration for each job submitted. This report updates in near-real time and is very useful to track progress while experimenting with setting changes.

## 3. Optimizing Jobs

Optimizing jobs means making sure the memory, CPU, storage, and namespace are being used efficiently. If we can perform the same tasks using fewer resources, we can improve throughput or runtime. We will discuss each resource in the TCO calculation in detail and common issues and improvements. We will also discuss ways to improve job duration and possible tradeoffs with resources to complete jobs faster.

### 3.1. Memory

Optimizing the memory settings is often the easiest way to improve the TCO for a job. Other resources have some setting based controls, but often require more structural changes. Many of the Hadoop clusters are memory constrained and requests for additional memory quota are unlikely to be approved without waiting for more physical hardware to arrive.

Memory and CPU are allocated for containers which are used to execute the JVM process or task that does the actual work in your job. The JVM uses more than just heap memory (Fig 1), especially if you use native code or buffers, or subprocesses. The total process memory used by your task is reported in the PHYSICAL_MEMORY_BYTES[3] counter. This is reported in the Tez UI under Vertex Counters. The values don't necessarily represent the smallest values that would work for your job, just the amount used; the JVM will often spread out if free space is available. The size of the container is set by **mapreduce.{map,reduce}.memory.mb** and the maximum heap size is controlled by **mapreduce.{map,reduce}.java.opts** using the -Xmx setting.

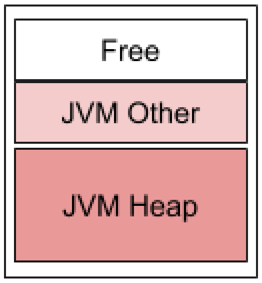


Fig 1 Container memory

Tez jobs typically use both map and reduce tasks, both with independent settings. Reduce tasks are those with a SHUFFLE_BYTES counter and use the reduce settings. Otherwise, the task is a mapper and will use map settings. Mappers read data directly from HDFS at the beginning of your job. As maps and reduces

[1] https://doppler.cloud.corp.yahoo.com:4443/doppler/hadoop/environment/762

[2] https://doppler.cloud.corp.yahoo.com:4443/doppler/tools/hadoop-apps

[3] http://johnjianfang.blogspot.com/2015/07/hadoop-two-mapreduce-task-counters-part.html

perform different functions, their memory usage can differ and they should be tuned separately.

In addition to using Doppler Downsizer[4], the Tez UI can be used to quickly see if a container size is a good fit for the tasks running in it. Select the vertex and sort the PHYSICAL_MEMORY_BYTES to see the largest value. If this is less than the container size, you can generally shrink the container and heap settings in tandem. Be sure to determine if vertex is a map or reduce task so you change the appropriate settings. The defaults are a 1.5GB container and a 1GB heap (-Xmx1024m). This 512MB gap is usually enough for the "JVM Other" portion regardless of heap size, so the container size should typically be JVM Heap + 512MB. Also note containers are allocated in multiples of 512MB, rounding up as necessary. When using a 1GB container, 640MB/1024MB heap/container setting is common.

A more thorough examination of the heap can often be used to yield more memory savings. The JVM will use more memory when plenty is available, throwing off the physical bytes counter. The container GC log can be examined using a tool like GCViewer[5] to see exactly how the heap memory is being used. You may see multiple tasks start and finish in the log as the container and JVM are reused. Objects remaining after a full GC will likely live until the task finishes. Adding 256MB for temporary objects to this heap size after a full GC is typically enough to run the task efficiently and may allow you to shrink your containers even smaller. However, if GC_TIME_MILLIS starts to grow to more than 5-10% of CPU_MILLISECONDS, you may be trading memory for CPU. You may also be causing data spills to disk as memory pressure increases (duration increase). Setting changes should always be confirmed using full data volume before applying them in production.

## 3.2. CPU

CPU is difficult to optimize with setting changes alone because it is related to the actual amount of work performed. Unlike memory, vCores only report what is used, not what was requested, and to affect change, we need to actually do less work. Changing compression codecs, LZ4 for quick and GZIP (or emergent ZSTD), for output can change CPU load, but will also affect other metrics such as storage space. Avoid repeated or unnecessary computations through caching or pruning fields. Combiners are used on some types of jobs to reduce shuffle data and downstream computation. If the combiner ratio (COMBINE_OUTPUT_RECORDS / COMBINE_INPUT_RECORDS) is more than 50%, try disabling them using **pig.exec.nocombiner=true** and evaluate performance. The combiner optimization incurs CPU overhead that is not recovered when the records are mostly unique.

## 3.3. Storage

Storage is likewise difficult to improve with setting changes alone. Adjusting retention (how long job output is kept in HDFS), removal of unused fields, or even obsolete jobs is often the best way to shrink the project storage footprint. One easy setting change is to ensure output data is being compressed using **mapreduce.output.fileoutputformat.compress=true**, which is surprisingly **false** by default. Compression codec was discussed in the CPU section as a tradeoff with storage space. Storage is an ongoing cost each month where CPU is a one time cost. If the data has long retention, consider stronger compression. Binary output formats such as Avro or ORC typically compress better with the same codec due to how they organize and represent the data and have other significant advantages such as schemas and splittable files (can be divided into pieces for different tasks).

## 3.4. Namespace and Parallelism

Namespace, the number of files and directories, has a few more options for improvement than storage. If the total output size stays constant, we can reduce the namespace usage by making fewer, larger files. If the size of output files from your job are smaller than 128MB, consider making changes.

The number of files created by a job is controlled by the parallelism of the output vertex in the job DAG. Parallelism is either set explicitly for certain operators in Pig such as joins and group by operations or

[4] https://doppler.cloud.corp.yahoo.com:4443/doppler/tools/downsizer

[5] https://github.com/chewiebug/GCViewer

controlled automatically. Reducer output vertices use **pig.exec.reducers.bytes.per.reducer** and **pig.exec.reducers.max** to determine the auto-parallelism. The upstream OUTPUT_BYTES counter is divided by the bytes per reducer setting to determine the number of tasks. Automatic parallelism is more flexible and scalable because it grows and shrinks with input volume changes, but is sensitive to aggregation ratios. Note this setting affects all auto-parallelism throughout the job, not just the final output.

Sometimes we have a special case where no reducer is necessary. This is called a "mapper only" job where the output is created by map tasks, not reducer tasks. These tend to be very efficient, but different settings are used to control the parallelism of the mappers which are always automatic. The settings **mapreduce.input.fileinputformat.split.{minsize,maxsize}** can be increased to give each task more input data and thus use less parallelism[6]. Generally, both should be set to the same value. Pig also has a setting[7] to combine small input files into work for a single task (split): **pig.maxCombinedSplitSize**. Typically, this should be set to the same value as the minsize setting. Note for input from HDFS, *.gz files cannot be divided into pieces, only combined with other files when forming a split. This may result in some tasks getting more input than the configuration properties would suggest. *.bz2 files support splitting and have better compression but are much slower to write. Alternatively, you may want to switch to a format such as Avro or ORC which support splitting independently from compression. Counters SPLIT_RAW_BYTES (input bytes assigned to task) and HDFS_BYTES_READ (input bytes actually read) can give more insight to task input size. Columnar formats like ORC that can prune unused columns in the data, these may not match. One final way to improve namespace usage is to HAR[8] the data. It will remain readable via the har:// scheme, but will appear as a directory of large files in HDFS.

[6] https://stackoverflow.com/a/19196022

[7] https://pig.apache.org/docs/r0.17.0/perf.html#combine-files

[8] https://hadoop.apache.org/docs/current/hadoop-archives/HadoopArchives.html

## 3.5. Job Duration

Often runtime or duration of a job is as important as the resource usage. Delays in processing can be costly in missed opportunity, delayed reporting, and slow feedback cycles. Fortunately, improving runtime is often possible without increasing TCO. We will discuss task skew and manipulating parallelism.

Task skew happens when a few tasks in the vertex has more work to do than the other tasks, delaying vertex completion. Tiny tasks cause little trouble as container reuse minimizes the overhead of startup and TCO impact. Large tasks however delay completion and generally prevent the next vertex from starting. This is not necessarily inefficient from a TCO perspective as the work would need to be done somewhere, but it adversely affects runtime of the job. To identify task skew, observe each vertex in the Tez Swimlane tab. Clicking on each one, the vertex dashboard gives the longest and shortest running tasks. You can also see the distribution by sorting the SUCCEEDED task durations in the Task Attempts tab. Additionally, you can check if SHUFFLE_BYTES_DECOMPRESSED for reducer tasks is comparable to others. If not, you may have a GROUP BY or JOIN on a NULL or other default key. If the offending key cannot be filtered out, consider using a "skewed" or "replicated" (in-memory) join type in Pig[9]. This will distribute this key across more than task and improve runtime at the cost of additional complexity in the job.

If skew is not a problem, increasing the parallelism may be used to reduce the runtime. The settings discussed for changing parallelism for namespace can be used to increase parallelism and create more, smaller tasks. For instance, 100 ten minute tasks can be changed to 200 five minute tasks by dividing up the input into smaller splits. Use the Tez UI Swimlane tab to find which vertex is most impacting runtime. Target this for parallelism adjustment, keeping an eye on the TCO metrics that may be affected. Mappers generally do not

[9] https://pig.apache.org/docs/r0.17.0/perf.html#skewed-joins

affect namespace usage and may be easier to adjust. In either of these cases, one should also set **mapreduce.{map,reduce}.speculative=true** to limit the effect of a bad node. This starts parallel speculative attempts along side if the task appears to be running slowly. Whichever attempt finishes first wins and other attempts are killed. If the code interacts with an external service, this may have unintended consequences, as repeated evaluations are expected. For instance, sending multiple emails or re-registering an ad would be undesirable.

## 4. Case Study

Here we review a job from the start of the Cardinal Ad data pipeline that powers One Mobile. The runtime was long and affecting the SLA of the rest of the pipeline. We found several setting changes and a logic change leading to improvements in the TCO and duration of the job. These changes were originally enacted in two phases, but are combined here for simplicity.

The job reads events totaling 70GB, delivered during a 5 minute interval from Data Highway in Avro format. The records represent ad auctions from One Mobile (Fig 2). First, 800MB of dimension data is used to annotate the records using Aviary[10] MDQLookup, a memory efficient lookup table. The events are then split into records with and without uids (MDQLookup & Split by uid). Records containing uids are grouped and counted (Count by uid). This is joined to the original record so each row includes a count of records with the same uid (Annotate). Records without uid have count 0 appended. The two sets of data are unioned and passed to a UDF for traffic protection purposes and stored. Pig is clever and writes the records without uid directly from the "MDQLookup & Split by uid" mapper. The other records must follow the more complex path.

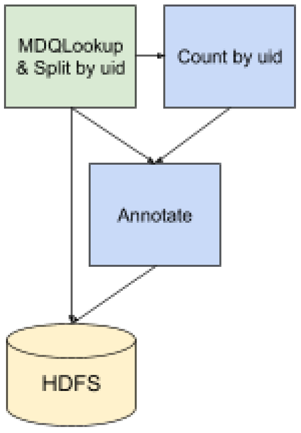


Fig 2 Vertex Diagram, green: map, blue: reduce

[10] https://git.ouroath.com/aviary/udfs

The following tunings were performed on the Jet Blue cluster with a fixed data input. The baseline settings and resource usage including TCO can be seen in the optimization worksheet[11]. The final optimization results in comparison to the original job can be seen in the conclusion.

The original container and heap settings were copied from similar pipeline jobs. To determine the actual memory needs, we examined the PHYSICAL_MEMORY_BYTES for all tasks from the initial run. The largest mapper task used 1.9GB and largest reducer was 1.4GB, far less than the 4/6GB container sizes. After checking the GC logs for the heap after full GC, we changed both mapper and reducer memory (1.0/1.5GB containers and 640/1024MB heap settings).

Next, we examined the namespace usage. Looking at the output directory, the job produced 917 files with most only 47MB, smaller than the desired 128MB. Due to the splitting of data between records with and without uids, the final output file count = mapper task parallelism + reducer task parallelism so we needed to reduce parallelism overall to produce bigger files and reduce namespace usage. By looking at the runtime of mappers in the Vertex Swimlane, we found the longest task took around 1m30s, so we could easily double the input size of 128MB. We set **mapreduce.input.fileinputformat.split.{minsize,maxsize}** and **pig.maxCombinedSplitSize** to 256MB to increase the input data size. We also adjusted the input size for reducers by increasing **pig.exec.reducers.bytes.per.reducer**.

For the first round of tuning with above setting changes, memory was cut to a quarter and namespace usage cut in half. There is often a tradeoff between runtime and

[11] https://docs.google.com/spreadsheets/d/1tAQjGdMHRTRjrAXrh9G3O4nxEsBVxzJjO0-l_uzR5_I/edit?ts=5d3b590c#gid=0

namespace usage. It only produced 390 files, but our runtime increased from 16 to 19 minutes.

Finally, with resource usage under control, we turned to address the runtime. From the Vertex Swimlane, we found the Annotate vertex took the majority of the time compared to the other vertices. Looking at all durations for that join vertex, one task was consuming much more data and ran much longer than any other task. It was clear we had skewed data. We first tried a skewed join instead of using the default hash join type. Pig "...computes a histogram of the key space and uses this data to allocate reducers for a given key."[12] More than one reducer is allocated for the skewed keys, distributing the burden and improving skew.

Another thing we noticed this round was the combiner was not effective (COMBINE_OUTPUT_RECORDS / COMBINE_INPUT_RECORDS) = 70%, so we disabled it off to save CPU.

The second round with above setting changes saw the runtime reduced to around 5 min at the cost of more resources, including producing 1100 small files. Relying on Pig's skewed join helped runtime but was not ideal as it increased TCO. A different route to solve the skew was necessary, by changing the logic. We investigated the data and found one particular uid was very dominant. Do-Not-Track is an iOS feature that causes all users enabling that feature to have the same uid. From our business logic perspective, we can treat these like the records without uids, and eliminate the skewed data from the join. For the third and final test round, we applied the above logic for Do-Not-Track uids and reverted to the default hash join. The resource usage were better than the previous tuning results while the runtime was just under 4 min, a 4x improvement in runtime. Full results can be seen in the conclusion.

## 5. Related Work

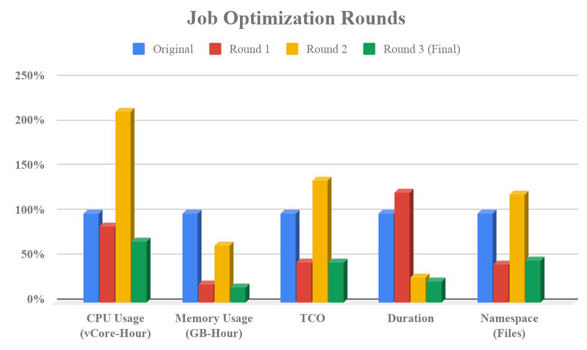
The Aviary group works to improve synergy across ad pipelines through shared code and tools. We mentioned MDQLookup for efficient in-memory annotations, similar to replicated joins, but with less memory overhead. The CDB variant is disk based for even lower memory footprint. These tools also include an indexed join framework[13] used by many ad pipelines for primary/secondary joins. It is much more efficient than a hash join when only a small fraction of the left table is needed to join with the right table through indexed lookups. Future guides may introduce more Aviary tools and include Hive specific settings and optimizations.

In terms of job optimization, Dr Elephant[14] is a tool that evaluates job performance and makes suggestions for improvement, similar to Downsizer. It connects to resource manager and history server to retrieve metrics and check for problems based on pluggable hursitics. It is unknown if the software can scale to handle clusters of our size.

## 6. Conclusions

In conclusion, we have introduced the resources used to compute TCO (memory, CPU, storage, and namespace), discussed common ways to improve each, and how they interact with each other. We discussed identifying job skew, and eliminating or mitigating its effects using skewed joins in Pig.

In the case study, we introduced an important job in the Cardinal One Mobile pipeline, and discussed each step taken to improve it. We adjusted memory settings and parallelism to decrease TCO, then improved runtime by eliminating skew. The final job uses 17% memory, 67% CPU, and 48% namespace of the original job, while slashing runtime from 16 min to under 4.



12 https://pig.apache.org/docs/r0.17.0/perf.html#skewed-joins

13 https://git.ouroath.com/adv-ds-core/feed-utils/blob/master/src/main/java/com/yahoo/adv_ds/feed_utils/IndexedAvroStorage.java

14 https://github.com/linkedin/dr-elephant