The [base value of Tez](http://hortonworks.com/blog/apache-tez-a-new-chapter-in-hadoop-data-processing/) had more complex expressions of **DAG job tasks**. Developers were no longer limited to simply map and reduce but they could develop complex networks of Map and Reduce tasks. Ideas like a set of job tasks that do Map then Reduce and Reduce again could not be expressed previously. More importantly, **Tez allowed for the intermediate results of tasks to go directly to the next task** skipping the dreaded write to disk step that was so costly when processing big data. My personal favorite was **a feature that was added to include container decay delay.** This means that the Yarn container that was allocated would not immediately disappear. Subsequent repeated identical queries would run dramatically faster since they did not incur the cost of a new container launch. Containers were reused allowing shared access to data along with lower latency queries again due to the lack of container setup time. There were more improvements involved in [Tez](http://www.slideshare.net/Hadoop_Summit/w-1205phall1saha)but the core changes described above were enough to confer a dramatic improvement in Hive.

1. DAG instaed of multiple map reduce.
2. Do not need to wait for resource for every map-reduce, works continiously till end.
3. Dowsnt need all map phases, as data from reduce can go directly to next reduce.
4. On the fly tuning, like choosing number of reducers, choosing join order.
5. Eliminates multiple lauch phase of container.
6. Can reuse the in memory views avoiding re computation of broadcast tables .

# Apache Tez Overview

First published on: May 5, 2016

Categories: [BigData](http://moi.vonos.net/category/BigData/)

# Introduction

Apache Tez is an orchestration framework for containers running on Apache YARN. It is primarily used as a back-end for tools such as Apache Hive and Apache Pig to run logic in parallel against potentially large datasets.

This article describes what it does, and (to a moderate depth) how it works.

# MapReduce and Tez

The [MapReduce design pattern](http://moi.vonos.net/bigdata/hmapreduce) is widely known, as is the Hadoop implementation of it (which is referred to here as HMapReduce). The primary goal of MapReduce is to allow software developers to write programs to process large datasets in parallel without the complexities of inter-process communication, distributed locks, etc. There is no doubt it is extremely successful at that. This article assumes the reader is familiar with MapReduce and HMapReduce.

There are many higher-level software tools for data-processing that would also like to be able to execute complex processing against large datasets, but without requiring developers to hand-write code. Such tools include Apache Hive and Apache Pig. Hive allows users to write queries in HiveQL, a language resembling standard SQL including support for joins, nested selects, grouping and sorting, and many other operations which can be expressed in a few lines of text but which are rather complex to actually implement. Pig is similar; its “language” is much less SQL-like but it also can express quite complex data processing in a concise form. One option for such tools to execute the user-provided logic efficiently against large data-sets is to “compile” the query to an equivalent Hadoop MapReduce program, submit it to YARN, and let it run. This is exactly what the initial versions of Hive, Pig, and a number of other similar tools did.

Unfortunately many of the things a user can express concisely cannot be performed as a single MapReduce program; the tool needs to generate a sequence of MapReduce programs and run them in order, with the output of one being fed into the input of the next pass. This does work - and worked better than any other approach available at the time. However it isn’t perfect; the primary problems are:

* generating the sequence of HMapReduce ApplicationMaster programs is complex;
* managing them (running them one after another) is complex;
* the overhead of starting so many individual processes via YARN is high;
* that HMapReduce mappers generally read from HDFS, and reducers generally write to HDFS (lots of IO);
* that map and reduce tasks must alternate, ie dataflows like map->map and reduce->reduce are not supported;
* that all mappers in a phase must finish executing before the corresponding reducers start;
* that each mapper and reducer runs just one “pass” then terminates, making it impossible to cache results in memory for a different function.

The significance of the overhead when starting processes depends upon the kind of work being done. When a query is doing a lot of work, and is going to take a few hours to run anyway, then the overhead of multiple processes managed via YARN is not too bad. For queries against small datasets, as done for example while developing and testing queries, ie when using a tool interactively, this overhead is very annoying and is referred to as latency. It is worst for very long sequences of HMapReduce operations each of which processes relatively small amounts of input.

The HMapReduce map phase works best when its input is a file in HDFS, so it can allocate splits effectively, and start mapper tasks on hosts which already have a copy of the input data blocks. When chaining mappers and reducers it is therefore necessary for the reducer to write to HDFS so the next mapper phase can be applied to that output in parallel. Unfortunately this results in a lot of extra disk and network IO for deep chains of MapReduce programs.

Some problems are best solved by a sequence of reduce passes. HMapReduce doesn’t support this directly; instead “no-op” mapper tasks need to be used which do nothing useful - but take up significant time. Theoretically, there should be no need to run a sequence of mapper tasks; the map-functions can instead simply be composed (chained) together within a single process and only the end result is output. In practice, this is difficult and tools often want to chain mapper tasks together. HMapReduce does not support directly chaining mappers either; instead “no-op” reducer tasks need to be used which do nothing useful - but waste even more time than no-op mappers.

When multiple mappers are running in parallel, and one finishes early, there is no way for a reducer to start processing that data. A reducer always needs to see all data for a single input category as produced by all mappers and thus needs to gather input from all mappers before it starts. This is part of the fundamental design of MapReduce, not a bug. However when the dataflow really should be mapper->mapper and the reducers are just “no-ops” then this limitation is unfortunate; in a system supporting direct mapper->mapper communication the second mapper could get started as soon as the first one starts generating output.

Tez was developed with the primary aim of replacing HMapReduce as a “back end” engine for tools such as Hive and Pig. It addresses all of the bullet-points above to allow such tools to execute user queries with less latency and higher throughput, while still running on a YARN cluster. The following sections describe how it achieves this.

Note that these problems aren’t significant for the kinds of simple cases where developers can write HMapReduce programs by hand. When the problem can be solved with a single HMapReduce pass, then the YARN overhead is relatively low, and there is no irrelevant disk-io. The HMapReduce design only causes significant issues when it is being used as a “back end” for a higher-level tool which needs to generate deep processing pipelines. Tez can be used directly by software developers, as an alternative to HMapReduce, and the API is elegant. However the advantages of Tez apply mostly to deep chains of HMapReduce programs, or to interactive use of them. Developers don’t write deep chains of HMapReduce programs by hand (sane ones, anyway) nor are such programs used interactively, so there is little incentive to hand-code against the Tez APIs. The existence of good documentation and extensive knowledge of coding against the HMapReduce APIs also mean developers are best advised to stick to the HMapReduce APIs for hand-written code.

At the current time (early 2016), Hive, Pig, and many other tools have added support for Tez as a back-end engine (usually in addition to the traditional HMapReduce back-end) and that support is considered stable (though still fairly new).

# Project Background

The motivation for the creation of Tez came from the “stinger initiative” within the Hive community, a self-chosen goal of improving Hive performance by a factor of 100. Stinger included improvements to file-formats (specifically the creation of ORCFile format) and addressing various other bottlenecks. Solving the HMapReduce inefficiencies via Tez was a significant part. The Stinger initiative is still in progress at the current time, with item such as “LLAP” (long lived daemon processes) and HBase as the meta-store still under development.

Tez is an Apache Foundation project, but was largely funded by Hortonworks, a company which provides a bundle of big-data tools and sell support and training. The primary developers are Bikas Saha and Hitesh Shah.

The Tez architecture is inspired partly from Hadoop MapReduce, and partly by an academic paper from Microsoft Research on a system named Dryad, which is in use within Microsoft Bing.

The [Tez Website](http://tez.apache.org/) sadly provides very little documentation on how Tez works, or how to use it - possibly due to its primary role as a “back end” for other tools. There is a good [video presentation](https://www.youtube.com/watch?v=9ZLLzlsz7h8) available, and the [corresponding slides](http://www.slideshare.net/Hadoop_Summit/w-1205phall1saha) are also available.

Due to the lack of available information on Tez, the content of this article is based on knowledge of its predecessor HMapReduce, the video and slides listed above, some time spent thinking about the issues Tez needs to address, and in some cases pure speculation. This article should therefore be taken as a guideline only. If you find any errors or misunderstandings in this article, please add your corrections as a comment!

# Alternatives

Tez is not the only project that is addressing the difficulties of using HMapReduce as a back-end for executing complex logic. In particular, the Apache Spark project can be used as a back-end in a similar way. Like Tez, it reduces interactive latency and increases overall throughput. At the current time, support in Hive for using Spark as a back-end is approaching production quality.

[Some postings](https://www.quora.com/When-would-someone-use-Apache-Tez-instead-of-Apache-Spark-or-vice-versa) make the claim that Tez is “irrelevant” because Spark can also act as a back-end for such tools, and has a “higher level API”. Often such articles miss the point that although Tez provides an API that can be directly coded against (and that API is arguably nicer than MapReduce), it is primarily designed as a target for higher-level tools. As an example, the [LLVM](http://llvm.org/) core libraries support a kind of “assembly language” for compilers and translate input in that form to executable form for many different types of computer; compilers for various high-level languages exist which generate LLVM-format output and then use the LLVM core libraries to generate executable applications. Criticising LLVM for “not having a high-level interface” is simply misguided - the “high level API” is provided by the compilers that build on top of it. Tez can be considered the equivalent of LLVM - an intermediate layer. When Spark is used as a back-end for tools such as Hive or Pig, it is actually just a small part of Spark (the “engine”) that is used, not the parts that provide a high-level API for developers to code against.

# The DAG

When a high-level tool (eg Hive, Pig) needs to chain multiple map/reduce phases together, HMapReduce provides no help; the high-level tool must generate multiple independent programs which pass data via the filesystem.

Tez instead allows its client application to define a graph where the vertices are “processing logic” and the edges are dataflows. The processing logic may be either a map-function or a reduce-function. The edges may be of type shuffle, one-to-one or broadcast. The graph may be arbitrarily deep, representing many transformations and different ways of passing data between steps.

A graph which contains a mapper node connected via a shuffle-edge to a reducer node will perform the same logic as an HMapReduce program.

A graph which contains a mapper node connected via a 1:1 node to another mapper node implements “mapper chaining” without needing the ugly “no-op reducer” that HMapReduce requires.

A graph which contains a reducer node connected via a shuffle-edge to a reducer node implements “reducer chaining” without a no-op mapper.

In Tez, such a processing+dataflow graph is called a DAG (Directed Acyclic Graph). Lots of articles on Tez talk about the wonders of the “DAG” without explaining what it actually does; we’ve already looked into the motivations for it and it will be described in even more detail later.

A Tez graph indicates how the dataflow would go for single-threaded processing, ie one process for each node. At runtime the engine may generate multiple copies of each node - as many as needed.

Because a multi-step algorithm in HMapReduce is a series of programs, HMapReduce itself can do no cross-phase optimisations. Tez has a declarative graph which it can analyse, eg to see how many steps are involved and calculate the maximum effective number of containers - before processing starts.

What is perhaps even more interesting than managing dataflows better is that an application which wants to execute logic no longer has to generate a complete Java program that calls HMapReduce APIs. Previously it was necessary to generate a unique program to submit to YARN which encoded the configuration-logic as java code (see the main-function in any HMapReduce tutorial). This program sets items such as the mapper and reducer classes to use, the reader and writer, etc. The higher-level application not only needs to generate such code but also compile it to a jarfile. With Tez the application still needs to generate the Mapper and Reducer classes that implement the desired logic. However after that point, the DAG is simply a data-structure; what YARN starts is a totally standard Tez interpreter which then executes the DAG as a script.

# Tez Sessions

To address the issues of latency when using an interactive system to execute queries over YARN, Tez invented sessions.

In HMapReduce, when a query is executed then an ApplicationMaster process is started via YARN. This starts mapper and reducer processes which eventually produce the desired result, and the ApplicationMaster then terminates. At this point, all resources in the cluster have been released. This is the correct behaviour for a background batch-mode application. However as an interactive user, having to wait for this sequence each time a query is submitted becomes tiresome. The busier the cluster is, the worse it gets.

An application using Tez (eg the Hive or Pig console) may choose to enable sessions, in which case the Tez equivalent of the ApplicationMaster program becomes a long-running process instead of exiting at the end of each query. It is still started via YARN as usual, but instead waits in a loop for DAGs to execute. The interactive user’s queries therefore run without the startup delay.

The Tez process belonging to a session can even hold onto containers allocated by YARN for running processing logic (map and reduce tasks), and reuse them for later queries. Again, this is something that a background batch-mode job should not do, as it is holding onto cluster resources which are idle. However as long as it doesn’t hold excessive resources, and interactive users aren’t consuming a major part of the cluster’s capacity, then this is a win for interactive users without a significant penalty.

# TezTaskHost

In HMapReduce, the Application Master runs a mapper task by requesting a YARN container and then sending a command to the relevant YARN node-manager to start a new Java process with the mapper code as the entry-point. When the mapper has processed all its input, the JVM terminates.

Tez also allocates a YARN container, and starts a Java process in it. However this process is the generic TezTaskHost program. The processing logic to execute (mapper or reducer) is then passed to it, and executed within the TezTaskHost process (presumably using java’s security mechanisms to contain it).

When the processing logic is complete (ie all input has been processed), the TezTaskHost program does not have to exit (as HMapReduce does); it can optionally wait for the next piece of logic to execute. This allows some significant optimisations. In particular, when multiple mappers are chained together via a 1:1 edge, then the second-pass mapper can be executed in the same TezTaskHost process as its predecessor, because the data it consumes will be on the local filesystem. This saves allocating a container via YARN, saves starting a JVM and loading all the Tez libraries. This approach also helps when a large number of identical tasks should be executed in parallel on a small cluster; here the same logic can be executed repeatedly via the same TezTaskHost process with different parameters (eg different byte-ranges of the same file). Reusing the same JVM in this way makes the JVM JIT implementation very happy and efficient.

When two mappers are connected via a 1:1 edge and are being run in the same TezTaskHost then the second phase can be run as soon as the first one has completed, without the artificial “sync point” that HMapReduce introduces due to the mandatory use of a “no-op” reducer between the mappers.

Such reuse cannot always be applied; mapper tasks should be run near the block of data they are to read, which means they cannot run on just any host. The TezTaskHost is nevertheless a useful optimisation.

# Tez Edges

A Tez edge is labelled with a “data movement” type:

* one-to-one: a process passes to one instance of the target node for the edge;
* broadcast: sent to all instances of the target node for the edge;
* scatter-gather: triggers a “sort/shuffle” identical to the one between HMapReduce mapper and reducer; all data for a category goes to one instance of the target node for the edge.

Edges are also labelled with a “data source” type:

* persisted: basically same as Mapper output; written to local FS
* persisted-reliable: basically same as Reducer output; written to HDFS
* ephemeral: tez isn’t required to write data generated by the “producer” node to disk.

Ephemeral edges can potentially be faster, as Tez can choose to buffer data in memory, or even stream it across a socket as it is generated (though Tez documentation doesn’t indicate if this optimisation actually exists). There is a disadvantage, however, if a node fails while processing ephemeral input data; Tez will need to determine which is the most recent persisted/persisted-reliable node and start processing again from there.

Using persisted/persisted-reliable for alternating nodes in the graph is identical to HMapReduce behaviour.

And edges are also labelled with a scheduling type:

* sequential: consumer task may be scheduled after producer task completes: default behaviour for HMapReduce. Presumably requires edge to also be persisted; it isn’t clear how “ephemeral” could work well with the sequential scheduling type.
* concurrent: consumer must be started together with producer.

Again, Tez documentation isn’t clear what actually happens when scheduling=concurrent is used. In theory, a socket could be established between producer and consumer, with data passed along it as it is produced - though it isn’t clear if such behaviour has been implemented. Error-handling would be tricky, as a failure of either the producer or consumer would presumably require the other component to be terminated and then both to be restarted. The implications of a consumer running slower than its producer are also unclear.

## Scatter-Gather Edges

What Tez calls a “scatter-gather” edge is identical to the normal dataflow between HMapReduce mapper and reducer processes.

Unlike HMapReduce, such an edge can be used to connect the output of a reduce-function to the input of another reduce-function.

## One-to-One Edges

As already discussed, the concept of applying a map-function to the output of another map-function is often useful. The most efficient way to do this is to simply run them within the same process, ie directly chain the functions together. However it appears that some tools using HMapReduce/Tez as “back end engines” prefer to run multiple map-functions in different processes. In this case, it makes sense to use the same number of processes for both the first and second pass, and to “pair” them up such that all the output of a process in the first pass goes to a single process running the second pass. Such a link can be represented in Tez with a “one-to-one” edge between two processing nodes.

As Tez is capable of reusing containers (TezTaskHost processes), one efficient solution Tez could apply would be to allocate N containers and run N instances of the first map-function, then run N instances of the second map-function in the same containers. As the TezTaskHost processes don’t terminate, the output (or at least part of it) of the first pass can potentially be buffered in memory as well as being written to disk. Even if written to disk, the second map-function will be reading from the local FS without any network IO being involved, which is efficient.

AFAIK, it makes no sense to connect reduce-nodes together with a one-to-one edge, as the whole point of a reduce-node is to gather all available records which have a specific category together.

## Broadcast Edges

In this edge-type, each record emitted by the instances of the “producer” node are sent to every instance of the “consumer” node.

This kind of dataflow isn’t available in HMapReduce at all. It can be used to implement a few interesting algorithms, including a more efficient join in some cases.

# Reducing Disk IO

As noted above, a one-to-one edge connecting two mappers gives Tez the possibility to buffer the output of a producer task in memory, and then run the corresponding consumer task on the same node, and feed it the in-memory data. Whether this provides a significant improvement in performance isn’t clear; the Apache Kafka message-broker documentation includes statistics showing that sequential writes to a local filesystem are implemented very efficiently by modern operating systems, as are sequential reads. In effect, the operating-system buffers such data itself using any available unused memory, so the benefits of caching at the application level are not as significant as would initially seem.

Potentially, combining map-functions via a one-to-one edge, with data-source-type of ephemeral and scheduling-type of concurrent could allow Tez to run multiple processes concurrently and pass the data across a socket as it is generated, without caching it in memory or writing it to disk. Tez documentation does not indicate if this has actually been implemented, or whether it has any significant performance benefits.

It isn’t clear how a map->reduce flow could be implemented without writing to disk. Nor is it clear how a reduce->map flow could be efficiently implemented without writing to HDFS; it is the write to HDFS that scatters blocks across multiple hosts so that mappers can run in parallel.

While Tez documentation claims that Tez “reduces disk-io”, I suspect this may have been over-hyped. It is clear that Tez has significant performance benefits over HMapReduce, but I have seen no measurements indicating how much of this is due to reduced disk-io.

# Failure Handling

One of the great things about HMapReduce is that it transparently deals with things such as nodes crashing, disks failing or becoming full, and any of the other disasters that can befall components in a large cluster. When a step fails, it simply restarts it.

Tez works in a similar way, also tracking the state of processing steps and restarting them as needed.

This does rely on the map-function and reduce-function being functional ie stateless. If a map-function or reduce-function has side-effects such as writing to databases or sending text-messages then of course all bets are off.

# Dynamically Selecting Reducer Parallelism

In HMapReduce, the number of reducer tasks to be run in parallel must be set before the program starts running. However the optimum number depends upon the number of categories generated by the mappers, and the number of records in each category. The maximum number of reducer processes is the number of different categories produced, but when few records are present then it is more efficient for each reducer process to handle the records for multiple categories.

Tez allows the user to provide a class which chooses the reducer parallelism. It is invoked after the mappers have generated their output and reported statistics back to the central Tez instance. Tez then starts exactly that number of instances of the reducer.

# Other Stuff

Tez supports running existing HMapReduce programs unaltered. HMapReduce has a plugable API for “frameworks” which it uses to switch between “local mode” (used during development), “classic” (Hadoop 1.x clusters) and “yarn” (Hadoop 2.x clusters). The Tez project has implemented a compatible framework which uses Tez to perform the underlying orchestration.

Tez is a purely client-side technology; nothing needs to be pre-installed on the YARN worker nodes. It is even cleaner than HMapReduce, which requires that YARN nodemanagers be configured to run a “shuffle handler” auxiliary service to handle requests from reducers wanting to retrieving output generated by earlier mappers. As the TezTaskHost does not exit after the map-function has completed its work, it can handle the reducer requests itself.

# Final Words

The start of this article described Tez as an orchestration framework for YARN; I hope it is now clear what was meant. Tez accepts a graph describing the processing steps to be executed, and organises all of the rest: allocating containers from YARN, starting processes in appropriate places, restarting them on failure, copying data around as needed, and other necessary work.