KIWI Project@Data Management Research Section
ETRI Technical Report 14ZS1410-TR-64

218 Gajeong-ro, Yuseong-gu
Daejeon, 305-700, South Korea
http://www.etri.re.kr/
http://sungsoo.github.com/

Analysis of the Apache Tez Design

Sung-Soo Kim
sungsoo@etri.re.kr

Abstract

MapReduce has served us well. For years it has been the processing engine for Hadoop and has been the backbone upon which a huge amount of value has been created. While it is here to stay, new paradigms are also needed in order to enable Hadoop to serve an even greater number of usage patterns. A key and emerging example is the need for *interactive query*, which today is challenged by the *batch-oriented nature* of MapReduce. A key step to enabling this new world was *Apache YARN* and today the community proposes the next step... *Tez*

1 Introduction

Apache Hadoop 2.0 (aka **YARN**) continues to make its way through the open source community process at the Apache Software Foundation and is getting closer to being declared ready from a community development perspective. YARN on its own provides many benefits over Hadoop 1.x and its MapReduce job execution engine:

- Concurrent cluster applications via multiple independent AppMasters
- Reduced job startup overheads
- Pluggable scheduling policy framework
- Improved security framework

The support for third party AppMasters is the crucial aspect to flexibility in YARN. It permits new job runtimes in addition to classical map-reduce, whilst still keeping M/R available and allowing both the old and new to co-exist on a single cluster. Apache Tez is one such job runtime that provides richer capabilities than traditional map-reduce [1]. The motivation is to provide a better runtime for scenarios such as relational-querying that do not have a strong affinity for the map-reduce primitive. This need arises because the Map-Reduce primitive mandates a very particular shape to every job and although this mandatory shape is very general and can be used to implement essentially any batch-oriented data processing job, it conflates too many details and provides too little flexibility.

1. Client-side determination of input pieces
2. Job startup
3. Map phase, with optional in-process combiner
   Each mapper reads input from durable storage
4. Hash partition with local per-bucket sort.
5. Data movement via framework initiated by reduce-side pull mechanism
6. Ordered merge
7. Reduce phase
8. Write to durable storage

The map-reduce primitive has proved to be very useful as the basis of a reliable cluster computation runtime and it is well suited to data processing tasks that involve a small number of jobs that benefit from the standard behavior. However, algorithms that require many iterations suffer from the high overheads of job startup and from frequent reads and writes to durable storage. Relation query languages such as Hive suffer from those issues and from the need to massage multiple datasets into homogeneous inputs as a M/R job can only consume one physical dataset (excluding support for side-data channels such as distributed cache).

1.1 What is Tez?

Tez – Hindi for "*speed*" provides a general-purpose, highly customizable framework that creates simplifies data-processing tasks across both small scale (low-latency) and large-scale (high throughput) workloads in Hadoop [2]. It generalizes the MapReduce paradigm to a more powerful framework by providing the ability to execute a complex **DAG** (*directed acyclic graph*) of tasks for a single job so that projects in the Apache Hadoop ecosystem such as Apache Hive, Apache Pig and Cascading can meet requirements for human-interactive response times and extreme throughput at petabyte scale (clearly MapReduce has been a key driver in achieving this).

1.2 What Tez Does

Tez is the logical next step for Apache Hadoop after **Apache Hadoop YARN**. With YARN the community generalized

Hadoop MapReduce to provide a *general-purpose resource management framework* wherein MapReduce became merely one of the applications that could process data in a Hadoop cluster. Tez provides a more general data-processing application to the benefit of the entire ecosystem.

Tez will speed Pig and Hive workloads by an order of magnitude. By eliminating unnecessary tasks, synchronization barriers, and reads from and write to HDFS, Tez speeds up data processing across both small-scale, low-latency and large-scale, high-throughput workloads.

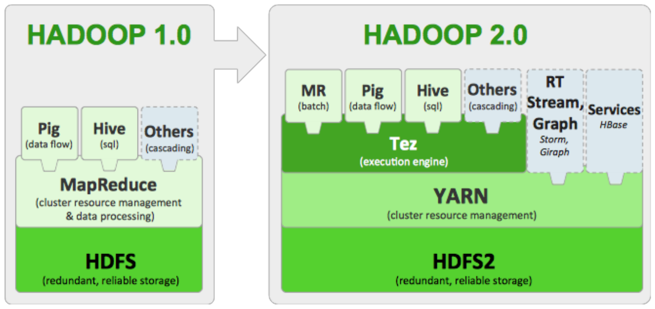

HADOOP 1.0
Pig (data flow)
Hive (sql)
Others (cascading)
MapReduce (cluster resource management & data processing)
HDFS (redundant, reliable storage)
HADOOP 2.0
MR (batch)
Pig (data flow)
Hive (sql)
Others (cascading)
RT Stream, Graph Storm, Giraph
Services HBase
Tez (execution engine)
YARN (cluster resource management)
HDFS2 (redundant, reliable storage)

Figure 1: Hadoop 1.0 vs. Hadoop 2.0

With the emergence of Apache Hadoop YARN as the basis of next generation data-processing architectures, there is a strong need for an application which can execute a complex DAG of tasks which can then be shared by Apache Pig, Apache Hive, Cascading and others. The constrained DAG expressible in MapReduce (one set of maps followed by one set of reduces) often results in multiple MapReduce jobs which harm latency for short queries (overhead of launching multiple jobs) and throughput for large-scale queries (too much overhead for materializing intermediate job outputs to the filesystem). With Tez, we introduce a more expressive DAG of tasks, within a single application or job, that is better aligned with the required processing task – thus, for e.g., any *given SQL query can be expressed as a single job* using Tez.

The below graphic illustrates the advantages provided by Tez for complex SQL queries in Apache Hive or complex Apache Pig scripts.

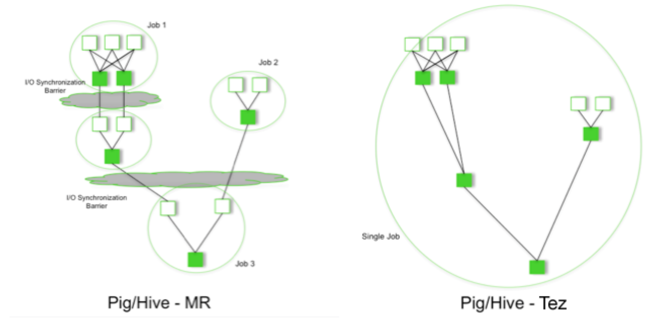


Figure 2: General example of a Map-Reduce execution plan compared to DAG execution plan

Tez is critical to the **Stinger Initiative** and goes a long way in helping Hive support both interactive queries and batch queries. Tez provides a single underlying framework to support both latency and throughput sensitive applications, there-by obviating the need for multiple frameworks and systems to be installed, maintained and supported, a *key advantage to enterprises looking to rationalize their data architectures*.

Essentially, Tez is the logical next step for Apache Hadoop after Apache Hadoop YARN. With YARN the community generalized Hadoop MapReduce to provide a general-purpose resource management framework(YARN) where-in MapReduce became merely *one of the applications* that could process data in your Hadoop cluster. With Tez, we build on YARN and our experience with the MapReduce to provide a more general data-processing application to the benefit of the entire ecosystem i.e. Apache Hive, Apache Pig etc.

1.3 Motivation

Distributed data processing is the core application that Apache Hadoop is built around [3]. Storing and analyzing *large volumes* and *variety* of data efficiently has been the cornerstone use case that has driven large scale adoption of Hadoop, and has resulted in creating enormous value for the Hadoop adopters. Over the years, while building and running data processing applications based on MapReduce, we have understood a lot about the strengths and weaknesses of this framework and how we would like to evolve the *Hadoop data processing framework* to meet the evolving needs of Hadoop users. As the Hadoop compute platform moves into its next phase with **YARN**, it has decoupled itself from MapReduce being the only application, and opened the opportunity to create a new data processing framework to meet the new challenges. Apache Tez aspires to live up to these lofty goals.

2 Key Design Themes

Higher-level data processing applications like Hive and Pig need an execution framework that can express their complex query logic in an efficient manner and then execute it with high performance. Apache Tez has been built around the following main design themes that solve these key challenges in the Hadoop data processing domain.

2.1 Ability to express, model and execute data processing logic

Tez models data processing as a *dataflow graph* with vertices in the graph representing *application logic* and edges representing *movement of data*. A rich dataflow definition API allows users to express *complex query logic* in an intuitive manner and it is a natural fit for *query plans* produced by higher-level declarative applications like **Hive** and **Pig**. As an example, the diagram shows how to model an *ordered distributed sort* using **range partitioning**. The *Preprocessor* stage sends samples to a **Sampler** that calculates sorted data ranges for each data partition such that the work is *uniformly distributed*. The ranges are sent to **Partition** and **Aggregate** stages that read their assigned ranges and perform the data *scatter-gather*. This dataflow pipeline can be expressed as a single Tez job that will run the entire computation. Expanding this logical graph into a physical graph of tasks and executing it is taken care of by Tez.

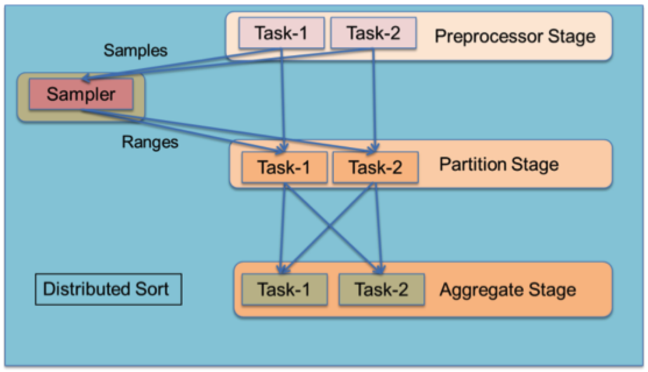


Figure 3: DAG execution

2.2 Flexible Input-Processor-Output task model

Tez models the user logic running in each vertex of the dataflow graph as a composition of **Input**, **Processor** and **Output** modules. Input & Output determine the *data format* and how and where it is read/written. *Processor* holds the *data transformation* logic. Tez does not impose any data format and only requires that a combination of Input, Processor and Output must be compatible with each other with respect to their formats when they are composed to instantiate a *vertex task*. Similarly, an Input and Output pair connecting two tasks should be compatible with each other. In the diagram, we can see how composing different Inputs, Outputs and Processors can produce different tasks.

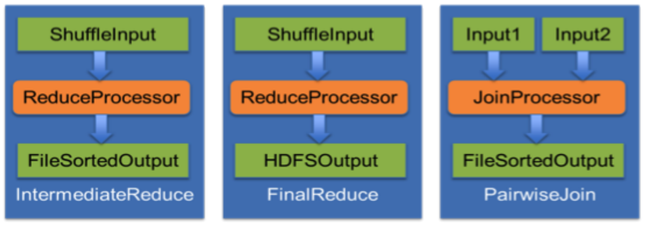
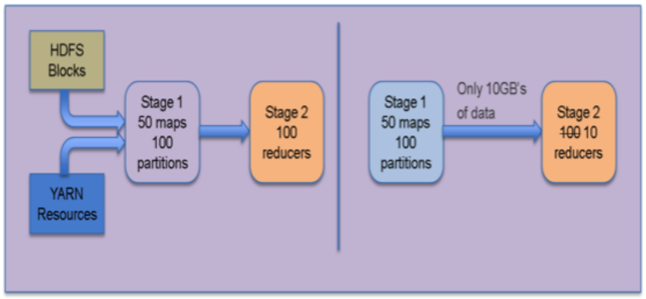


Figure 4: DAG execution

2.3 Performance via Dynamic Graph Reconfiguration

Distributed data processing is *dynamic* by nature and it is extremely difficult to statically determine *optimal concurrency* and *data movement methods* a priori. More information is available during runtime, like data samples and sizes, which may help optimize the *execution plan* further. We also recognize that Tez by itself cannot always have the smarts to perform these *dynamic optimizations*. The design of Tez includes support for pluggable vertex management modules to collect relevant information from tasks and change the dataflow graph at runtime to optimize for performance and resource usage. The diagram shows how Tez can determine an appropriate number of reducers in a MapReduce like job by observing the actual data output produced and the desired load per reduce task.



HDFS Blocks
YARN Resources
Stage 1 50 maps 100 partitions
Stage 2 100 reducers
Stage 1 50 maps 100 partitions
Only 10GB's of data
Stage 2 ~~100~~ 10 reducers

Figure 5: Plan Reconfiguration at Runtime

2.4 Performance via Optimal Resource Management

Resources acquisition in a *distributed multi-tenant environment* is based on cluster capacity, load and other quotas enforced by the *resource management framework* like **YARN**. Thus resource available to the user may vary over time and over different executions of the job. It becomes paramount to be able to efficiently use all available resources to run a job as fast as possible during one instance of execution and predictably over different instances of execution. The Tez execution engine framework allows for efficient acquisition of resources from YARN along with *extensive reuse* of every component in the pipeline such that no operation is duplicated unnecessarily. These efficiencies are exposed to user logic, where possible, such that users may also leverage this for *efficient caching* and avoid *work duplication*. The diagram shows how Tez runs multiple containers within the same YARN container host and how users can leverage that to store their own objects that may be shared across tasks.

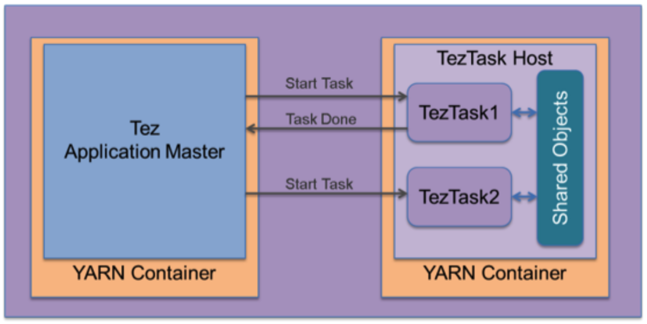
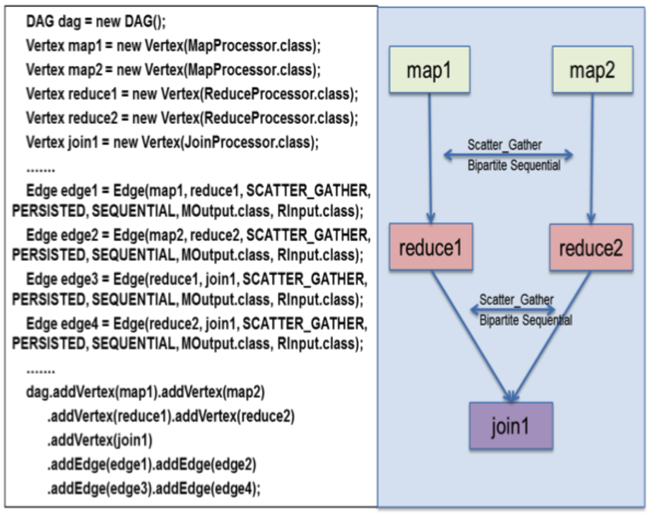


Figure 6: Optimal Resource Management

3 Data Processing API in Apache Tez

3.1 Overview

Apache Tez models data processing as a *dataflow graph*, with the **vertices** in the graph representing *processing of data* and **edges** representing *movement of data* between the processing. Thus *user logic*, that analyses and modifies the data, sits in the **vertices**. Edges determine the consumer of the data, how the data is transferred and the *dependency* between the *producer* and *consumer* vertices. This model concisely captures the *logical definition of the computation*. When the Tez job executes on the cluster, it expands this *logical graph* into a *physical graph* by adding parallelism at the vertices to scale to the data size being processed. Multiple tasks are created per logical vertex to perform the computation in parallel.



```
    DAG dag = new DAG();
    Vertex map1 = new Vertex(MapProcessor.class);
    Vertex map2 = new Vertex(MapProcessor.class);
    Vertex reduce1 = new Vertex(ReduceProcessor.class);
    Vertex reduce2 = new Vertex(ReduceProcessor.class);
    Vertex join1 = new Vertex(JoinProcessor.class);
    .......
    Edge edge1 = Edge(map1, reduce1, SCATTER_GATHER,
PERSISTED, SEQUENTIAL, MOutput.class, RInput.class);
    Edge edge2 = Edge(map2, reduce2, SCATTER_GATHER,
PERSISTED, SEQUENTIAL, MOutput.class, RInput.class);
    Edge edge3 = Edge(reduce1, join1, SCATTER_GATHER,
PERSISTED, SEQUENTIAL, MOutput.class, RInput.class);
    Edge edge4 = Edge(reduce2, join1, SCATTER_GATHER,
PERSISTED, SEQUENTIAL, MOutput.class, RInput.class);
    .......
    dag.addVertex(map1).addVertex(map2)
        .addVertex(reduce1).addVertex(reduce2)
        .addVertex(join1)
        .addEdge(edge1).addEdge(edge2)
        .addEdge(edge3).addEdge(edge4);
```

Figure 7: DAG execution

3.2 DAG Definition API

More technically, the data processing is expressed in the form of a *directed acyclic graph* (**DAG**). The processing starts at the root vertices of the DAG and continues down the *directed edges* till it reaches the leaf vertices. When all the vertices in the DAG have completed then the data processing job is done. The graph does not have cycles because the *fault tolerance mechanism* used by Tez is **re-execution** of failed tasks. When the input to a task is lost then the producer task of the input is re-executed and so Tez needs to be able to *walk up* the graph edges to locate a non-failed task from which to re-start the computation. *Cycles* in the graph can make this walk *difficult* to perform. In some cases, cycles may be handled by *unrolling* them to create a DAG.

Tez defines a simple Java API to express a DAG of data processing [4]. The API has three components

- **DAG.** this defines the overall job. The user creates a DAG object for each data processing job.
- **Vertex.** this defines the user logic and the resources & environment needed to execute the user logic. The user creates a Vertex object for each step in the job and adds it to the DAG.
- **Edge.** this defines the connection between producer and consumer vertices. The user creates an Edge object and connects the producer and consumer vertices using it.

The diagram shows a *dataflow graph* and its definition using the DAG API (simplified). The job consists of 2 vertices performing a "**Map**" operation on 2 datasets. Their output is consumed by 2 vertices that do a "**Reduce**" operation. Their output is brought together in the last vertex that does a "**Join**" operation.

Tez handles expanding this *logical graph* at runtime to perform the operations *in parallel* using multiple tasks. The diagram shows a runtime expansion in which the first M-R pair has a parallelism of 2 while the second has a parallelism of 3. Both branches of computation merge in the **Join operation** that has a parallelism of 2. *Edge properties* are at the heart of this runtime activity.

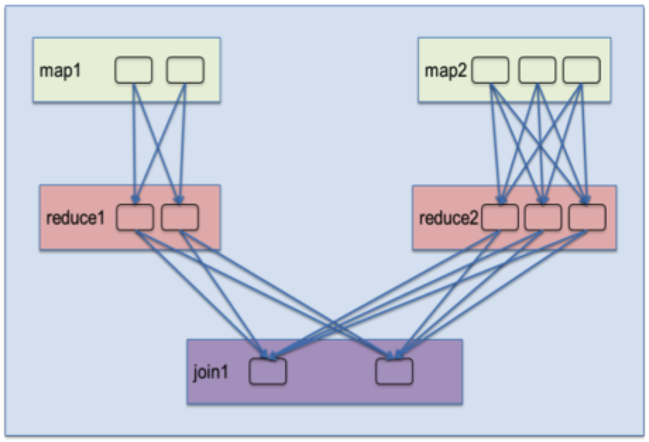


Figure 8: DAG execution

3.3 Edge Properties

The following edge properties enable Tez to instantiate the tasks, configure their inputs and outputs, schedule them appropriately and help *route* the data between the tasks. The parallelism for each vertex is determined based on *user guidance*, *data size* and *resources*.

- **Data movement.** Defines *routing* of data between tasks
  - *One-To-One*: Data from the *i*th producer task routes to the *i*th consumer task.
  - *Broadcast*: Data from a producer task routes to *all* consumer tasks.
  - *Scatter–Gather*: Producer tasks *scatter* data into *shards* and consumer tasks *gather* the *shards*. The *i*th shard from all producer tasks routes to the *i*th consumer task.

- **Scheduling.** Defines when a *consumer* task is scheduled
  - *Sequential*: Consumer task may be scheduled after a *producer task* completes.
  - *Concurrent*: Consumer task must be *co-scheduled* with a producer task.
- **Data source.** Defines the *lifetime/reliability* of a task output
  - *Persisted*: Output will be available after the task exits. Output may be lost later on.
  - *Persisted-Reliable*: Output is reliably stored and will always be available
  - *Ephemeral*: Output is available only while the producer task is running

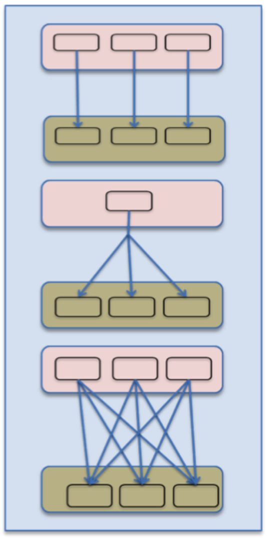


Figure 9: DAG execution

Some real life use cases will help in clarifying the edge properties. **Mapreduce** would be expressed with the *scatter-gather*, *sequential* and *persisted* edge properties. **Map tasks** *scatter* partitions and reduce tasks gather them. **Reduce tasks** are *scheduled* after the map tasks complete and the map task outputs are written to local disk and hence available after the map tasks have completed. When a vertex *checkpoints* its output into HDFS then its *output edge* has a *persisted-reliable* property. If a producer vertex is *streaming data* directly to a consumer vertex then the edge between them has *ephemeral* and *concurrent* properties. A *broadcast* property is used on a *sampler vertex* that produces a **global histogram** of data ranges for *range partitioning*.

We hope that the Tez dataflow definition API will be able to express a broad spectrum of *data processing topologies* and enable higher level languages to elegantly transform their queries into Tez jobs.

4 Runtime API in Apache Tez

Apache Tez models data processing as a *dataflow graph*, with the **vertices** in the graph representing *processing of data* and **edges** representing *movement of data* between the processing [5]. Thus *user logic*, that analyses and modifies the data, sits in the **vertices**. Edges determine the consumer of the data, how the data is transferred and the *dependency* between the *producer* and *consumer* vertices.

For users of **MapReduce** (**MR**), the most primitive functionality that Tez can provide is an ability to run a *chain of Reduce stages* as compared to a *single* Reduce stage in the current MR implementation. Via the Task API, Tez can do this and much more by facilitating execution of any form of processing logic that does not need to be retrofitted into a Map or Reduce task and also by supporting multiple options of data transfer between different vertices that are not restricted to the **MapReduce** *shuffle transport mechanism*.

4.1 The Building Blocks of Tez

The Task API provides the building blocks for a user to plug-in their logic to analyze and modify data into the vertex and augment this processing logic with the necessary plugins to *transfer* and *route* data between vertices.

Tez models the user logic running in each vertex as a composition of a set of Inputs, a Processor and a set of Outputs.

- **Input:** An input represents a pipe through which a processor can accept input data from a *data source* such as HDFS or the output generated by another vertex.
- **Processor:** The entity responsible for *consuming* one or more Inputs and *producing* one or more Outputs.
- **Output:** An output represents a pipe through which a processor can generate output data for another vertex to consume or to a *data sink* such as HDFS.

Given that an edge in a DAG is a logical entity that represents a number of physical connections between the tasks of 2 connected vertices, to improve ease of programmability for a developer implementing a new Processor, there are 2 kinds of Inputs and Outputs to either expose or hide the level of complexity:

- **Logical:** A corresponding pair of a *LogicalInput* and a *LogicalOutput* represent the *logical edge* between 2 vertices. The implementation of Logical objects hides all the underlying physical connections and exposes a single view to the data.
- **Physical:** The pair of Physical Input and Output represents the *connection* between a task of the *Source vertex* and a task of a *Destination vertex*.

An example of the *Reduce stage* within an MR job would be a **Reduce Processor** that receives data from the maps via

ShuffleInput and generates output to HDFS. Likewise, an intermediate Reduce stage in an **MRR chain** would be quite similar to the final Reduce stage except for the difference in the Output type.

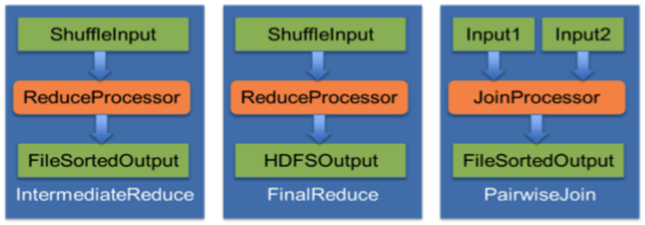
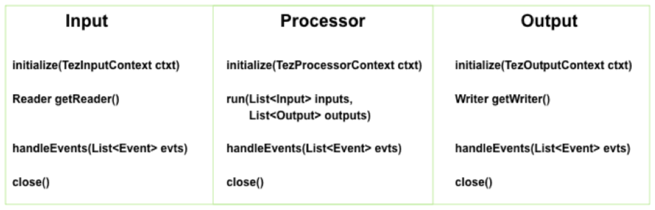


Figure 10: DAG execution

4.2 Tez Runtime API



Input	Processor	Output
initialize(TezInputContext ctxt)	initialize(TezProcessorContext ctxt)	initialize(TezOutputContext ctxt)
Reader getReader()	run(List<Input> inputs, List<Output> outputs)	Writer getWriter()
handleEvents(List<Event> evts)	handleEvents(List<Event> evts)	handleEvents(List<Event> evts)
close()	close()	close()

Figure 11: DAG execution

To implement a new Input, Processor or Output, a user to implement the appropriate interfaces mentioned above. All objects are given a Context object in their initialize functions. This context is the hook for these objects to communicate to the *Tez framework*. The Inputs and Outputs are expected to provide implementations for their respective Readers and Writers which are then used by the Processor to read/write data. In a task, after the Tez framework has initialized all the necessary Inputs, Outputs and the Processor, the Tez framework invokes the Processor's run function and passes the appropriate handles to all the Inputs and Outputs for that particular task.

Tez allows all inputs and outputs to be *pluggable*. This requires support for passing of information from the Output of a source vertex to the Input of the destination vertex. For example, let us assume that the Output of a source vertex writes all of its data to a *key-value store*. The Output would need to communicate the "**key**" to the Input of the next stage so that the Input can retrieve the correct data from the key-value store. To facilitate this, Tez uses **Events**.

4.3 Events in Tez

Events in Tez are a way to pass information amongst different components.

- The Tez framework uses Events to pass information of system events such as *task failures* to the required components.
- Inputs of a vertex can inform the framework of any failures encountered when trying to retrieve data from the source vertex's Output that in turn can be used by the framework to take failure recovery measures.
- An Output can pass information of the location of the data, which it generates, to the Inputs of the destination vertex. An example of this is described in the Shuffle Event diagram which shows how the output of a Map stage informs the Shuffle Input of the Reduce stage of the location of its output via a Data Movement Event.

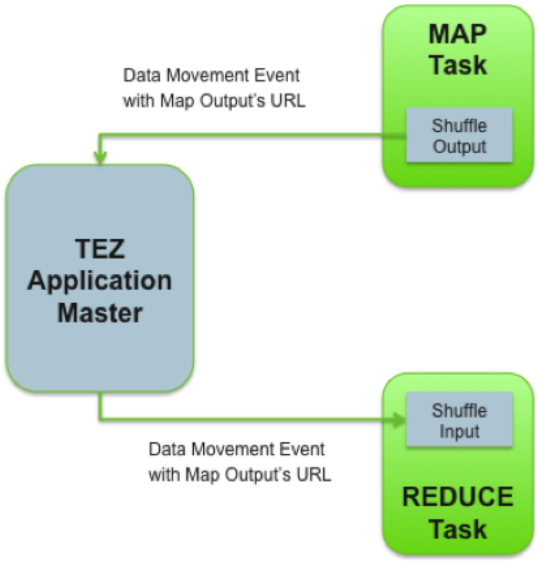

MAP Task
Shuffle Output
Data Movement Event with Map Output's URL
TEZ Application Master
Shuffle Input
Data Movement Event with Map Output's URL
REDUCE Task

Figure 12: DAG execution

Another use of Events is to enable run-time changes to the DAG execution plan. For example, based on the amount of the data being generated by a Map stage, it may be more optimal to run less reduce tasks within the following Reduce stage. Events generated by Outputs are routed to the pluggable Vertex/Edge management modules, allowing them to make the necessary decisions to modify some run-time parameters as needed.

4.4 Available implementations of Inputs/Processors/Outputs

The *flexibility* of Tez allows anyone to implement their Inputs and Outputs, whether they use *blocking/non-blocking transport protocols*, handle data in the form of raw bytes/records/key-value pairs etc., and build Processors to handle these variety of Inputs and Outputs.

There is already a small repository of various implementations of Inputs/Outputs/Processors:

- `MRInput` and `MROutput`: Basic input and outputs to handle data to/from HDFS that are MapReduce compatible as they use MapReduce constructs such as InputFormat, RecordReader, OutputFormat and RecordWriter.

- `OnFileSortedOutput` and `ShuffleMergedInput`: A pair of key-value based Input and Output that use the local disk for all I/O and provide the same sort+merge functionality that is required for the "shuffle" edge between the Map and Reduce stages in a MapReduce job.

- `OnFileUnorderedKVOutput` and `ShuffledUnorderedKVInput`: These are similar to the shuffle pair mentioned earlier except that the data is not sorted implicitly. This can be a big performance boost in various situations.

- `MapProcessor` and `ReduceProcessor`: As the names suggest, these processors are available for anyone trying to run a MapReduce job on the Tez execution framework. They can be used to run an MRR chain too.

As the Hive and Pig projects adapt to use Tez, we hope this repository will grow to house a common set of building blocks for use across the different projects.

5 Writing a Tez Input, Processor and Output

5.1 Tez Task

Tez task is constituted of all the Inputs on its incoming edges, the Processor configured for the Vertex, and all the Output(s) on it's outgoing edge [6].

The number of tasks for a vertex is equal to the parallelism set for that vertex – which is set at DAG construction time, or modified during runtime via user plugins running in the AM.

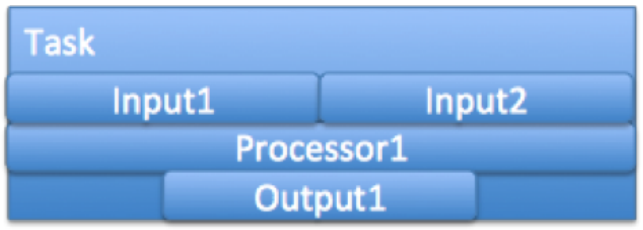


Figure 13: tez01

The diagram shows a single task. The vertex is configured to run Processor1 – has two incoming edges – with the output of the edge specified as Input1 and Input2 respectively, and has a single outgoing edge – with the input to this edge configured as Output1. There will be n such Task instances created per Vertex – depending on the parallelism.

5.2 Initialization of a Tez task

The following steps are followed to initialize and run a Tez task.

The Tez framework will first construct instances of the specified Input(s), Processor, Output(s) using a 0 argument constructor.

For a LogicalInput and a LogicalOutput – the Tez framework will set the number of physical connections using the respective setNumPhysicalInputs and setNumPhysicalOutputs methods.

The Input(s), Processor and Output(s) will then be initialized via their respective initialize methods. Configuration and context information is made available to the Is/P/Os via this call. More information on the Context classes is available in the JavaDoc for TezInputContext, TezProcessorContext and TezOutputContext.

The Processor run method will be called with the initialized Inputs and Outputs passed in as arguments (as a Map – connected vertexName to Input/Output).

Once the run method completes, the Input(s), Processor and Output(s) will be closed, and the task is considered to be complete.

Notes for I/P/O writers:

Each Input / Processor / Output must provide a 0 argument constructor. No assumptions should be made about the order in which the Inputs, Processor and Outputs will be initialized, or closed. Assumptions should also not be made about how the Initialization, Close and Processor run will be invoked – i.e. on the same thread or multiple threads.

5.2.1 Common Interfaces to be implemented by Input/Processor/Output

List initialize(Tez*Context) -This is where I/P/O receive their corresponding context objects. They can, optionally, return a list of events. handleEvents(List events) – Any events generated for the specific I/P/O will be passed in via this interface. Inputs receive DataMovementEvent(s) generated by corresponding Outputs on this interface – and will need to interpret them to retrieve data. At the moment, this can be ignored for Outputs and Processors. List close() – Any cleanup or final commits will typically be implemented in the close method. This is generally a good place for Outputs to generate DataMovementEvent(s). More on these events later.

5.2.2 Providing User Information to an Input / Processor / Output

Information specified in the bytePayload associated with an Input/Processor/Output is made available to the respective I/P/O via their context objects.

Users provide this information as a byte array – and can specify any information that may be required at runtime by the I/P/O. This could include configuration, execution plans for Hive/PIG, etc. As an example, the current inputs use a Hadoop Configuration instance for backward compatibility. Hive may choose to send it's vertex execution plan as part of this field instead of using the distributed cache provided by YARN.

Typically, Inputs and Outputs exist as a pair – the Input knows how to process DataMovementEvent(s) generated by the corresponding Output, and how to interpret the data. This information will generally be encoded into some form of configuration (specified via the userPayload) used by the Output-Input pair, and should match. As an example – the output Key type configured on an Output should match the Input key type on the corresponding Input.

5.3 Writing a Tez LogicalOutput

A LogicalOutput can be considered to have two main responsibilities – 1) dealing with the actual data provided by the Processor – partitioning it for the 'physical' edges, serializing it etc, and 2) Providing information to Tez (in effect the subsequent Input) on where this data is available.

5.3.1 Processing the Data

Depending on the connection pattern being used – an Output will generate data to a single 'physical' edge or multiple 'physical' edges. A LogicalOutput is responsible for partitioning the data into these 'physical' edges.

It would typically work in conjunction with the configured downstream Input to write data in a specific data format understood by the downstream Input. This includes a serialization mechanism, compression etc.

As an example: OnFileSortedOutput which is the Output used for a MapReduce shuffle makes use of a Partitioner to partition the data into n partitions ('physical' edges) – where n corresponds to the number of downstream tasks. It also sorts the data per partition, and writes it out as Key-Value pairs using Hadoop serialization which is understood by the downstream Input (ShuffledMergedInput in this case).

5.3.2 Providing information on how the data is to be retrieved

A LogicalOutput needs to send out information on how data is to be retrieved by the corresponding downstream Input defined on an edge. This is done by generating DataMovementEvent(s). These events are routed by the AM, based on the connection pattern, to the relevant LogicalInputs.

These events can be sent at anytime by using the TezOutputContext with which the Output was initialized. Alternately, they can be returned as part of the initialize() or close() calls. More on DataMovementEvent(s) further down.

Continuing with the OnFileSortedOutput example: This will generate one event per partition – the sourceIndex for each of these events will be the partition number. This particular Output makes use of the MapReduce ShuffleHandler, which requires downstream Inputs to pull data over HTTP. The payload for these events contains the host name and port for the http server, as well as an identifier which uniquely identifies the specific task and Input instance running this output.

In case of OnFileSortedOutput – these events are generated during the close() call.

View OnFileSortedOutput.java

5.3.3 Specific interface for a LogicalOutput

setNumPhysicalOutputs(int) – This is where a Logical Output is informed about the number of physical outgoing edges for the output. Writer getWriter() – An implementation of the Writer interface, which can be used by a Processor to write to this Output.

5.4 Writing a Tez LogicalInput

The main responsibilities of a Logical Input are 1) Obtaining the actual data over the 'physical' edges, and 2) Interpreting the data, and providing a single 'Logical' view of this data to the Processor.

5.4.1 Obtaining the Data

A LogicalInput will receive DataMovementEvent(s) generated by the corresponding LogicalOutput which generated them. It needs to interpret these events to get hold of the data. The number of DataMovementEvent(s) a LogicalInput receives is typically equal to the number of physical edges it is configured with, and is used as a termination condition.

As an example: ShuffledMergedInput (which is the Input on the OnFileSortedOutput-ShuffledMergedInput O-I edge) would fetch data from the ShuffleHandler by interpretting the host, port and identifier from the DataMovementEvent(s) it receives.

5.4.2 Providing a view of the data to the Processor

A LogicalInput will typically expose the data to the Processor via a Reader interface. This would involve interpreting the data, manipulating it if required – decompression, ser-de etc.

Continuing with the ShuffledMergedInput example: This input fetches all the data – one chunk per source task and partition – each of which is sorted. It then proceeds to merge the sorted chunks and makes the data available to the Processor only after this step – via a KeyValues reader implementation.

View ShuffledMergedInput.java

View ShuffledUnorderedKVInput.java

5.4.3 Specific interface for a LogicalInput

setNumPhysicalInputs(int) – This is where a LogicalInput is informed about the number of physical incoming edges. Reader getReader() – An implementation of the Reader interface, which can be used by a Processor to read from this Input

5.5 Writing a Tez LogicalIOProcessor

A logical processor receives configured LogicalInput(s) and LogicalOutput(s). It is responsible for reading source data from the Input(s), processing it, and writing data out to the configured Output(s).

A processor is aware of which vertex (vertex-name) a specific Input is from. Similarly, it is aware of the output vertex (via the vertex-name) associated with a specific Output. It would typically validate the Input and Output types, process the Inputs based on the source vertex and generate output for the various destination vertices.

As an example: The MapProcessor validates that it is configured with only a single Input of type MRInput – since that is the only input it knows how to work with. It also validates the Output to be an OnFileSortedOutput or a MROutput. It then proceeds to obtain a KeyValue reader from the MRInput, and KeyValueWriter from the OnFileSortedOutput or MROutput. The KeyvalueReader instance is used to walk all they keys in the input – on which the user configured map function is called, with a MapReduce output collector backed by the KeyValue writer instance.

5.5.1 Specific interface for a LogicalIOProcessor

run(Map inputs, Map outputs) – This is where a processor should implement it's compute logic. It receives initialized Input(s) and Output(s) along with the vertex names to which the Input(s) and Output(s) are connected.

5.6 Data Movement Event

A DataMovementEvent is used to communicate between Outputs and Inputs to specify location information. A byte payload field is available for this – the contents of which should be understood by the communicating Outputs and Inputs. This byte payload could be interpreted by user-plugins running within the AM to modify the DAG (Auto reduce-parallelism as an example).

DataMovementEvent(s) are typically generated per physical edge between the Output and Input. The event generator needs to set the sourceIndex on the event being generated – and this matches the physical Output/Input that generated the event. Based on the ConnectionPattern specified for the DAG – Tez sets the targetIndex, so that the event receiver knows which physical Input/Output the event is meant for. An example of data movement events generated by a ScatterGather connection pattern (Shuffle) follows, with values specified for the source and target Index.

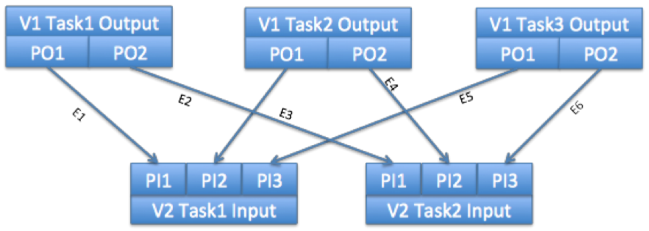


Figure 14: tez01

In this case the Input has 3 tasks, and the output has 2 tasks. Each input generates 1 partition (physical output) for the downstream tasks, and each downstream task consumes the same partition from each of the upstream tasks.

Vertex1, Task1 will generate two DataMovementEvents – E1 and E2. E1, sourceIndex = 0 (since it is generated by the 1st physical output) E2, sourceIndex = 1 (since it is generated by the 2nd physical output)

Similarly Vertex1, Task2 and Task3 will generate two data movement events each. E3 and E5, sourceIndex=0 E4 and E6, sourceIndex=1

Based on the ScatterGather ConnectionPattern, the AM will rout the events to respective tasks. E1, E3, E5 with sourceIndex 1 will be sent to Vertex2, Task1 E2, E4, E6 with sourceIndex 2 will be sent to Vertex2, Task2

The destination will see the following targetIndex (based on the physical edges between the tasks (arrows)) E1, targetIndex=0 – first physical input to V2, Task1 E3, targetIndex=1 – second physical input to V2, Task1 E5, targetIndex=5 – third physical input to V2, Task1 Similarly, E2, E4, E6 will have target indices 0,1 and 2 respectively – i.e. first, second and third physical input to V2 Task2.

DataMovement events generated by an Input are routed to the corresponding upstream Input defined on the edge. Similarly data movement events generated by an Output are routed to the corresponding downstream Input defined on the edge.

If the Output is one of the Leaf Outputs for a DAG – it will typically not generate any events.

5.7 Error Handling

Reporting errors from an Input/Processor/Output Fatal Errors – fatal errors can be reported to Tez via the fatalError method available on the context instances, with which the I/P/O was initialized. Alternately, throwing an Exception from the initialize, close or run methods are considered to be fatal. Fatal errors cause the current running task to be killed. Actionable Non Fatal Errors – Inputs can report the failure to obtain data from a specific Physical connection by sending an InputReadErrorEvent via the InputContext. Depending on the Edge configuration, this may trigger a retry of the previous stage task which generated this data. Errors reported to an Input If the AM determines that data generated by a previous task is no longer available, Inputs which require this data are informed via an InputFailedEvent. The sourceIndex, targetIndex and attemptNumber information on this event would correspond to the DataMovementEvent event with the same values. The Input will typically handle this event by not attempting to obtain data based on the specific DataMovement event, and would wait for an updated DataMovementEvent for the same data.

Notes on Reader and Writer Tez does not enforce any interface on the Reader and Writer to stay data format agnostic. Specific Writers and Readers can be implemented for Key-Value, Record or other data formats. A KeyValue and KeyValues Reader/Writer interface and implementation, based on Hadoop serialization, is used by the Shuffle Input/Output provided by the Tez Runtime library.

6 Apache Tez Dynamic Graph Reconfiguration

6.1 Case Study: Automatic Reduce Parallelism

6.1.1 Motivation

Distributed data processing is dynamic by nature and it is extremely difficult to statically determine *optimal concurrency* and *data movement methods* a priori. More information is available during runtime, like *data samples and sizes*, which may help optimize the execution plan further. We also recognize that Tez by itself cannot always have the smarts to perform these *dynamic optimizations*. The design of Tez includes support for *pluggable vertex management* modules to collect relevant information from tasks and change the *dataflow graph* at runtime to optimize for **performance** and **resource usage** [7]. The diagram shows how we can determine an appropriate number of reducers in a MapReduce like job by observing the actual data output produced and the *desired load* per reduce task.

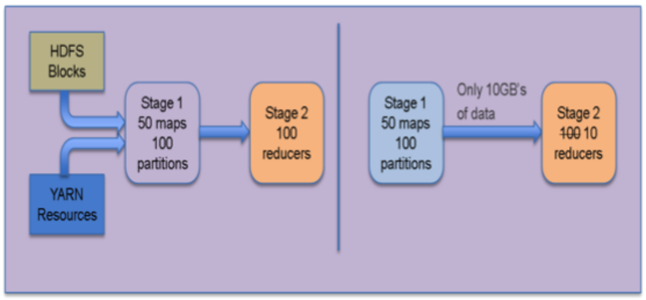

HDFS Blocks
YARN Resources
Stage 1 50 maps 100 partitions
Stage 2 100 reducers
Stage 1 50 maps 100 partitions
Only 10GB's of data
Stage 2 ~~100~~ 10 reducers

Figure 15: tez01

6.1.2 Performance & Efficiency via Dynamic Graph Reconfiguration

Tez envisions running computation by the most *resource efficient* and *high-performance* means possible given the runtime conditions in the cluster and the results of the previous steps of the computation. This functionality is constructed using a couple of basic building blocks

- **Pluggable Vertex Management Modules**: The control flow architecture of Tez incorporates a *per-vertex pluggable module* for user logic that deeply understands the data and computation. The vertex state machine invokes this user module at significant transitions of the state machine such as vertex start, source task completion etc. At these points the user logic can examine the runtime state and provide hints to the main Tez execution engine on attributes like vertex task parallelism.
- **Event Flow Architecture**: Tez defines a set of events by which different components like vertices, tasks etc. can pass information to each other. These events are routed from source to destination components based on a *well-defined routing logic* in the Tez control plane. One such event is the **VertexManager** event that can be used to send any kind of user-defined payload to the VertexManager of a given vertex.

6.2 Case Study: Reduce task parallelism and Reduce Slow-start

Determining the correct number of reduce tasks has been a long standing issue for Map Reduce jobs. The output produced by the map tasks is not known a priori and thus determining that number before job execution is hard. This becomes even more difficult when there are several stages of computation and the reduce parallelism needs to be determined for each stage. We take that as a case study to demonstrate the graph reconfiguration capabilities of Tez.

Reduce Task Parallelism: Tez has a **ShuffleVertexManager** that understands the semantics of *hash based partitioning* performed over a *shuffle transport layer* that is used in MapReduce. Tez defines a **VertexManager** event that can be used to send an arbitrary user payload to the vertex manager of a given vertex. The *partitioning tasks* (say the **Map tasks**) use this event to send *statistics* such as the size of the output partitions produced to the **ShuffleVertexManager** for the reduce vertex. The manager receives these events and tries to model the final output statistics that would be produced by the all the tasks. It can then advise the *vertex state machine* of the Reduce vertex to decrease the parallelism of the vertex if needed. The idea being to first *over-partition* and then determine the correct number at runtime. The *vertex controller* can cancel extra tasks and proceed as usual.

Reduce Slow-start/Pre-launch: Slow-start is a MapReduce feature where-in the reduce tasks are launched before all the map tasks complete. The hypothesis being that reduce tasks can start fetching the completed map outputs while the remaining map tasks complete. Determining when to pre-launch the reduce tasks is tricky because it depends on output data produced by the map tasks. It would be inefficient to run reduce tasks so early that they finish fetching the data and sit idle while the remaining maps are still running. In Tez, the *slow-start logic* is embedded in the **ShuffleVertexManager**. The vertex state controller informs the manager whenever a *source task* (here the **Map task**) completes. The manager uses this information to determine when to *pre-launch* the reduce tasks and how many to pre-launch. It then advises the vertex controller.

Its easy to see how the above can be extended to determine the correct parallelism for *range-partitioning* scenarios. The data samples could be sent via the **VertexManager** events to the vertex manager that can create the *key-range histogram* and determine the correct number of partitions. It can then assign the appropriate key-ranges to each partition. Thus, in Tez, this operation could be achieved without the overhead of a separate sampling job.

7 Re-Using Containers in Apache Tez

7.1 Motivation

Tez follows the traditional Hadoop model of *dividing a job into individual tasks*, all of which are run as processes via **YARN**, on the users' behalf – for *isolation*, among other reasons. This model comes with inherent costs – some of which are listed below.

- Process *startup* and *initialization* cost, especially when running a Java process is fairly high. For short running tasks, this initialization cost ends up being a significant fraction of the actual task runtime. *Re-using containers* can significantly reduce this cost.
- Stragglers have typically been another problem for jobs – where *a job runtime is limited by the slowest running task*. With reduced static costs per tasks – it becomes possible to run more tasks, each with a smaller work-unit. This reduces the runtime of stragglers (*smaller work-unit*), while allowing faster tasks to process additional work-units which can *overlap* with the stragglers.
- *Re-using containers* has the additional advantage of not needing to allocate each container via the **YARN ResourceManager (RM)** [8].

Other than helping solve some of the existing concerns, re-using containers provide additional opportunities for optimization where data can be *shared* between tasks.

7.2 Consideration for Re-Using Containers

7.2.1 Compatibility of containers

Each vertex in Tez specifies parameters, which are used when launching containers. These include the requested resources (memory, CPU etc), **YARN LocalResources**, the environment, and the command line options for tasks belonging to this **Vertex**. When a container is first launched, it is launched for a specific task and uses the parameters specified for the *task* (or *vertex*) – this then becomes the container's signature. An already running container is considered to be compatible for another task when the running container's signature is a superset of what the task requires.

7.2.2 Scheduling

Initially, when no containers are available, the Tez AM will request containers from the RM with location information specified, and rely on YARN's scheduler for locality-aware assignments. However, for containers which are being considered for re-use, the scheduling smarts offered by YARN are no longer available.

The **Tez scheduler** works with several parameters to take decisions on *task assignments – task-locality requirements*, *compatibility of containers* as described above, total available resources on the cluster, and the priority of pending task requests.

When a task completes, and the container running the task becomes available for re-use – a task may not be assigned to it immediately – as tasks may not exist, for which the data is local to the container's node. The Tez scheduler first makes an attempt to find a task for which the data would be *local* for the container. If no such task exists, the scheduler holds on to the container for a specific time, before actually allocating any *pending tasks* to this container. The expectation here, is that more tasks will complete – which gives additional opportunities for scheduling tasks on nodes which are close to the data. Going forward, *non-local containers* may be used in a speculative manner.

Priority of pending tasks (across different vertices), compatibility and cluster resources are considered to ensure that tasks which are deemed to be of higher priority (either due to a must-run-before relationship, failure, or due to specific scheduling policies) have an available container.

In the future, *affinity* will become part of the *scheduling decision*. This could be dictated by common resources shared between tasks, which need only be loaded by the first task running in a container, or by the data generated by the first task, which can then directly be processed by subsequent tasks, without needing to move/serialize the data – especially in the case of **One-to-One edges**.

7.3 Beyond simple JVM Re-Use

7.3.1 Cluster Dependent Work Allocation

At the moment, the number of tasks for a vertex, and their corresponding '**work-units**' are determined up front. Going forward, this is likely to change to a model, where a certain number of tasks are setup up front based on cluster resources, but *work-units* for these tasks are determined at runtime. This allows additional optimizations where tasks which complete early are given additional work, and also allows for better *locality-based assignment* of work.

7.3.2 Object Registry

Each Tez JVM (or *container*) contains an *object cache*, which can be used to *share data* between different tasks running within the same container. This is a simple **Key-Object store**, with different levels of *visibility/retention*. Objects can be cached for use within tasks belonging to the same Vertex, for all tasks within a **DAG**, and for tasks running across a **Tez Session** (more on Sessions in a subsequent post). The resources being cached may, in the future, be made available as a hint to the Tez Scheduler for affinity based *scheduling*.

7.3.3 Examples of usage:

1) **Hive** makes use of this *object registry* to cache data for **Broadcast Joins**, which is fetched and computed once by the first task, and used directly by remaining tasks which run in the same JVM.

2) The sort buffer used by `OnFileSortedOutput` can be cached, and re-used across tasks.

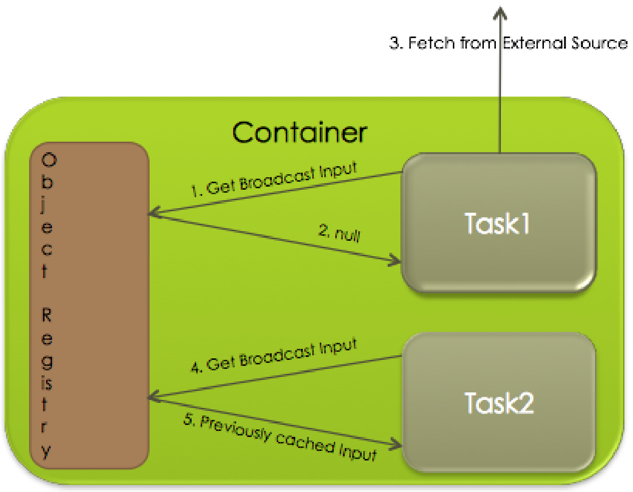


Figure 16: tez01

8 Tez Sessions

8.1 Introduction

Most relational databases have had a notion of sessions for quite some time. A database session can be considered to represent a connection between a user/application and the database or in more general terms, an instance of usage of a database. A session can encompass multiple queries and/or transactions. It can leverage common services, for example, caching, to provide some level of performance optimizations.

A Tez session, currently, maps to one instance of a Tez Application Master (AM). For folks who are familiar with YARN and MapReduce, you would know that for each MapReduce job, a corresponding MapReduce Application Master is launched. In Tez, using a Session, a user can can start a single Tez Session and then can submit DAGs to this Session AM serially without incurring the overhead of launching new AMs for each DAG.

8.2 Motivation for Tez Sessions

As mentioned earlier, the main proponents for Tez are Apache projects such as Hive and Pig. Consider a Pig script, the amount of work programmed into a script may not be doable within a single Tez DAG. Or let us take a common data analytics use-case in Hive where a user uses a Hive Shell for data drill-down (for example, multiple queries over a common data-set). There are other more general use-cases such as users of Hive connecting to the Hive Server and submitting queries over the established connection or using the Hive shell to execute a script containing one or more queries.

All of the above can leverage Tez Sessions [9].

8.3 Using Tez Sessions

Using a Tez Session is quite simple:

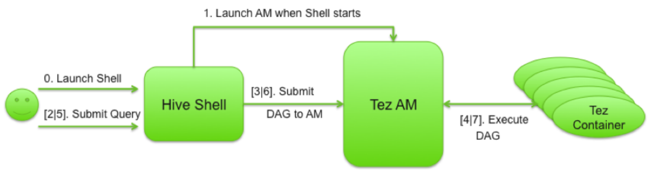


Figure 17: tez01

1. Firstly, instantiate a `TezSession` object with the required configuration using `TezSessionConfiguration`.
2. Invoke `TezSession::start()`
3. Wait for the TezSession to reach a ready state to accept DAGs by using the `TezSession::getSessionStatus()` api (this step is optional)
4. Submit a DAG to the Session using `TezSession::submitDAG(DAG dag)`
5. Monitor the DAG's status using the `DAGClient` instance obtained in step (4).
6. Once the DAG has completed, repeat step (4) and step (5) for subsequent DAGs.
7. Shutdown the Session once all work is done via `TezSession::stop()`.

There are some things to keep in mind when using a Tez Session:

- A **Tez Session** maps to a single **Application Master** and therefore, all resources required by any user-logic (in any subsequent DAG) running within the **ApplicationMaster** should be available when the **AM** is launched.
  - This mostly pertains to code related to the **VertexOutputCommitter** and any user-logic in the **Vertex** *scheduling* and *management* layers.
  - User-logic run in tasks is not governed by the above restriction.
- The resources (memory, CPU) of the **AM** are fixed so please keep this in mind when configuring the AM for use in a session. For example, memory requirements may be higher for a very large DAG.

8.4 Performance Benefits of using Tez Sessions

Container Re-Use. We know that *re-use* of containers was doable within a single DAG. In a Tez Session, containers are re-used even across DAGs as long as the containers are compatible with the task to be run on them. This vastly improves performance by not incurring the overheads of launching containers for subsequent DAGs. Containers, when not in use, are kept around for a *configurable period* before being released back to YARN's **ResourceManager**.

Caching with the Session. When running drill-down queries on common datasets, smarting caching of meta-data and potentially even caching of intermediate data or previous results can help improve performance. *Caching* could be done either within the **AM** or within *launched containers*. Such caching allows for more *fine-grained controls* with respect to caching policies. A *session-based cache* as compared to a global cache potentially provides more predictable performance improvements.

8.5 Example Usage of a Tez Session

The Tez source code has a simple **OrderedWordCount** example. This DAG is similar to the **WordCount** example in **MapReduce** except that it also orders the words based on their frequency of occurrence in the dataset. The DAG is an **MRR chain** i.e. a *3-vertex linear chain* of **Map-Reduce-Reduce**.

To run the **OrderedWordCount** example to process different data-sets via a single Tez Session, use:

```
bin/hadoop jar tez-mapreduce-examples-0.4.0-SNAPSHOT.jar
orderedwordcount -DUSE_TEZ_SESSION=true -DINTER_JOB_SLEEP_INTERVAL=0
/input1/ /output1/ /input2 /output2/ /input3/ /output3/ /input4/ /output4/
```

Below is a graph depicting the times seen when running *multiple* MRR DAGs on the same dataset (the dataset had 6 files to ensure multiple containers are needed in the map stage) in the same session. This test was run on my old MacBook running a single node Hadoop cluster having only one **DataNode** and one **NodeManager**.

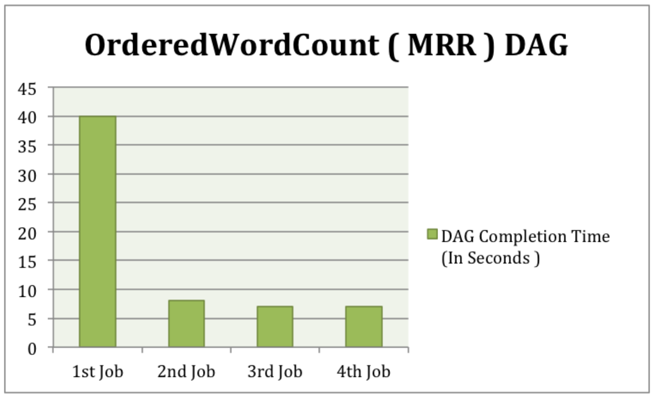


Figure 18: tez01

As you can see, even though this is just a simulation test running on a *very small data* set, leveraging containers across DAGs has a huge performance benefit.

References

[1] B. Saha, "Tez design," September 2013.

[2] ——, "Apache tez: A new chapter in hadoop data processing," September 2013. [Online]. Available: http://hortonworks.com/blog/apache-tez-a-new-chapter-in-hadoop-data-processing/

[3] P. J. Sadalage and M. Fowler, "Nosql distilled; a brief guide to the emerging world of polyglot persistence," June 2013.

[4] B. Saha, "Data processing api in apache tez," September 2013. [Online]. Available: http://hortonworks.com/blog/expressing-data-processing-in-apache-tez/

[5] ——, "Runtime api in apache tez," September 2013. [Online]. Available: http://hortonworks.com/blog/task-api-apache-tez/

[6] ——, "Writing a tez input/processor/output," October 2013. [Online]. Available: http://hortonworks.com/blog/writing-a-tez-inputprocessoroutput-2/

[7] ——, "Apache tez: Dynamic graph reconfiguration," October 2013. [Online]. Available: http://hortonworks.com/blog/apache-tez-dynamic-graph-reconfiguration/

[8] ——, "Reusing containers in apache tez," October 2013. [Online]. Available: http://hortonworks.com/blog/re-using-containers-in-apache-tez/

[9] ——, "Introducing tez sessions," November 2013. [Online]. Available: http://hortonworks.com/blog/introducing-tez-sessions/

Appendix