Tez Design

Disclaimer

This is a live document intended to describe feature set and broad design of Tez. Discussion of potential features and usage scenarios is included as motivation for the chosen design and to inform adopters of scenarios that are under consideration.

All details are subject to revision.

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 Map-Reduce job execution engine:

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

The support for 3rd 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. 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

Behavior of a Map-Reduce job under Hadoop 1.x

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).

Tez aims to be a general purpose execution runtime that enhances various scenarios that are not well served by classic Map-Reduce. In the short term the major focus is to support Hive and Pig, specifically to enable performance improvements to batch and ad-hoc interactive queries. Specific support for additional scenarios may be added in the future.

Systems similar in spirit to Tez have been developed academically and commercially. Some notable examples are Dryad, HTCondor, Clustera, Hyracks, and Nephele-PACTS.

Tez pre-requisites

For normal operation, Tez assumes:

- A cluster running YARN
- Access to a durable shared filesystem accessible through Hadoop Filesystem interface

Tez can operate in debugging scenarios that do not require a full cluster. This mode uses instantiations of HDFS and YARN running in a single process.

What services will Tez provide

Tez provides runtime components:

- An execution environment that can handle traditional map-reduce jobs
- An execution environment that handles DAG-based jobs comprising various built-in and extendable primitives.
- Cluster-side determination of input pieces.
- Runtime planning such as task cardinality determination and dynamic modification to the DAG structure.

Tez provides APIs to access these services:

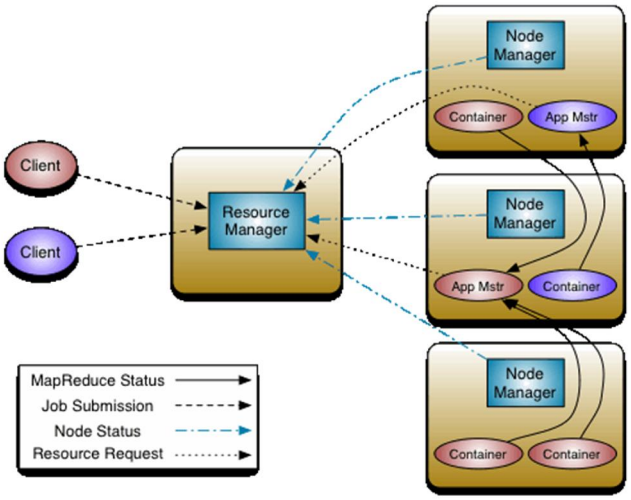
- Traditional map-reduce functionality is accessed via java classes written to the Job interface: org.apache.hadoop.mapred.Job and/or org.apache.hadoop.mapreduce.v2.app.job.Job; and by specifying in yarn-site that the map-reduce framework should be Tez.
- DAG-based execution is accessed via the new Tez DAG API: org.apache.tez.dag.api.*, org.apache.tez.engine.api.*.

Tez provides pre-made primitives for use with the DAG API (org.apache.tez.engine.common.*)

- Vertex Input
- Vertex Output
- Sorting
- Shuffling
- Merging
- Data transfer

Tez-YARN architecture

Tez is a framework that builds upon Apache Hadoop 2.0 (YARN). YARN provides cluster management and resource allocation services and Tez provides a new AppMaster that processes a new job definition. The Tez AppMaster calls on the YARN ResourceManager to allocate worker containers and calls the YARN NodeManagers to actually execute worker processes within the allocated containers.



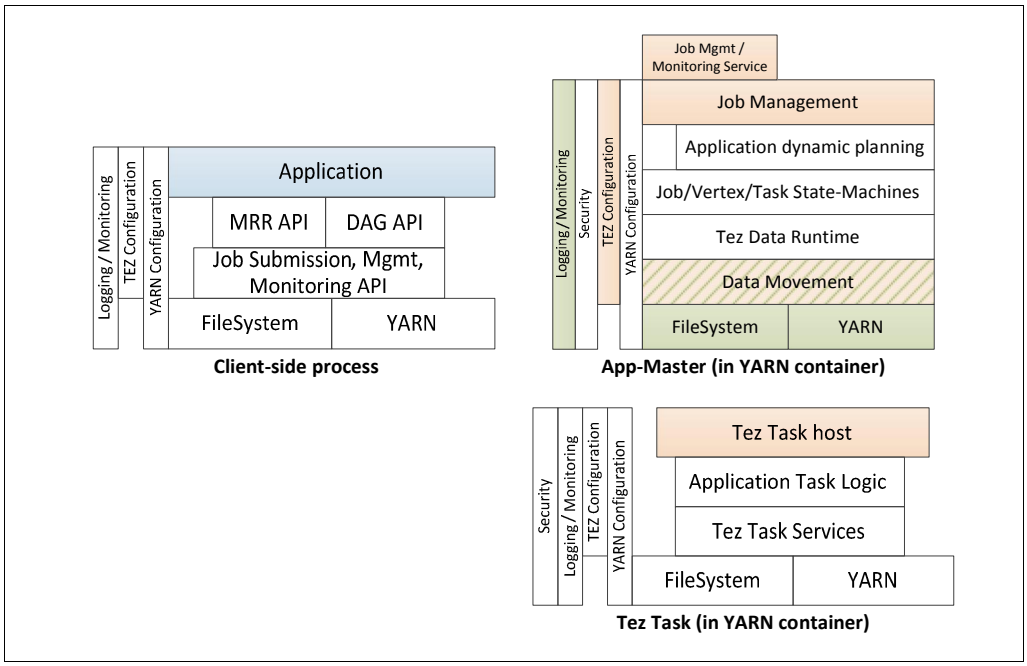
Hadoop 2.0 (YARN) Runtime Components

In the above figure Tez is represented by the red components: client-side API, an AppMaster, and multiple containers that execute child processes under the control of the AppMaster. The purple components represent either a second Tez job, an unrelated map-reduce job, or some other runtime that layers on YARN.

As far as YARN is concerned, the Containers allocated to a Tez job are all equivalent and are opaque. The Tez AppMaster takes full responsibility to use the containers to implement an effective job runtime. To do this the Tez AppMaster manages a rich state machine that tracks all the inputs of a job through all processing stages until all outputs are produced. The Tez AppMaster is responsible for dealing with transient container execution failures and must respond to RM requests regarding allocated and possibly deallocated Containers.

The Tez AppMaster is a single point of failure for a given job but given that many AppMasters can be executing on a YARN cluster the cost of AppMaster failure is largely mitigated by design. Check-pointing or similar techniques could be used to enable AppMaster restart should the cost of AppMaster failure be too great.

Three separate software stacks are involved in the execution of a Tez job, each using components from the client-application, Tez, and YARN:



Tez components

The application can be involved in the Job Management by providing an "application dynamic planning" module that will be accessed by the AppMaster. Because the AppMaster is not running on a special node nor running with privileged security access, security and system robustness are not compromised by running arbitrary application code in the AppMaster.

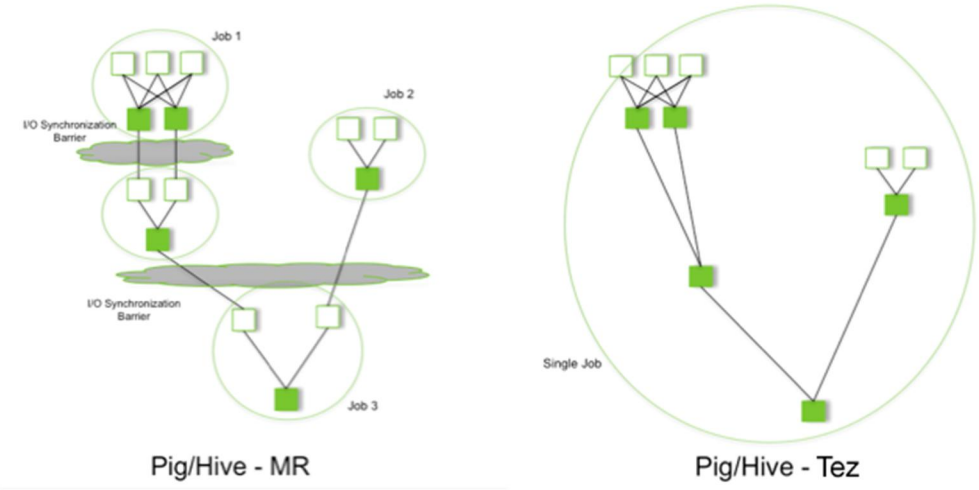
Future architecture-level capabilities that will be considered for inclusion:

- Task execution preemption
- Task execution check-pointing
- AppMaster execution check-pointing
- Task reuse (multiple tasks run in one container instance)
- Application sessions (e.g. containers kept alive and reused between client jobs)
- Post-serialization caches and other data-movement optimizations.

High level Feature Set

The general charter for Tez is to create an execution runtime that can support the DAG-style execution plans needed for Hive and Pig. The usefulness of DAG plans to relational query processing is well known and has been explored in research and mission-critical systems (various SQL engines, Dryad, etc.). The flexibility of DAG-style execution plans makes them useful for various data processing tasks such as iterative batch computing.

Ad-hoc execution plans may involve various stages of processing such as transformation of two input sets, perhaps with some aggregation, then a join between the inputs. When this type of plan is implemented via map-reduce primitives, there are an inevitable number of job boundaries which introduce overheads of read/write to durable storage and job startup, and which may miss out on easy optimization opportunities such as worker node reuse and warm caches.



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

Another issue with Map-Reduce is that it mandates a local sort of the output from each Map task. When the sort is required it is very convenient to have the runtime take care of it but when it is not required it is an unnecessary overhead. Tez increases the flexibility of task behaviors by eliminating mandatory behaviors.

Simple Tez query shapes

Hive can immediately benefit from the introduction of a few simple query plans.

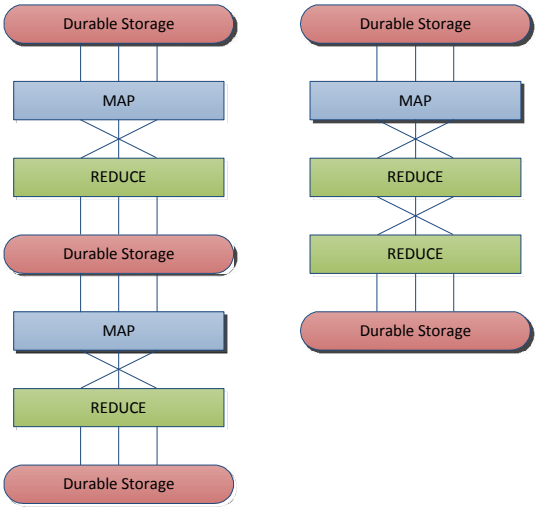
MRR*

Transformations of a single data set may involve multiple shuffles. For example a groupby–orderby query:

```
SELECT DeptName, COUNT(*) as c FROM EmployeeTable
GROUP BY DeptName ORDER BY c;
```

The map-reduce execution plan produced by Hive-0.9.0 and executed by Map-Reduce/Hadoop-1.x comprises two distinct map-reduce jobs.

A better execution plan can be implemented by structuring a Map-Reduce-Reduce job.



MR vs. MRR for Group-By-Order-By query

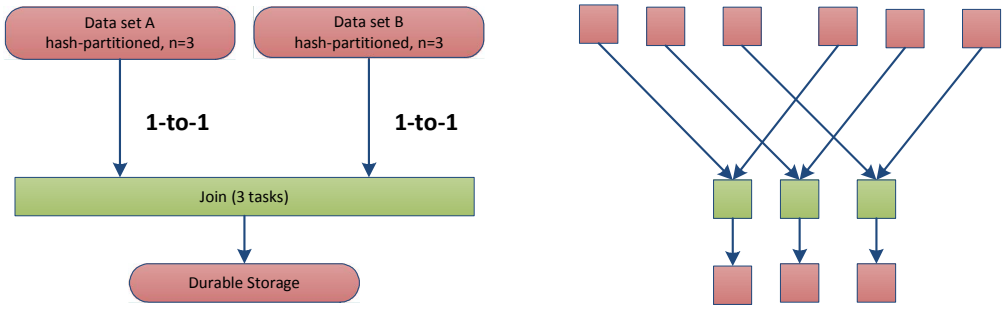
Join

Joins can be implemented using the map-reduce primitive but it is not a natural fit. The requirement that the primary input to map-reduce is a single homogeneous dataset is a frustration that must be worked around. The most general implementation is a reduce-side-join which introduces inefficiency by augmenting each true input row with an "input identifier" and writing the entire augmented inputs back to durable storage. Map-side joins either require precisely bucketed inputs or that one dataset be small and passed via a side-channel such as the Hadoop distributed cache.

A DAG execution plan can make use of multiple distinct input sets such that augmenting each input row is not necessary. There is no need to read/write to durable storage and various implementations for join can be naturally represented.

The most straightforward case is when two or more data sets are already hash-partitioned on the same key and with the same partition-count. Join can be implemented in this case by a job with DAG with a single vertex that takes two data sets as input, connected with 1-to-1 edges. At runtime, one partition from each data set is passed to each Task.

The diagram below illustrates the plan and runtime behavior for a Join on two inputs that each are hash-partitioned on the Join key. Each data set comprises three partitions.



DAG plan and corresponding runtime plan for binary join on two similarly hash-partitioned data sets

There are many other scenarios for Join that may be implemented using a variety of techniques and DAG structures. A complete discussion is beyond the scope of this document.

DAG topologies and scenarios

There are various DAG topologies and associated dynamic strategies that aid efficient execution of jobs. A comprehensive nomenclature and encompassing framework is not yet established, however we can describe various topologies and scenarios of interest.

The following terminology is used:

Job Vertex:	A "stage" in the job plan.
Job Edge:	The logical connections between Job Vertices.
Vertex:	A materialized stage at runtime comprising a certain number of materialized tasks
Edge:	Represents actual data movement between tasks.
Task:	A process performing computation within a YARN container.

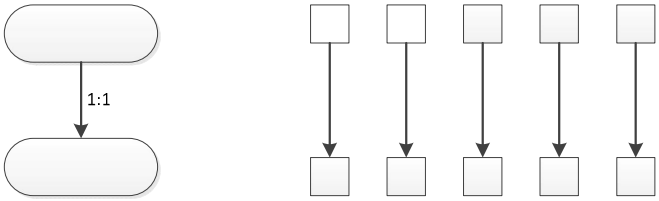
Task cardinality:	The number of materialized tasks in a Vertex.
Static plan:	Planning decisions fixed before job submission
Dynamic plan:	Planning decisions made at runtime in the AppMaster process.

In terms of responsibilities, the client application is responsible for the total *logical* outcome of the job whereas Tez is responsible for *physical* behavior. For example, Tez has no notion of hash-partitioning, rather it only knows about vertices, their connections and how to move data between them. It is the client's responsibility to ensure data semantics are correctly observed.

Tez 1-to-1 edge

A basic connection between job vertices that indication each Task in first stage has a 1:1 connection to Tasks in subsequent stage.

This type of edge appears in a variety of scenarios such as in the hash-Join plan discussed earlier. Job composition may also lead to graphs with 1-to-1 edges.



Job Plan Topology

Runtime Topology

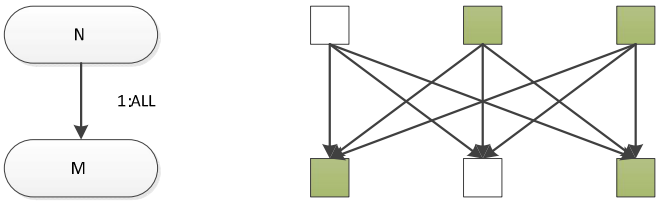
1-to-All

A 1-to-All edge indicates that each source task will produce data that is read by all of the destination tasks. There are a variety of ways to make this happen and so a 1-to-All edge may have properties describing its exact behavior.

Two important variants are described below

1-to-All (basic)

In a graph of two job vertices having cardinality N and M, a basic 1-to-All edge has each task produce M outputs, one for each of the M destination tasks. Each destination task receives one input from each of the source tasks. This shape is also known as *complete bipartite*.



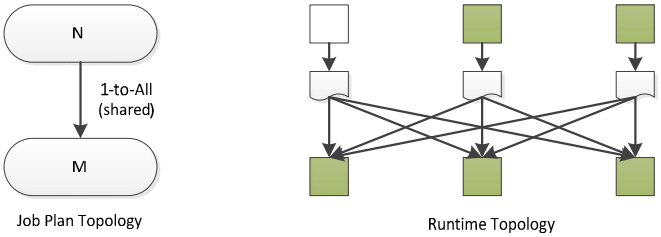
Job Plan Topology

Runtime Topology

The primary point to note is that the tasks in the first vertex must open M output targets and write to them individually. If implemented with real files or network ports this may not scale to support thousands of destination tasks.

1-to-All (shared)

A "1-to-All shared" edge has each task produce one output file and that file is made available to all the destination tasks. The primary use of this edge is to implement hash-partitioning without the need to open many outputs per task that would be required with the basic 1-to-All edge. The strategy is to have each source task produce a single output file that comprises a packed and indexed representation of M partitions. The destination tasks receive an identifier for each output, read the index, then only transfer their portion of the output.



1-to-All (shared) edge, for N=3 and M=3

Dynamic Task Cardinality

For relational query processing it is frequently necessary to support runtime decisions for task-cardinality. Some scenarios include:

1. A standard container might be known to handle X bytes of data without suffering OOM or taking excessively long. At runtime, each stage should create sufficient tasks such that each receives no more than X bytes of input.

2. A job hint might specify resource usage guidelines such as “Use as few resources as required to make progress” or “Use as many cluster resources as makes sense”.

3. Dynamic decision making may also involve how to schedule tasks against available container slots. If there are N files requiring processing but only M slots available, we have choices:
    a. run M tasks, and provide groups of files to each
    b. run N tasks but only activate M at a time
    c. file-sized based dynamic decisions.

Tez expects to supports these types of data-size constraints as needed by adopters.

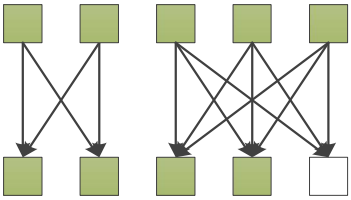
Skew Handling

Data skew is an ever present issue for relational queries as it routinely leads to a vertex in which the tasks have unequal data sizes to process. Various strategies such as over-partitioning and dynamic task cardinality are useful to resolve data-skew situations.

A particular cause of concern is a data-skew in which a large portion of rows share a common key. For example, if a key column contains many NULLs, or perhaps simply a frequent value such as “en-US”, then a hash-partitioning operation may yield very uneven buckets.

Some solutions can be implemented entirely via application logic. For example, http://www.nearinfinity.com/blogs/stephen_mouring_jr/2012/11/30/limiting-joins-in-hive.html describes how to use application logic to handle null-skew. The resulting query plans are better than naïve, but not necessarily optimal as they may introduce additional stages of processing.

Higher performance solutions may require runtime support. For example the runtime may allow a graph such as the following to be materialized:



Such a topology could be useful if the data was known to be separable into a problematic set and an easy set. The problematic set would receive more resources (i.e. more Tasks and Containers) and perhaps different task logic.

Tez can be extended to support advanced topologies as necessary.

Tez API

The Tez API comprises many services that support applications to run DAG-style jobs. An application that makes use of Tez will need to:

1. Create a job plan (the DAG) comprising vertices, edges, and data source references
2. Create task implementations that perform computations and interact with the DAG AppMaster
3. Configure Yarn and Tez appropriately

A detailed discussion of adopting Tez and consuming the APIs correctly is beyond the scope of this document. The APIs are listed here to give a flavor of what is "in the box".

A concrete example of a Tez adopter is the Tez Map-Reduce implementation. See the following for pointers:

org.apache.tez.mapreduce.YarnRunner

org.apache.tez.mapreduce.processor.MRTask

org.apache.tez.mapreduce.processor.map.MapProcessor

org.apache.tez.mapreduce.processor.reduce.ReduceProcessor

NOTE: All API details are subject to change. Please refer to latest codebase

DAG definition API

The Tez DAG API is not involved in data semantics, rather it is only concerned with data movement. Tez jobs are specified via calls to the DAG API to set up the job structure, register which classes will be used during task execution and so on.

org.apache.hadoop.yarn.api.records (defined in YARN codebase)

```
public abstract class Resource
  void setMemory(int);
  void setVirtualCores(int);
```

```
    int getMemory();
    int getVirtualCores();
```

org.apache.Tez.dag.api

```
  public class DAG
    DAG();
    void addVertex(Vertex);
    void addEdge(Edge);
    void addConfiguration(String, String);
    void setName(String);
    void verify();
    DAGPlan createDaG();

  public class Vertex
    Vertex(String vertexName, String processorName, int parallelism);
    void setTaskResource();
    void setTaskLocationsHint(TaskLocationHint[]);
    void setJavaOpts(String);
    String getVertexName();
    String getProcessorName();
    int getParallelism();
    Resource getTaskResource();
    TaskLocationHint[] getTaskLocationsHint();
    String getJavaOpts();

  public class VertexLocationHint
    VertexLocationHint();
    VertexLocationHint(int numTasks);
    VertexLocationHint(int numTasks, TaskLocationHint[] taskLocationHints);
    void setTaskLocationHints(TaskLocationHint[]);
    int getNumTasks();
    TaskLocationHint[] getTaskLocationHints();


  public static class VertexLocationHint.TaskLocationHint
    TaskLocationHint(String[] hosts, String[] racks)
    void setDataLocalHosts(String[] hosts)
    void setRacks(String[] racks);
    String[] getRacks();
    String[] getDataLocalHosts();

  public class Edge
    Edge(Vertex inputVertex,  Vertex outputVertex, EdgeProperty edgeProperty);
    String getInputVertex();
    String getOutputVertex();
    EdgeProperty getEdgeProperty();
    String getId();

  public enum EdgeProperty.ConnectionPattern {ONE_TO_ONE, ONE_TO_ALL, BIPARTITE }

  public enum EdgeProperty.SourceType {STABLE, STABLE_PERSISTED, STREAMING}

  public class EdgeProperty
```

```
    EdgeProperty(ConnectionPattern connectionPattern, SourceType sourceType,
                 String inputClass, String outputClass);
    ConnectionPattern getConnectionPattern();
    SourceType getSourceType();
    String getInputClass();
    String getOutputClass();
```

Execution APIs

These APIs support the runtime pieces of a job such as input processing, data processing and output.

org.apache.Tez.engine.api

```
  public interface Master
    // a context object for task execution. currently only stub

  public interface Input
    void initialize(Configuration conf, Master master)
    boolean hasNext()
    Object getNextKey()
    Iterable<Object> getNextValues()
    float getProgress()
    void close()

  public interface Output
    void initialize(Configuration conf, Master master);
    void write(Object key, Object value);
    OutputContext getOutputContext();
    void close();

  public interface Partitioner
    int getPartition(Object key, Object value, int numPartitions);

  public interface Processor
    void initialize(Configuration conf, Master master)
    void process(Input[] in, Output[]  out)
    void close()

  /* Task is the base Tez entity which consumes
   * input key/value pairs through an Input pipe,
   * processes them via a Processor
   * produces output key/value pairs for an Output pipe.
   */
  public interface Task
    void initialize(Configuration conf, Master master)
    Input[] getInputs();
    Processor getProcessor();
    Output[] getOutputs();
    void run()
    void close()
```

Comment [b1]:
Tez-157 will change Input/Output to no longer have "key/value" concepts but rather simply ByteStream concepts.

```
    Configuration getConfiguration();
```

org.apache.tez.common

```
public abstract class TezTaskContext
    TezDAGID getDAGID();
    String getUser();
    String getJobName();
    String getVertexName();
    void statusUpdate();
    public void write(DataOutput out);
    public void readFields(DataInput in);

  public class TezEngineTaskContext extends TezTaskContext
    public List<InputSpec> getInputSpecList();
    public List<OutputSpec> getOutputSpecList();


  public class InputSpec
    public String getVertexName();
    public int getNumInputs();
    public String getInputClassName();

  public class OutputSpec
    public String getVertexName();
    public int getNumOutputs();
    public String getOutputClassName();

  public interface TezTaskReporter
    <various progress related APIs>

  public interface TezTaskUmbilicalProtocol extends Master
    ContainerTask getTask(ContainerContext containerContext)
    boolean statusUpdate(TezTaskAttemptID taskId, TezTaskStatus taskStatus)
    void reportDiagnosticInfo(TezTaskAttemptID taskid, String trace);
    boolean ping(TezTaskAttemptID taskid) throws IOException;
    void done(TezTaskAttemptID taskid) throws IOException;
    void commitPending(TezTaskAttemptID taskId, TezTaskStatus taskStatus)
    boolean canCommit(TezTaskAttemptID taskid) throws IOException;
    void shuffleError(TezTaskAttemptID taskId, String message) throws IOException;
    void fsError(TezTaskAttemptID taskId, String message) throws IOException;
    void fatalError(TezTaskAttemptID taskId, String message) throws IOException;
```

Comment [ML2]: These will be extended to support multiple inputs/outputs.

org.apache.Tez.engine.records

```
  public interface OutputContext
    int getShufflePort();
    public void write(DataOutput out);
    void readFields(DataInput in);
```

Comment [SS3]: May likely be removed, and moved into a custom OutputContext for the Output being used.

Tez API Configuration

Comment [SS4]: Configuration is very much a work in progress.
Things requiring configuration:
-AppMaster configuration
-Task configuration
-Task building-block configuration

Most of the properties mentioned below are config parameters for the AppMaster.
Configuring running tasks is still WIP – and is really dependent on how individual building blocks need to be configured. Currently, this is a mix between TezContext and Configuration for the OnDiskSortedOutput, Shuffle, etc.

TezConfiguration source:
/tez-dag-api/src/main/java/org/apache/tez/dag/api/TezConfiguration.java

The following configuration settings are accurate Tez-0.2.0:

Environment variables:

```
$env:Tez_CONF_DIR
$env:Tez_HOME
```

Standard configuration file:

```
Tez-site.xml
```

Configuration settings

```
Tez.staging-dir                                    default = "/tmp/hadoop-yarn/staging"
Tez.jobSubmitDir
Tez.task.listener.thread-count                     default = 30
Tez.container.listener.thread-count                default=30
Tez.maxtaskfailures.per.node                       default=3
Tez.node-blacklisting.enabled                      default=true
Tez.node-blacklisting.ignore-threshold-node-percent  default=33
Tez.resource.memory.mb                             default=1024;
Tez.resource.cpu.vcores                            default=1
Tez.slowstart-vertex-scheduler.min-src-fraction    default=0.5
Tez.slowstart-vertex-scheduler.max-src-fraction    default=0.8;
Tez.application.classpath                          default=<see below>
```

Comment [b5]: These will likely be removed.

Default classpath

```
$env:Tez_CONF_DIR,
$env:Tez_HOME_ENV + "/*",
$env:Tez_HOME_ENV + "/lib/*"
```

Job monitoring APIs

org.apache.Tez.dag.api.client

```
public interface DAGClient
  List<String> getAllDAGs();
  DAGStatus getDAGStatus(String dagId);
  VertexStatus getVertexStatus(String dagId, String vertexName);

public enum DAGStatus.State
  { SUBMITTED,  INITING,  RUNNING,  SUCCEEDED,  KILLED,  FAILED, ERROR};
```

```
public class DAGStatus
  State getState();
  List<String> getDiagnostics();
  Progress getProgress();
  Map<String,Progress> getVertexProgress();

public class Progress
  int getTotalTaskCount();
  int getSucceededTaskCount();
  int getRunningTaskCount();
  int getFailedTaskCount();
  int getKilledTaskCount();

public enum VertexStatus.State
  {INITED, RUNNING, SUCCEEDED, KILLED, FAILED, ERROR}

public class VertexStatus
  State getState();
  List<String> getDiagnostics();
  Progress getProgress();
```

References

Tez/YARN raw materials:

http://hortonworks.com/blog/introducing-tez-faster-hadoop-processing/

http://hortonworks.com/blog/45x-performance-increase-for-hive/

http://hortonworks.com/blog/moving-hadoop-beyond-batch-with-apache-yarn/

http://hortonworks.com/blog/moving-hadoop-beyond-batch-with-apache-yarn/?mkt_tok=3RkMMJWWfF9wsRolsqTPZKXonjHpfsX57uUkXKO2lMI%2F0ER3fOvrPUfGjI4DRcRjI%2BSLDwEYGJlv6SgFT7TMMbFh1rgNUxc%3D

Clustera: http://www.vldb.org/pvldb/1/1453865.pdf

Dryad: http://research.microsoft.com/en-us/projects/DryadLINQ/

Condor: http://research.cs.wisc.edu/htcondor/