KIWI Project@Data Management Research Section
ETRI Technical Report 14ZS1410-TR-65

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*.

This report amis to analyze the insights of the design aspects of the Apache Tez. In exploring the questions of the designing an efficient distributed execution engine such as the Tez, this technical report will be limited to consideration of major capabilities and their application programming interface (API) structures of the DAG execution engine.

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

1.1 Motivation

Distributed data processing is the core application that Apache Hadoop is built around [1]. 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.

The common behavior of a Map-Reduce job under Hadoop 1.x is as follows.

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

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. Fundamentally, YARN resource scheduling is a **2-step framework** with *resource allocation* done by YARN and *task scheduling* done by the application. This allows YARN to be a generic compute platform while still allowing flexibility of scheduling strategies. An analogy would be general purpose operating systems that allocate computer resources among concurrent processes.

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 [2]. 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.

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*).

The **YARN Resource Manager** service is the central controlling authority for resource management and makes allocation decisions as shown in Figure 1. It exposes a Scheduler API that is specifically designed to negotiate resources and not schedule tasks. Applications can request resources at different layers of the cluster topology such as nodes, racks etc. The scheduler determines how much and where to allocate based on resource availability and the configured sharing policy.

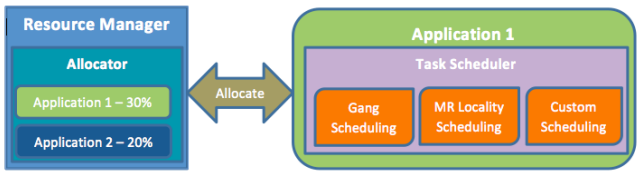

Figure 1: The resource manager in the YARN.

Currently, there are two sharing policies *fair scheduling* and *capacity scheduling*. Thus, the API reflects the Resource Managers role as the resource allocator. This API design is also crucial for Resource Manager scalability because it limits the complexity of the operations to the size of the cluster and not the size of the tasks running on the cluster.The actual task scheduling decisions are delegated to the application manager that runs the application logic. It decides when, where and how many tasks to run within the resources allocated to it. It has the flexibility to choose its locality, co-scheduling, co-location and other scheduling strategies.

1.2 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 [3]. 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).

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.

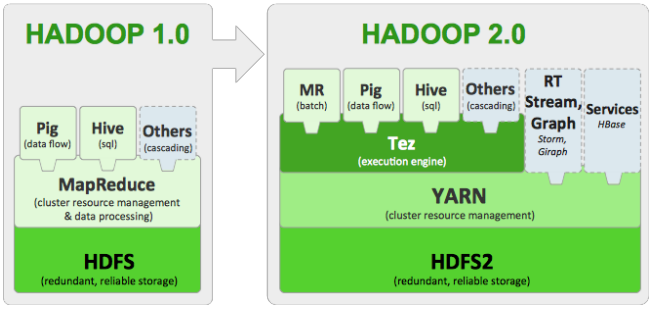

Figure 2: *Monolithic* Hadoop 1.0 vs. *Layered* Hadoop 2.0

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 as shown in Figure 2. 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.

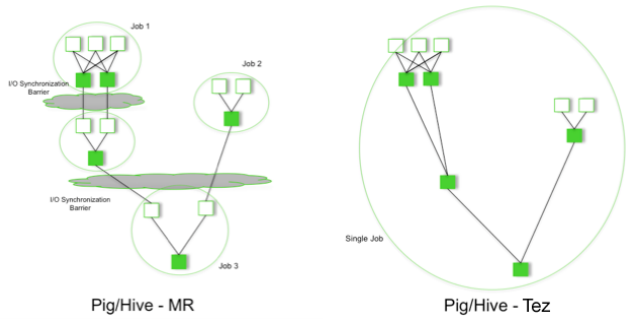

Figure 3: General example of a Map-Reduce execution plan compared to DAG execution plan

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. Figure 3 illustrates the advantages provided by Tez for complex SQL queries in Apache Hive or complex Apache Pig scripts.

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.

Major community achievements in the recent Tez were:

- **Application Recovery** – This is a major improvement to the Tez framework that preserves work when the job controller (YARN Tez Application Master) gets restarted due to node loss or cluster maintenance. When the Tez Application Master restarts, it will recover all the work that was already completed by the previous master. This is especially useful for long running jobs where restarting from scratch would waste work already completed.

- **Stability for Hive on Tez** – We did considerable testing with the Apache Hive community to make sure the imminent release of Hive 0.13 is stable on Tez. We appreciate the great partnership.

- **Data Shuffle Improvements**– Data shuffling re-partitions and re-distributes data across the cluster. This is a major operation in distributed data processing, so performance and stability are important. Tez 0.4 includes improvements in memory consumption, connection management, and in the handling of errors and empty partitions.

- **Windows Support** – The community fixed bugs and made changes to Tez so that it runs as smoothly on Windows as it does on Linux. We hope this will encourage adoption of Tez on Windows-based systems.

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 Dataflow Graph Representation

Expressing the Computation: 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, Figure 4 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*.

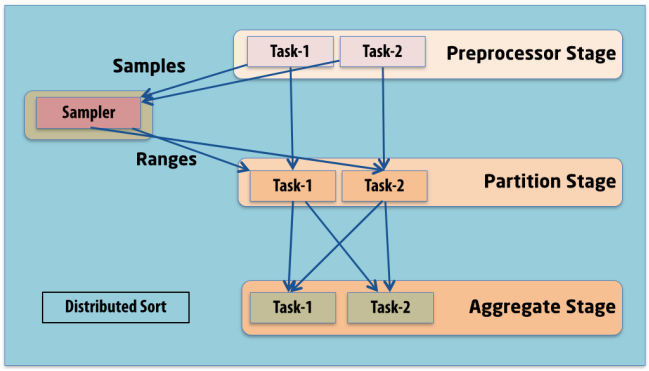

Figure 4: The representation of the data processing flow

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. Tez provides the following APIs to define the processing.

- **DAG API**
  - Defines the structure of the data processing and the relationship between producers and consumers.
  - Enable definition of complex data flow pipelines using simple graph connection APIs. Tez expands the logical DAG at runtime.
  - This is how all the tasks in the job get specified.

- **Runtime API**
  - Defines the interfaces using which the framework and application code interact with each other.
  - Application code transforms data and moves it between tasks.
  - This is how we specify what actually executes in each task on the cluster nodes.

2.2 Flexible 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 Figure 5, we can see how composing different Inputs, Outputs and Processors can produce different tasks.

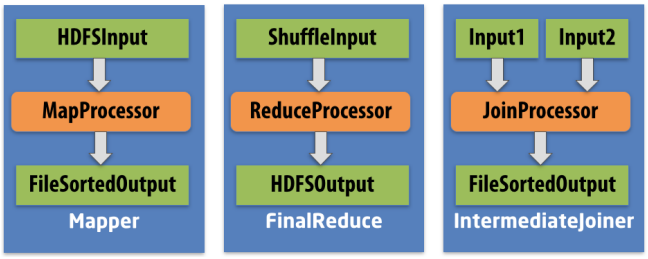

Figure 5: Flexible Input-Processor-Output runtime model

2.3 Dynamic Graph Reconfiguration

Late Binding: 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. Figure 6 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.

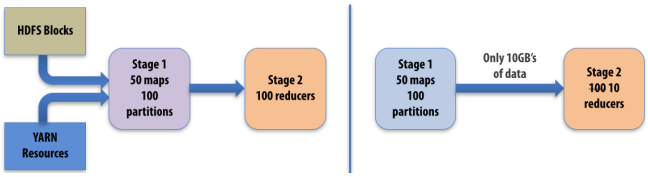

Figure 6: Plan Reconfiguration at Runtime

2.4 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*. Figure 7 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.

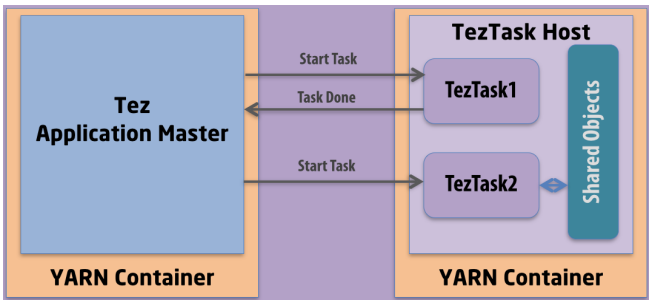

Figure 7: Optimal Resource Management

3 Data Processing API

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 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

work *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.

```
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);
```

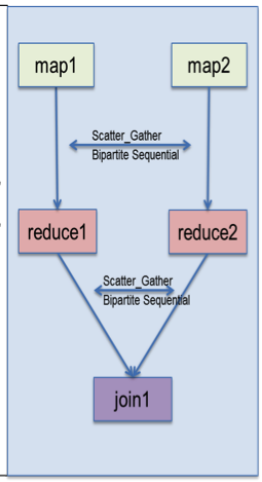

Figure 8: A dataflow logical graph using the DAG API

Figure 8 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. Figure 9 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.

3.1 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

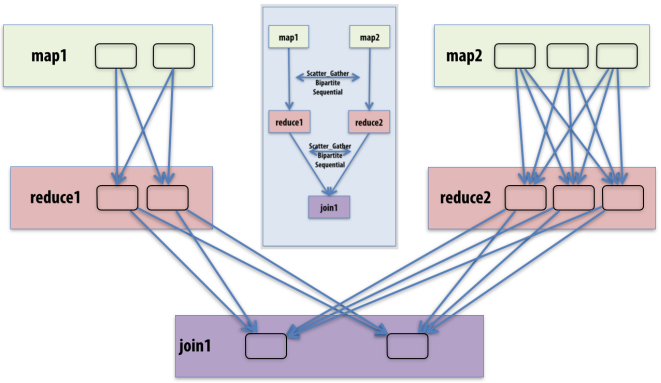

Figure 9: Expanding the logical graph at runtime

  - *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

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*.

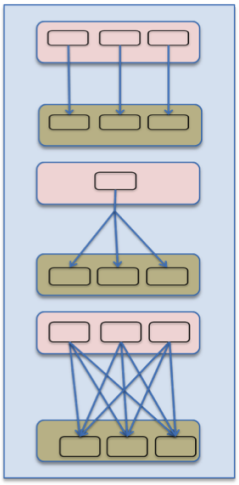

Figure 10: Data movement patterns (*one-to-one, broadcast, scatter-gather*)

DAG Topologies and Scenarios

There are various DAG *topologies* and associated dynamic strategies that aid efficient execution of jobs.

One-to-One 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*. Job composition may also lead to graphs with 1-to-1 edges.

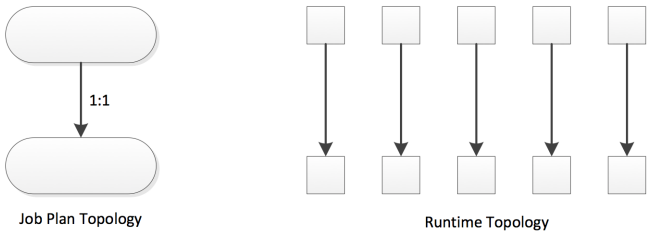

Figure 11: One-to-One edge

A *One-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 One-to-All edge may have properties describing its exact behavior. Two important variants are described below. **One-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*.

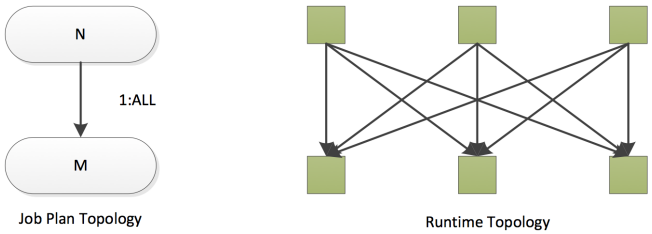

Figure 12: One-to-All basic edge

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.

One-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.

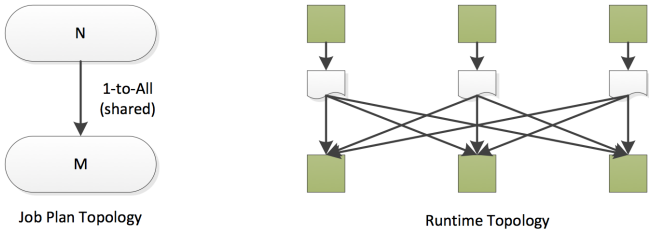

Figure 13: One-to-All shared edge

Dynamic Task Cardinality

For relational query processing it is frequently necessary to support *runtime decisions* for *task-cardinality* [2]. 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.

4 Runtime API

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

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 as shown in Figure 14 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.

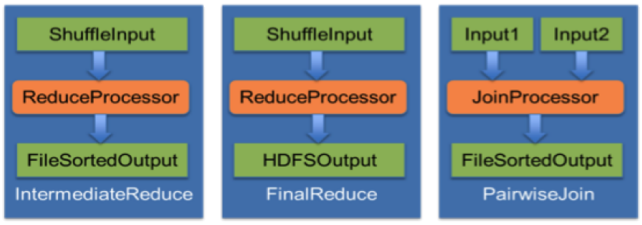

Figure 14: An example of the Reduce stage within an MR jobs

4.2 Tez Runtime API

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.

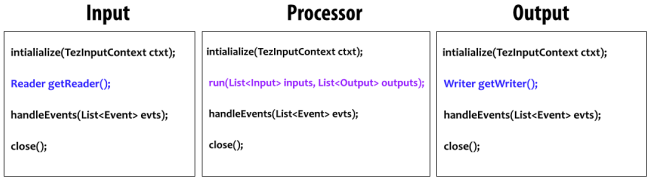

Figure 15: Tez runtime API (Input, Processor and Output)

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 used to communicate between the tasks and between ApplicationMaster (AM). *Data Movement Event* used by producer task to inform the consumer task about data location, size etc. *Input Error event* sent by task to AM to inform about errors in reading input. AM then takes action by re-generating the input. Other events to send task completion notification, data statistics and other control plane information. Similarly, *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* as shown in Figure 16 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*.

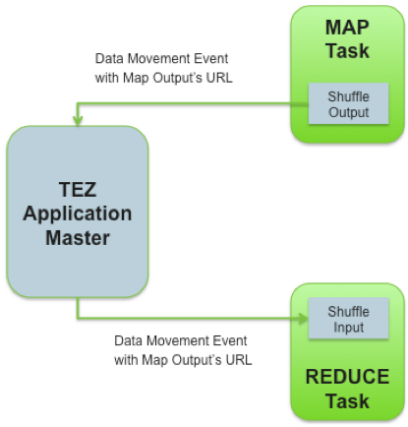

Figure 16: An example of the Suhffle Event

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 Implementations

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.

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.

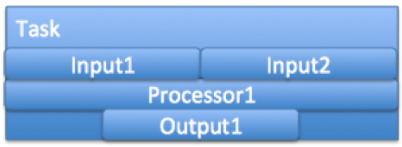

Figure 17: A single Tez task

Figure 17 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.

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.

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.
2. Providing information to Tez (in effect the subsequent Input) on where this data is available.

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).

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. For more detail, see Listing 1 (OnFileSortedOutput.java) in the Appendix.

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.

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.

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. For more detail, see Listing 2 (ShuffledMergedInput.java) and Listing 3 (ShuffledUnorderedKVInput.java) in the Appendix.

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.

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 DataMovementEvent

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.

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.

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

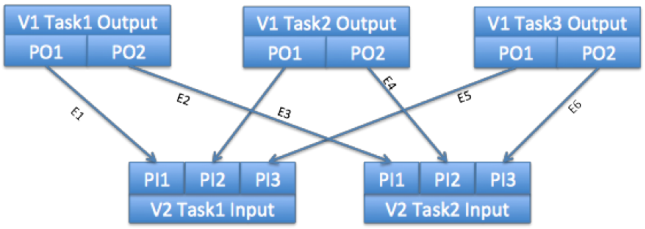

Figure 18: An example of data movement events

- E2, sourceIndex = 1 (since it is generated by the 2nd physical output)

2. Similarly Vertex1, Task2 and Task3 will generate two data movement events each.
   - E3 and E5, sourceIndex=0
   - E4 and E6, sourceIndex=1
3. 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
4. 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 *InputReaderErrorEvent* 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: 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 Dynamic Graph Reconfiguration

6.1 Case Study: Automatic Reduce Parallelism

Performance and Efficiency

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

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)** [7].

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

7.1 Consideration for Re-Using Containers

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.

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*.

7.2 Beyond simple JVM Re-Use

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.

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*.

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.

8 Tez Sessions

Most relational databases have had a notion of sessions for quite some time. A database session can be considered to represent

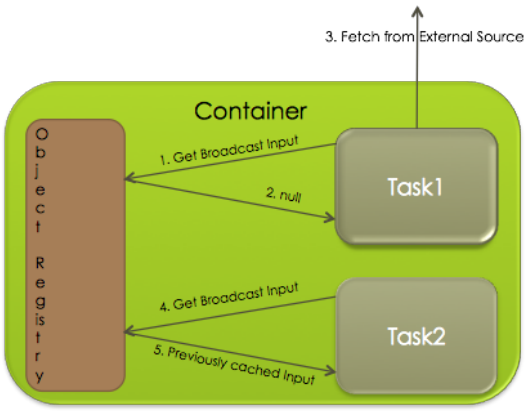

Figure 19: Container reuse in the Tez

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.

Motivation: 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 [8].

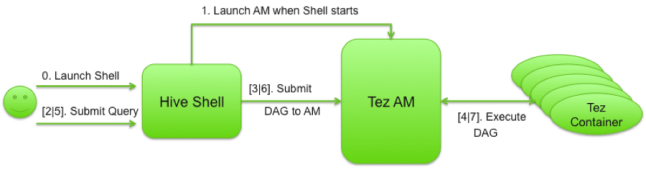

Figure 20: Tez sessions

8.1 Using Tez Sessions

Using a Tez Session is quite simple:

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.

8.2 Performance Benefits

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.

8.3 Example Usage

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:

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

Figure 21 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*.

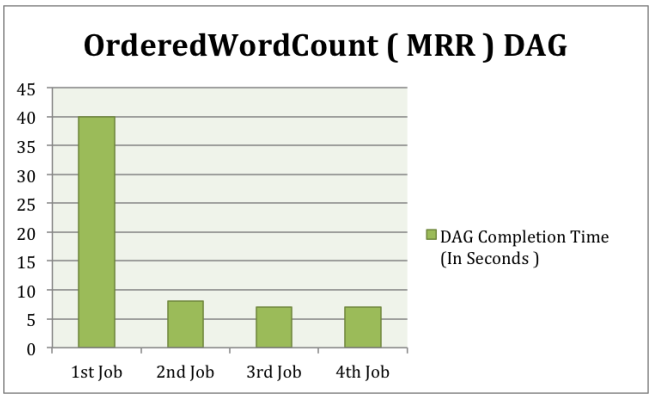

Figure 21: Performance results of multiple MRR DAGs on the same dataset

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.

9 Discussion and Future Work

Let us briefly compare the five tools for SQL on Hadoop based on their potential advantages (SQL mode, SQL ANSI completeness, client access methods, file format support and data sources). There are many different methods and tools for interacting and querying data within Hadoop. The most widely used tools allow for SQL based querying of the data. The following article summarizes a great comparison by MapR of the most common SQL on Hadoop technologies available today [9].

Initially developed by Facebook, Apache Hive is a data warehouse infrastructure that is built on top of Hadoop. It allows querying data stored on HDFS for analysis via HQL, an SQL-like language that is translated to MapReduce jobs. Although it seems to provide SQL functionality, Hive still runs jobs on Hadoop as batch processing and does not provide interactive

SQL Mode	Hive	Drill	Impala	Presto	Spark/Shark
	Batch	Interactive	Interactive	Interactive	In-memory /streaming

Figure 22: A Common Tool Comparison: SQL Mode

querying. It stores metadata in a relational database and requires maintaining a schema for the data. Only four file formats are supported by Hive: text, SequenceFile, ORC and RCFile. Hive supports processing compressed on Hadoop and also user defined functions.

SQL ANSI Completeness	Hive	Drill	Impala	Presto	Spark/Shark
SELECT query	Medium	Medium	Medium	Medium	Medium
DDL/DML	Medium	Low	Low		Medium
Packaged Analytic functions	Low			Low	
UDFs/Custom functions	High	Low	Low		High

Figure 23: SQL ANSI Completeness

Cloudera's Impala is a query engine that runs on top of Hadoop and executes interactive SQL queries on HDFS and HBase. While Hive runs in batch processing, Impala runs the queries in real-time, thus integrating SQL based business intelligence tools with Hadoop. Although Cloudera is the main developer behind this tool, it is fully open source and supports the following file formats: text, LZO, SequenceFile, Avro and RCFile. Impala can also run on the cloud via Amazons Elastic MapReduce.

Client Access	Hive	Drill	Impala	Presto	Spark/Shark
Shell	Yes	Yes	Yes	Yes	Yes
JDBC	Yes	Yes	Yes	Yes	Yes
ODBC	Yes	Yes	Yes		Yes

Figure 24: Client Access Methods

Presto is also an interactive SQL query engine. It runs on top of Hive, HBase, and even relational databases and proprietary data stores, thus combining data from multiple sources across the organization. Facebook is the main developer behind Presto and the company uses it to query internal data stores, including a 300PB data warehouse. Airbnb and Dropbox also use Presto, so it seems tried and tested for the enterprise.

SQL-on-Hadoop Challenges

File Formats: While Hadoop supports storing all file formats,

Common File Format Support	Hive	Drill	Impala	Presto	Spark/Shark
Text	Yes		Yes	Yes	Yes
CSV	Yes	Yes			Yes
Sequence	Yes		Yes	Yes	Yes
RC	Yes		Yes	Yes	Yes
ORC	Yes				
Parquet	Yes	Yes	Yes		
Avro	Yes		Yes		Yes
JSON	Yes	Yes			Yes
Compression	Yes		Yes		Yes
Hive SerDe	Yes	Yes			Yes

Figure 25: Common File Format Support

Data Sources	Hive	Drill	Impala	Presto	Spark/Shark
Files	Yes	Yes	Yes	Yes	Yes
HBase	Yes	Yes	Yes		Yes
Query non-Hadoop sources?		Yes		Yes	
Data Types	**Hive**	**Drill**	**Impala**	**Presto**	**Spark/Shark**
Relational	Yes	Yes	Yes	Yes	Yes
Complex	Yes	Yes		Yes	Yes
Metadata	**Hive**	**Drill**	**Impala**	**Presto**	**Spark/Shark**
Hive Metadata Store	Yes	Yes	Yes	Yes	Yes

Figure 26: Data Sources, Data Types and Metadata

SQL-on-Hadoop technologies require data to be in rigid formats in order to process it. Therefore, they might not support storing and querying all of an organizations data, which can arrive in various formats or in no format at all. Jethro and other tools require using their own file structure. Impala is fully compatible with text files and Parquet, a columnar storage format for Hadoop, while providing partial support for other formats. Presto is supposed to work with Hadoop file formats such as text, RCFile, and SquenceFile. Hive supports implementing a custom serializer/deserializer function that can read/write any file format, but this requires extra programming [10].

Server Maintenance: Some SQL-on-Hadoop technologies such as CitusDB, Hadapt, and BigSQL require PostgreSQL to be installed on each node in the cluster. This could be cumbersome to deploy and maintain, especially when dealing with large clusters.

Schema Maintenance: One of Hadoops advantages is the lack of schema. However, making SQL available on Hadoop requires defining and managing a schema, something which may present a problem when new data comes in that does not fit the schema. Hadapt, one of the SQL-on-Hadoop solutions, claims it does not require schema definition for self descriptive JSON or XML formats, but this ability is already available with standard SQL databases.

ACID: SQL databases support ACID (Atomicity, Consistency, Isolation, Durability) to guarantee reliable database transactions. Hadoop does not support it, so it is up to the relevant technology to provide it, if it does provide it at all. Hive plans to support ACID in the future.

OLTP: Since Hadoop is based on sequential reads and does not support updates, it is a lot more useful for On-line Analytical Processing (OLAP) by definition. Therefore, Hive, which is based on MapReduce, does not support On-line Transaction Processing (OLTP) since MapReduce does not do single row operations (future support is planned as part of ACID). Although other tools are not based on MapReduce, they still target analytical queries. HBase does provide transactional functionality, although it isnt ACID compliant yet.

SQL Functionality: SQL supports more features than just queries such as views, stored procedures, and user defined functions. Most SQL-on-Hadoop tools do not support them and require writing extra code instead - e.g. Java for Hive and C++ for Impala.

Update Statements: Unlike SQL, HDFS does not support update statements. SQL-on-Hadoop tools may implement it, but it isnt clear exactly how since it requires random read/write access to all the data on Hadoop, a feature that Hadoop does not provide. Maybe they implement it like HBase which uses in-memory indexes and compacts files once in a while to remove older versions.

Joins and Dimensions: HDFS automatically manages how to spread blocks of data over the cluster, and this process cannot be controlled manually. In certain cases this could be counter-productive. Saving several pieces of data together on the same node or maybe on all of the cluster nodes could be necessary to help execute joins and dimensions more efficiently, data such as product names, categories, or clients. Otherwise it could take much more time to bring all the relevant data together from across the network.

Summary Hadoop works quite differently from SQL and requires learning new concepts and technologies. SQL-on-Hadoop tools can help bridge this knowledge gap. The best strategy though is to use each technology for its strength rather than to bend it into something else - SQL for transactional queries and Hadoop for batch processing.

10 Conclusion

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 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.
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.
- DAG-based execution is accessed via the new Tez DAG API.

Tez provides pre-made primitives for use with the *DAG API*.

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

We expect 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.

As the Hive and Pig projects adapt to use Tez, we expect that the repository of various implementations of Inputs/Outputs/Processors will grow to house a common set of building blocks for use across the different projects.

References

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

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

[3] B. Saha and H. Blog, “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/

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

[5] B. Saha and H. Blog, “Runtime api in apache tez,” September 2013. [Online]. Available: http://hortonworks.com/blog/task-api-apache-tez/

[6] H. Blog and B. Saha, “Writing a tez input/processor/output,” October 2013. [Online]. Available: http://hortonworks.com/blog/writing-a-tez-inputprocessoroutput-2/

[7] Hortonworks and B. Saha, “Reusing containers in apache tez,” October 2013. [Online]. Available: http://hortonworks.com/blog/re-using-containers-in-apache-tez/

[8] B. Saha and H. Blog, “Introducing tez sessions,” November 2013. [Online]. Available: http://hortonworks.com/blog/introducing-tez-sessions/

[9] D. Intelligence, “Sql on hadoop a common tool comparison,” Feb 2014. [Online]. Available: http://discoveredintelligence.ca/sql-on-hadoop-tool-comparison/

[10] Y. Mor, “Eight sql on hadoop challenges,” April 2014. [Online]. Available: https://www.xplenty.com/blog/2014/04/eight-sql-on-hadoop-challenges/

Appendix

```java
package org.apache.tez.runtime.library.output;

import java.io.IOException;
import java.nio.ByteBuffer;
import java.util.BitSet;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.atomic.AtomicBoolean;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.yarn.api.ApplicationConstants;
import org.apache.tez.common.TezJobConfig;
import org.apache.tez.common.TezRuntimeFrameworkConfigs;
import org.apache.tez.common.TezUtils;
import org.apache.tez.common.counters.TaskCounter;
import org.apache.tez.runtime.api.AbstractLogicalOutput;
import org.apache.tez.runtime.api.Event;
import org.apache.tez.runtime.api.events.
    CompositeDataMovementEvent;
import org.apache.tez.runtime.api.events.VertexManagerEvent
    ;
import org.apache.tez.runtime.library.api.KeyValueWriter;
import org.apache.tez.runtime.library.common.
    MemoryUpdateCallbackHandler;
import org.apache.tez.runtime.library.common.sort.impl.
    ExternalSorter;
import org.apache.tez.runtime.library.common.sort.impl.
    PipelinedSorter;
import org.apache.tez.runtime.library.common.sort.impl.
    TezIndexRecord;
import org.apache.tez.runtime.library.common.sort.impl.
    TezSpillRecord;
import org.apache.tez.runtime.library.common.sort.impl.dflt
    .DefaultSorter;
import org.apache.tez.runtime.library.shuffle.common.
    ShuffleUtils;
import org.apache.tez.runtime.library.shuffle.impl.
    ShuffleUserPayloads.DataMovementEventPayloadProto;
import org.apache.tez.runtime.library.shuffle.impl.
    ShuffleUserPayloads.VertexManagerEventPayloadProto;

import com.google.common.base.Preconditions;
import com.google.common.collect.Lists;
import com.google.protobuf.ByteString;

/**
 * <code>OnFileSortedOutput</code> is an {@link
     AbstractLogicalOutput} which sorts key/value pairs
 * written to it and persists it to a file.
 */
public class OnFileSortedOutput extends
    AbstractLogicalOutput {

  private static final Log LOG = LogFactory.getLog(
      OnFileSortedOutput.class);

  protected ExternalSorter sorter;
  protected Configuration conf;
  protected MemoryUpdateCallbackHandler
      memoryUpdateCallbackHandler;
  private long startTime;
  private long endTime;
  private boolean sendEmptyPartitionDetails;
  private final AtomicBoolean isStarted = new AtomicBoolean
      (false);

  @Override
  public synchronized List<Event> initialize() throws
      IOException {
    this.startTime = System.nanoTime();
    this.conf = TezUtils.createConfFromUserPayload(
        getContext().getUserPayload());
    // Initializing this parametr in this conf since it is
        used in multiple
    // places (wherever LocalDirAllocator is used) -
        TezTaskOutputFiles,
    // TezMerger, etc.
    this.conf.setStrings(TezRuntimeFrameworkConfigs.
        LOCAL_DIRS, getContext().getWorkDirs());
    this.memoryUpdateCallbackHandler = new
        MemoryUpdateCallbackHandler();
    getContext().requestInitialMemory(
        ExternalSorter.getInitialMemoryRequirement(conf,
            getContext().getTotalMemoryAvailableToTask()),
                memoryUpdateCallbackHandler);

    sendEmptyPartitionDetails = this.conf.getBoolean(
        TezJobConfig.
            TEZ_RUNTIME_EMPTY_PARTITION_INFO_VIA_EVENTS_ENABLED
            ,
        TezJobConfig.
            TEZ_RUNTIME_EMPTY_PARTITION_INFO_VIA_EVENTS_ENABLED_DEFAULT
            );
    return Collections.emptyList();
  }

  @Override
  public synchronized void start() throws Exception {
    if (!isStarted.get()) {
      memoryUpdateCallbackHandler.validateUpdateReceived();
      if (this.conf.getInt(TezJobConfig.
          TEZ_RUNTIME_SORT_THREADS,
          TezJobConfig.TEZ_RUNTIME_SORT_THREADS_DEFAULT) >
              1) {
        sorter = new PipelinedSorter(getContext(), conf,
            getNumPhysicalOutputs(),
            memoryUpdateCallbackHandler.getMemoryAssigned()
                );
      } else {
        sorter = new DefaultSorter(getContext(), conf,
            getNumPhysicalOutputs(),
            memoryUpdateCallbackHandler.getMemoryAssigned()
                );
      }
      isStarted.set(true);
    }
  }

  @Override
  public synchronized KeyValueWriter getWriter() throws
      IOException {
    Preconditions.checkState(isStarted.get(), "Cannot get
        writer before starting the Output");
    return new KeyValueWriter() {
      @Override
      public void write(Object key, Object value) throws
          IOException {
        sorter.write(key, value);
      }
    };
  }

  @Override
  public synchronized void handleEvents(List<Event>
      outputEvents) {
    // Not expecting any events.
  }

  @Override
  public synchronized List<Event> close() throws
      IOException {
    if (sorter != null) {
      sorter.flush();
      sorter.close();
      this.endTime = System.nanoTime();
      return generateEventsOnClose();
    } else {
      LOG.warn("Attempting to close output " + getContext()
          .getDestinationVertexName()
          + " before it was started");
      return Collections.emptyList();
    }
  }

  protected List<Event> generateEventsOnClose() throws
      IOException {
    String host = System.getenv(ApplicationConstants.
        Environment.NM_HOST
        .toString());
    ByteBuffer shuffleMetadata = getContext()
        .getServiceProviderMetaData(ShuffleUtils.
            SHUFFLE_HANDLER_SERVICE_ID);
    int shufflePort = ShuffleUtils.
        deserializeShuffleProviderMetaData(shuffleMetadata
```

```
        );

    DataMovementEventPayloadProto.Builder payloadBuilder =
        DataMovementEventPayloadProto
        .newBuilder();

    if (sendEmptyPartitionDetails) {
      Path indexFile = sorter.getMapOutput().
          getOutputIndexFile();
      TezSpillRecord spillRecord = new TezSpillRecord(
          indexFile, conf);
      BitSet emptyPartitionDetails = new BitSet();
      int emptyPartitions = 0;
      for(int i=0;i<spillRecord.size();i++) {
        TezIndexRecord indexRecord = spillRecord.getIndex(i
            );
        if (!indexRecord.hasData()) {
          emptyPartitionDetails.set(i);
          emptyPartitions++;
        }
      }
      if (emptyPartitions > 0) {
        ByteString emptyPartitionsBytesString =
            TezUtils.compressByteArrayToByteString(TezUtils
                .toByteArray(emptyPartitionDetails));
        payloadBuilder.setEmptyPartitions(
            emptyPartitionsBytesString);
        LOG.info("EmptyPartition bitsetSize=" +
            emptyPartitionDetails.cardinality() + ",
            numOutputs="
                + getNumPhysicalOutputs() + ",
                    emptyPartitions=" + emptyPartitions
            + ", compressedSize=" +
                emptyPartitionsBytesString.size());
      }
    }
    payloadBuilder.setHost(host);
    payloadBuilder.setPort(shufflePort);
    payloadBuilder.setPathComponent(getContext().
        getUniqueIdentifier());
    payloadBuilder.setRunDuration((int) ((endTime -
        startTime) / 1000));
    DataMovementEventPayloadProto payloadProto =
        payloadBuilder.build();
    byte[] payloadBytes = payloadProto.toByteArray();

    long outputSize = getContext().getCounters()
        .findCounter(TaskCounter.OUTPUT_BYTES).getValue();
    VertexManagerEventPayloadProto.Builder vmBuilder =
        VertexManagerEventPayloadProto
        .newBuilder();
    vmBuilder.setOutputSize(outputSize);
    VertexManagerEvent vmEvent = new VertexManagerEvent(
        getContext().getDestinationVertexName(), vmBuilder.
            build().toByteArray());

    List<Event> events = Lists.newArrayListWithCapacity(
        getNumPhysicalOutputs() + 1);
    events.add(vmEvent);

    CompositeDataMovementEvent csdme = new
        CompositeDataMovementEvent(0,
        getNumPhysicalOutputs(), payloadBytes);
    events.add(csdme);

    return events;
  }
}
```

Listing 1: Tez LogicalInput

```
package org.apache.tez.runtime.library.input;

import java.io.IOException;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.atomic.AtomicBoolean;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.RawComparator;
import org.apache.tez.common.TezJobConfig;
import org.apache.tez.common.TezRuntimeFrameworkConfigs;
import org.apache.tez.common.TezUtils;
import org.apache.tez.common.counters.TaskCounter;
import org.apache.tez.common.counters.TezCounter;
import org.apache.tez.runtime.api.AbstractLogicalInput;
import org.apache.tez.runtime.api.Event;
import org.apache.tez.runtime.library.api.KeyValuesReader;
import org.apache.tez.runtime.library.common.ConfigUtils;
import org.apache.tez.runtime.library.common.
    MemoryUpdateCallbackHandler;
import org.apache.tez.runtime.library.common.ValuesIterator
    ;
import org.apache.tez.runtime.library.common.shuffle.impl.
    Shuffle;
import org.apache.tez.runtime.library.common.sort.impl.
    TezRawKeyValueIterator;

import com.google.common.base.Preconditions;


/**
 * <code>ShuffleMergedInput</code> in a {@link
     AbstractLogicalInput} which shuffles
 * intermediate sorted data, merges them and provides key/<
     values> to the
 * consumer.
 *
 * The Copy and Merge will be triggered by the
     initialization - which is handled
 * by the Tez framework. Input is not consumable until the
     Copy and Merge are
 * complete. Methods are provided to check for this, as
     well as to wait for
 * completion. Attempting to get a reader on a non-complete
     input will block.
 *
 */
public class ShuffledMergedInput extends
    AbstractLogicalInput {

  static final Log LOG = LogFactory.getLog(
      ShuffledMergedInput.class);

  protected TezRawKeyValueIterator rawIter = null;
  protected Configuration conf;
  protected Shuffle shuffle;
  protected MemoryUpdateCallbackHandler
      memoryUpdateCallbackHandler;
  private final BlockingQueue<Event> pendingEvents = new
      LinkedBlockingQueue<Event>();
  private long firstEventReceivedTime = -1;
  @SuppressWarnings("rawtypes")
  protected ValuesIterator vIter;

  private TezCounter inputKeyCounter;
  private TezCounter inputValueCounter;

  private final AtomicBoolean isStarted = new AtomicBoolean
      (false);

  @Override
  public synchronized List<Event> initialize() throws
      IOException {
    this.conf = TezUtils.createConfFromUserPayload(
        getContext().getUserPayload());

    if (this.getNumPhysicalInputs() == 0) {
      getContext().requestInitialMemory(0l, null);
      isStarted.set(true);
      getContext().inputIsReady();
      LOG.info("input fetch not required since there are 0
          physical inputs for input vertex: "
          + getContext().getSourceVertexName());
      return Collections.emptyList();
    }

    long initialMemoryRequest = Shuffle.
        getInitialMemoryRequirement(conf,
        getContext().getTotalMemoryAvailableToTask());
    this.memoryUpdateCallbackHandler = new
        MemoryUpdateCallbackHandler();
    getContext().requestInitialMemory(initialMemoryRequest,
        memoryUpdateCallbackHandler);
```

```
    this.inputKeyCounter = getContext().getCounters().
        findCounter(TaskCounter.REDUCE_INPUT_GROUPS);
    this.inputValueCounter = getContext().getCounters().
        findCounter(
        TaskCounter.REDUCE_INPUT_RECORDS);
    this.conf.setStrings(TezRuntimeFrameworkConfigs.
        LOCAL_DIRS, getContext().getWorkDirs());

    return Collections.emptyList();
  }

  @Override
  public synchronized void start() throws IOException {
    if (!isStarted.get()) {
      memoryUpdateCallbackHandler.validateUpdateReceived();
      // Start the shuffle - copy and merge
      shuffle = new Shuffle(getContext(), conf,
          getNumPhysicalInputs(),
          memoryUpdateCallbackHandler.getMemoryAssigned())
          ;
      shuffle.run();
      if (LOG.isDebugEnabled()) {
        LOG.debug("Initialized the handlers in shuffle..
            Safe to start processing..");
      }
      List<Event> pending = new LinkedList<Event>();
      pendingEvents.drainTo(pending);
      if (pending.size() > 0) {
        LOG.info("NoAutoStart delay in processing first
            event: "
            + (System.currentTimeMillis() -
                firstEventReceivedTime));
        shuffle.handleEvents(pending);
      }
      isStarted.set(true);
    }
  }

  /**
   * Check if the input is ready for consumption
   *
   * @return true if the input is ready for consumption, or
       if an error occurred
   *         processing fetching the input. false if the
       shuffle and merge are
   *         still in progress
   * @throws InterruptedException
   * @throws IOException
   */
  public synchronized boolean isInputReady() throws
      IOException, InterruptedException {
    Preconditions.checkState(isStarted.get(), "Must start
        input before invoking this method");
    if (getNumPhysicalInputs() == 0) {
      return true;
    }
    return shuffle.isInputReady();
  }

  /**
   * Waits for the input to become ready for consumption
   * @throws IOException
   * @throws InterruptedException
   */
  public void waitForInputReady() throws IOException,
      InterruptedException {
    // Cannot synchronize entire method since this is
        called form user code and can block.
    Shuffle localShuffleCopy = null;
    synchronized (this) {
      Preconditions.checkState(isStarted.get(), "Must start
          input before invoking this method");
      if (getNumPhysicalInputs() == 0) {
        return;
      }
      localShuffleCopy = shuffle;
    }

    TezRawKeyValueIterator localRawIter = localShuffleCopy.
        waitForInput();
    synchronized(this) {
      rawIter = localRawIter;
      createValuesIterator();
    }
  }


  @Override
  public synchronized List<Event> close() throws
      IOException {
    if (this.getNumPhysicalInputs() != 0 && rawIter != null
        ) {
      rawIter.close();
    }
    if (shuffle != null) {
      shuffle.shutdown();
    }
    return Collections.emptyList();
  }

  /**
   * Get a KVReader for the Input.</p> This method will
       block until the input is
   * ready - i.e. the copy and merge stages are complete.
       Users can use the
   * isInputReady method to check if the input is ready,
       which gives an
   * indication of whether this method will block or not.
   *
   * NOTE: All values for the current K-V pair must be read
       prior to invoking
   * moveToNext. Once moveToNext() is called, the
       valueIterator from the
   * previous K-V pair will throw an Exception
   *
   * @return a KVReader over the sorted input.
   */
  @Override
  public KeyValuesReader getReader() throws IOException {
    // Cannot synchronize entire method since this is
        called form user code and can block.
    TezRawKeyValueIterator rawIterLocal;
    synchronized (this) {
      rawIterLocal = rawIter;
      if (getNumPhysicalInputs() == 0) {
        return new KeyValuesReader() {
          @Override
          public boolean next() throws IOException {
            return false;
          }

          @Override
          public Object getCurrentKey() throws IOException
              {
            throw new RuntimeException("No data available
                in Input");
          }

          @Override
          public Iterable<Object> getCurrentValues() throws
              IOException {
            throw new RuntimeException("No data available
                in Input");
          }
        };
      }
    }
    if (rawIterLocal == null) {
      try {
        waitForInputReady();
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new IOException("Interrupted while waiting
            for input ready", e);
      }
    }
    @SuppressWarnings("rawtypes")
    ValuesIterator valuesIter = null;
    synchronized(this) {
      valuesIter = vIter;
    }
    return new ShuffledMergedKeyValuesReader(valuesIter);
  }

  @Override
  public void handleEvents(List<Event> inputEvents) {
    synchronized (this) {
      if (getNumPhysicalInputs() == 0) {
        throw new RuntimeException("No input events
            expected as numInputs is 0");
      }
```

```
      if (!isStarted.get()) {
        if (firstEventReceivedTime == -1) {
          firstEventReceivedTime = System.currentTimeMillis
              ();
        }
        pendingEvents.addAll(inputEvents);
        return;
      }
    }
    shuffle.handleEvents(inputEvents);
  }

  @SuppressWarnings({ "rawtypes", "unchecked" })
  protected synchronized void createValuesIterator()
      throws IOException {
    // Not used by ReduceProcessor
    vIter = new ValuesIterator(rawIter,
        (RawComparator) ConfigUtils.
            getIntermediateInputKeyComparator(conf),
        ConfigUtils.getIntermediateInputKeyClass(conf),
        ConfigUtils.getIntermediateInputValueClass(conf),
            conf, inputKeyCounter, inputValueCounter);

  }

  @SuppressWarnings("rawtypes")
  public RawComparator getInputKeyComparator() {
    return (RawComparator) ConfigUtils.
        getIntermediateInputKeyComparator(conf);
  }

  @SuppressWarnings("rawtypes")
  private static class ShuffledMergedKeyValuesReader
      implements KeyValuesReader {

    private final ValuesIterator valuesIter;

    ShuffledMergedKeyValuesReader(ValuesIterator valuesIter
        ) {
      this.valuesIter = valuesIter;
    }

    @Override
    public boolean next() throws IOException {
      return valuesIter.moveToNext();
    }

    @Override
    public Object getCurrentKey() throws IOException {
      return valuesIter.getKey();
    }

    @Override
    @SuppressWarnings("unchecked")
    public Iterable<Object> getCurrentValues() throws
        IOException {
      return valuesIter.getValues();
    }
  };
}
```

Listing 2: Tez LogicalOutput

```
package org.apache.tez.runtime.library.input;

import java.io.IOException;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.atomic.AtomicBoolean;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.DefaultCodec;
import org.apache.hadoop.util.ReflectionUtils;
import org.apache.tez.common.TezJobConfig;
import org.apache.tez.common.TezRuntimeFrameworkConfigs;
import org.apache.tez.common.TezUtils;
import org.apache.tez.common.counters.TaskCounter;
import org.apache.tez.common.counters.TezCounter;
import org.apache.tez.runtime.api.AbstractLogicalInput;
import org.apache.tez.runtime.api.Event;
import org.apache.tez.runtime.library.api.KeyValueReader;
import org.apache.tez.runtime.library.common.ConfigUtils;
import org.apache.tez.runtime.library.common.
    MemoryUpdateCallbackHandler;
import org.apache.tez.runtime.library.common.readers.
    ShuffledUnorderedKVReader;
import org.apache.tez.runtime.library.shuffle.common.
    ShuffleEventHandler;
import org.apache.tez.runtime.library.shuffle.common.impl.
    ShuffleInputEventHandlerImpl;
import org.apache.tez.runtime.library.shuffle.common.impl.
    ShuffleManager;
import org.apache.tez.runtime.library.shuffle.common.impl.
    SimpleFetchedInputAllocator;

import com.google.common.base.Preconditions;

public class ShuffledUnorderedKVInput extends
    AbstractLogicalInput {

  private static final Log LOG = LogFactory.getLog(
      ShuffledUnorderedKVInput.class);

  private Configuration conf;
  private ShuffleManager shuffleManager;
  private final BlockingQueue<Event> pendingEvents = new
      LinkedBlockingQueue<Event>();
  private long firstEventReceivedTime = -1;
  private MemoryUpdateCallbackHandler
      memoryUpdateCallbackHandler;
  @SuppressWarnings("rawtypes")
  private ShuffledUnorderedKVReader kvReader;

  private final AtomicBoolean isStarted = new AtomicBoolean
      (false);
  private TezCounter inputRecordCounter;

  private SimpleFetchedInputAllocator inputManager;
  private ShuffleEventHandler inputEventHandler;

  public ShuffledUnorderedKVInput() {
  }

  @Override
  public synchronized List<Event> initialize() throws
      Exception {
    Preconditions.checkArgument(getNumPhysicalInputs() !=
        -1, "Number of Inputs has not been set");
    this.conf = TezUtils.createConfFromUserPayload(
        getContext().getUserPayload());

    if (getNumPhysicalInputs() == 0) {
      getContext().requestInitialMemory(0l, null);
      isStarted.set(true);
      getContext().inputIsReady();
      LOG.info("input fetch not required since there are 0
          physical inputs for input vertex: "
          + getContext().getSourceVertexName());
      return Collections.emptyList();
    } else {
      long initalMemReq = getInitialMemoryReq();
      memoryUpdateCallbackHandler = new
          MemoryUpdateCallbackHandler();
      this.getContext().requestInitialMemory(initalMemReq,
          memoryUpdateCallbackHandler);
    }

    this.conf.setStrings(TezRuntimeFrameworkConfigs.
        LOCAL_DIRS, getContext().getWorkDirs());
    this.inputRecordCounter = getContext().getCounters().
        findCounter(
        TaskCounter.INPUT_RECORDS_PROCESSED);
    return Collections.emptyList();
  }

  @Override
  public synchronized void start() throws IOException {
    if (!isStarted.get()) {
      ////// Initial configuration
      memoryUpdateCallbackHandler.validateUpdateReceived();
      CompressionCodec codec;
      if (ConfigUtils.isIntermediateInputCompressed(conf))
          {
        Class<? extends CompressionCodec> codecClass =
```

```
            ConfigUtils
88          .getIntermediateInputCompressorClass(conf,
                DefaultCodec.class);
89        codec = ReflectionUtils.newInstance(codecClass,
              conf);
90      } else {
91        codec = null;
92      }
93
94      boolean ifileReadAhead = conf.getBoolean(TezJobConfig
            .TEZ_RUNTIME_IFILE_READAHEAD,
95          TezJobConfig.TEZ_RUNTIME_IFILE_READAHEAD_DEFAULT)
                ;
96      int ifileReadAheadLength = 0;
97      int ifileBufferSize = 0;
98
99      if (ifileReadAhead) {
100       ifileReadAheadLength = conf.getInt(TezJobConfig.
              TEZ_RUNTIME_IFILE_READAHEAD_BYTES,
101           TezJobConfig.
                  TEZ_RUNTIME_IFILE_READAHEAD_BYTES_DEFAULT)
                  ;
102     }
103     ifileBufferSize = conf.getInt("io.file.buffer.size",
104         TezJobConfig.
                TEZ_RUNTIME_IFILE_BUFFER_SIZE_DEFAULT);
105
106     this.inputManager = new SimpleFetchedInputAllocator(
            getContext().getUniqueIdentifier(), conf,
107         getContext().getTotalMemoryAvailableToTask(),
108         memoryUpdateCallbackHandler.getMemoryAssigned());
109
110     this.shuffleManager = new ShuffleManager(getContext()
            , conf, getNumPhysicalInputs(), ifileBufferSize,
111         ifileReadAhead, ifileReadAheadLength, codec,
                inputManager);
112
113     this.inputEventHandler = new
            ShuffleInputEventHandlerImpl(getContext(),
            shuffleManager,
114         inputManager, codec, ifileReadAhead,
                ifileReadAheadLength);
115
116     ////// End of Initial configuration
117
118     this.shuffleManager.run();
119     this.kvReader = createReader(inputRecordCounter,
            codec,
120         ifileBufferSize, ifileReadAhead,
                ifileReadAheadLength);
121     List<Event> pending = new LinkedList<Event>();
122     pendingEvents.drainTo(pending);
123     if (pending.size() > 0) {
124       LOG.info("NoAutoStart delay in processing first
              event: "
125           + (System.currentTimeMillis() -
                firstEventReceivedTime));
126       inputEventHandler.handleEvents(pending);
127     }
128     isStarted.set(true);
129   }
130 }
131
132 @Override
133 public synchronized KeyValueReader getReader() throws
      Exception {
134   Preconditions.checkState(isStarted.get(), "Must start
        input before invoking this method");
135   if (getNumPhysicalInputs() == 0) {
136     return new KeyValueReader() {
137       @Override
138       public boolean next() throws IOException {
139         return false;
140       }
141
142       @Override
143       public Object getCurrentKey() throws IOException {
144         throw new RuntimeException("No data available in
              Input");
145       }
146
147       @Override
148       public Object getCurrentValue() throws IOException
            {
149         throw new RuntimeException("No data available in
              Input");
150       }
151     };
152   }
153   return this.kvReader;
154 }
155
156 @Override
157 public void handleEvents(List<Event> inputEvents) throws
      IOException {
158   synchronized (this) {
159     if (getNumPhysicalInputs() == 0) {
160       throw new RuntimeException("No input events
              expected as numInputs is 0");
161     }
162     if (!isStarted.get()) {
163       if (firstEventReceivedTime == -1) {
164         firstEventReceivedTime = System.currentTimeMillis
                ();
165       }
166       // This queue will keep growing if the Processor
            decides never to
167       // start the event. The Input, however has no idea,
            on whether start
168       // will be invoked or not.
169       pendingEvents.addAll(inputEvents);
170       return;
171     }
172   }
173   inputEventHandler.handleEvents(inputEvents);
174 }
175
176 @Override
177 public synchronized List<Event> close() throws Exception
      {
178   if (this.shuffleManager != null) {
179     this.shuffleManager.shutdown();
180   }
181   return null;
182 }
183
184 private long getInitialMemoryReq() {
185   return SimpleFetchedInputAllocator.getInitialMemoryReq(
          conf,
186       getContext().getTotalMemoryAvailableToTask());
187 }
188
189
190 @SuppressWarnings("rawtypes")
191 private ShuffledUnorderedKVReader createReader(TezCounter
        inputRecordCounter, CompressionCodec codec,
192     int ifileBufferSize, boolean ifileReadAheadEnabled,
          int ifileReadAheadLength)
193     throws IOException {
194   return new ShuffledUnorderedKVReader(shuffleManager,
        conf, codec, ifileReadAheadEnabled,
195       ifileReadAheadLength, ifileBufferSize,
            inputRecordCounter);
196 }
197
198 }
```

Listing 3: Tez LogicalOutput