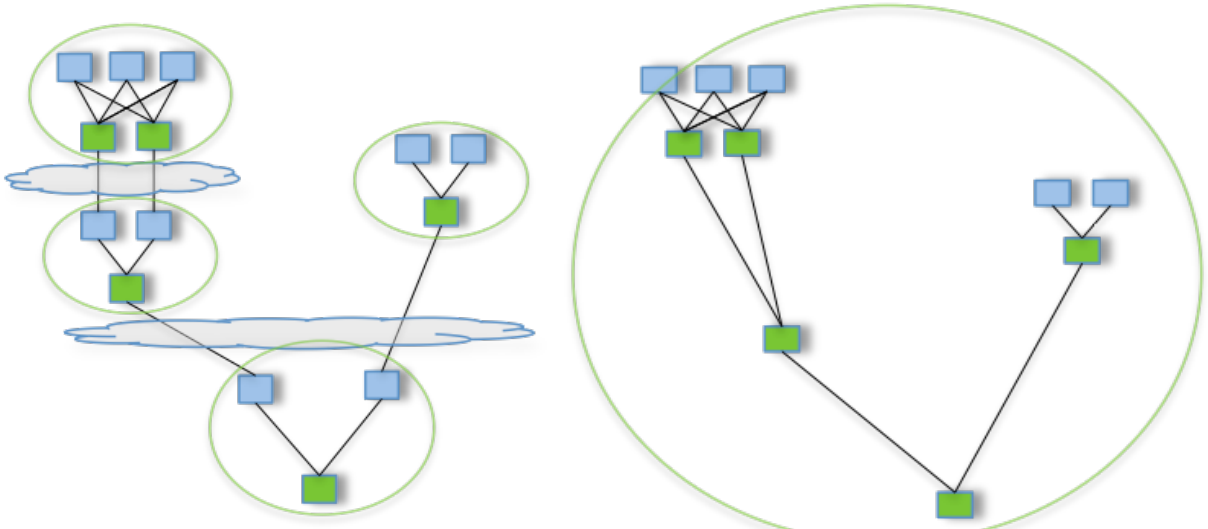
Introduction

The Apache TEZ® project is aimed at building an application framework which allows for a complex directed-acyclic-graph of tasks for processing data. It is currently built atop Apache Hadoop YARN.

The 2 main design themes for Tez are:

- **Empowering end users by:**
  - Expressive dataflow definition APIs
  - Flexible Input-Processor-Output runtime model
  - Data type agnostic
  - Simplifying deployment
- **Execution Performance**
  - Performance gains over Map Reduce
  - Optimal resource management
  - Plan reconfiguration at runtime
  - Dynamic physical data flow decisions

By allowing projects like Apache Hive and Apache Pig to run a complex DAG of tasks, Tez can be used to process data, that earlier took multiple MR jobs, now in a single Tez job as shown below.



Pig/Hive - MR

Pig/Hive - Tez

Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications

Bikas Saha[h], Hitesh Shah[h], Siddharth Seth[h],
Gopal Vijayaraghavan[h], Arun Murthy[h], Carlo Curino[m]
[h]Hortonworks, [m]Microsoft
[h]{bikas, hitesh, sseth, gopal, acm}@hortonworks.com, [m]ccurino@microsoft.com

ABSTRACT

The broad success of Hadoop has led to a fast-evolving and diverse ecosystem of application engines that are building upon the YARN resource management layer. The open-source implementation of MapReduce is being slowly replaced by a collection of engines dedicated to specific verticals. This has led to growing fragmentation and repeated efforts—with each new vertical engine re-implementing fundamental features (e.g. fault-tolerance, security, stragglers mitigation, etc.) from scratch.

In this paper, we introduce Apache Tez, an *open-source framework designed to build data-flow driven processing runtimes*. Tez provides a scaffolding and library components that can be used to quickly build scalable and efficient data-flow centric engines. Central to our design is fostering *component re-use, without hindering customizability* of the performance-critical data plane. This is in fact the key differentiator with respect to the previous generation of systems (e.g. Dryad, MapReduce) and even emerging ones (e.g. Spark), that provided an d mandated a fixed data plane implementation. Furthermore, Tez provides native support to build runtime optimizations, such as dynamic partition pruning for Hive.

Tez is deployed at Yahoo!, Microsoft Azure, LinkedIn and numerous Hortonworks customer sites, and a growing number of engines are being integrated with it. This confirms our intuition that most of the popular vertical engines can leverage a core set of building blocks. We complement qualitative accounts of real-world adoption with quantitative experimental evidence that Tez-based implementations of Hive, Pig, Spark, and Cascading on YARN outperform their original YARN implementation on popular benchmarks (TPC-DS, TPC-H) and production workloads.

1. INTRODUCTION

Large scale data analytics, once an exotic technology leveraged exclusively by large web-companies, is nowadays available and indispensable for most modern organizations. This broader user base has fostered an explosion of interest in this area, and led to a flourishing BigData industry. In this paper, we use the lens of the Hadoop ecosystem to describe industry-wide trends, as this provides the ideal context for introducing our system: Apache Tez. We postpone to Section 8 a broader comparison with the related projects like Dryad, Nephele, Hyracks[24, 14, 15] etc., which undeniably served as an inspiration and sometimes the blueprint for the design of Tez.

Hadoop, which was initially designed as a single-purpose system (to run MapReduce jobs to build a web index), has evolved into a catch-all data analytics platform. The first phase of this journey consisted of several efforts proposing *higher level abstractions* atop MapReduce[21], examples of which are Hive [30], Pig [28], and Cascading [4]. This sped-up the adoption of Hadoop, but led to inefficiencies and poor performance [29]. These limitations and the pressure towards more flexibility and efficiency led to the refactoring of Hadoop into a general purpose, OS-like resource management layer, namely YARN [32], and an application framework layer allowing for arbitrary execution engines. This enabled different applications to share a cluster, and made MapReduce *just another application* in the Hadoop ecosystem. Important examples of applications that break-free of the MapReduce model (and runtime) are Spark [38], Impala [25] and Flink [5]. This has accelerated innovation, but also led to a less efficient ecosystem, where common functionalities were being replicated across frameworks. For example, MapReduce and Spark independently developed mechanisms to implement delay scheduling[36].

In this paper, we introduce Tez, a project that embraces the architectural shift to YARN, and pushes it further, by proposing a reusable, flexible and extensible scaffolding that can support arbitrary data-flow oriented frameworks, while avoiding replicated functionalities. Tez APIs allow frameworks to clearly model the logical and physical semantics of their data flow graphs, with minimal code. It is important to clarify that Tez is a library to build data-flow based runtimes/engines and not an engine by itself—for example, the Hive runtime engine for Hadoop has been rewritten in version 0.13 to use Tez libraries.

Tez makes the following key contributions:

1. Allows users to model computation as a DAG (Directed-Acyclic-Graph) — akin to Dryad/Nephele/Hyracks. The novelty lies in a finer grained decomposition of the classical notions of vertex and edge, that delivers greater control and customization of the data plane.

2. Exposes APIs to dynamically evolve the (finer grained) DAG definition. This enables sophisticated runtime query optimizations, such as pruning data partitions, based on online information.

3. Provides a scalable and efficient implementation of state-of-the-art features, e.g., YARN-compatible security, data-locality awareness, resource-reuse, fault-tolerance and speculation.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2742790

4. Provides the opportunity for framework writers and researchers to innovate quickly and create real-world impact by providing experimentation support via pluggable APIs, and an open-source community to learn about the project and contribute back to it.

What sets Tez aside from many alternative proposals of 'unification frameworks' is: 1) proven flexibility and dynamic adaptation, 2) attention to operational concerns (production readiness), and 3) a community-driven effort to embed Tez in multiple existing domain-specific engines.

This is proven by the Tez support of MapReduce, Hive, Pig, Spark, Flink, Cascading, and Scalding, and its adoption in production data-processing clusters at Yahoo, Microsoft Azure, LinkedIn as well as several other organizations using the Hortonworks Data Platform. Beyond discussing the broad practical adoption of Tez, we demonstrate its competence to support Hive, Pig, and Spark, by running standard benchmarks such as TPC-H and TPC-DS, and production workloads from Yahoo!.

The rest of this paper is organized as follows: Section 2 provides some more historical context, and rationale for the design of Tez, while Section 3 introduces the architecture of Tez. Section 4 discusses the implementation of Tez, and highlights pragmatic considerations on efficiency and production-readiness. Section 5 and 6 are devoted to prove its practical relevance, by presenting real-world applications, and a broad experimental evaluation. We conclude by discussing future and related work in Sections 7 and 8, and conclude in Section 9

2. BACKGROUND AND RATIONALE

To understand the motivation and rationale behind Tez, we must first start by providing some background on terminology, and a historical context of distributed computation in Hadoop. The reader not interested in this historical and motivational perspective is invited to continue to Section 3, where we dive into the technical aspects of the Tez architecture.

Terminology. We have used graph terminology so far, appealing to the reader's intuitions. We now introduce our terminology more precisely:

DAG: Directed Acyclic Graph representing the structure of a data processing workflow. Data flows in the direction of the edges.

Vertex: Represents a logical step of processing. A processing step transforms data by applying application-supplied code to filter, or modify the data.

Logical DAG: A logical DAG is comprised of a set of vertices, where each vertex represents a specific step of the computation.

Task: Represents a unit of work in a vertex. In distributed processing, the logical work represented by a single vertex is physically executed as a set of tasks running on potentially multiple machines of the cluster. Each task is an instantiation of the vertex, that processes a subset (or partition) of the input data for that vertex.

Physical DAG: A physical DAG comprises of the set of tasks that are produced by expanding the vertices of a logical DAG into their constituent tasks.

Edge: Represents movement of data between producers and consumers. An edge between vertices of a logical DAG represents the logical data dependency between them. An edge between tasks in the physical DAG represents data transfers between the tasks.

This applies to problems in which different steps can be partitioned into smaller pieces that can be processed in parallel. Typically, the partitioning aligns with distributed shards of data and tries to co-locate processing with its data, thus reducing the cost of computation [21].

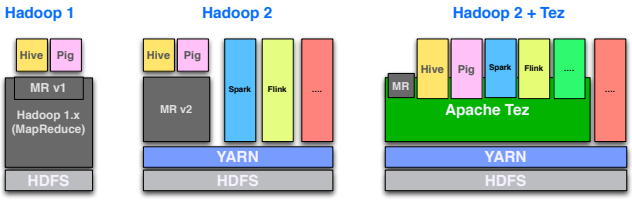


Figure 1: Evolution of Hadoop

Hadoop 1 and Hadoop 2 (YARN). Hadoop started off as a single monolithic software stack where MapReduce was the only execution engine [32]. All manners of data processing had to translate their logic into a single MapReduce job or a series of MapReduce jobs. MapReduce was also responsible for cluster resource management and resource allocation. Hadoop 2 is the current generation of Hadoop which separates these responsibilities by creating a general purpose resource management layer named YARN [32]. This de-couples applications from the core Hadoop platform and allows multiple application types to execute in a Hadoop cluster in addition to MapReduce. There are many domain-specific applications like Apache Hive for SQL-like data processing, Apache Pig for ETL scripting or Cascading for writing data processing applications in Java, which were earlier restricted to relying on MapReduce to execute their custom logic. These applications can now have a more customized implementation of their logic by running natively on YARN.

While specialization can deliver performance advantages, there is a substantial opportunity to create a common set of building blocks that can be used by these applications for their customized implementation on YARN. We try to seize that opportunity with Tez, as discussed next. An analysis of popular Hadoop ecosystem applications like Apache Hive, Pig, Spark etc. suggests that there are shared features that all of them need. These include negotiating resources from YARN to run application's tasks, handling security within the clusters, recovering from hardware failures, publishing metrics and statistics etc. A lot of this is highly specialized, hard to develop infrastructure that everyone has to replicate when building from scratch. A common implementation makes it easier to write applications because it removes that burden from the application writers and lets them focus on the unique logic of their application.

Henceforth, unless otherwise specified, when we mention Hadoop we imply the Hadoop 2 compute stack with YARN as the underlying resource allocation layer; and by Hadoop ecosystem we imply the compute ecosystem consisting of open source and commercial projects running on YARN such as Apache Hive, Pig etc.

An effort to provide these common features requires the creation of a framework to express and model these workloads optimally. Then this model can be applied and executed on the YARN application framework via a shared substrate library. This rationalizes the following requirements for such a shared library, which we have high-lighted by comparisons with MapReduce — a general purpose engine that has been forced to act as shared substrate until now.

Expressiveness. MapReduce has a simple modeling API for describing the computation by requiring all application algorithms to be translated into *map* and *reduce* functions. As observed by others before in [24, 14, 15], this is too constraining, and a DAG-oriented model can more naturally capture a broader set of computations. Thus we define Tez's central model around DAGs of execution as well. Moreover, MapReduce also provides built-in se-

mantics to the logic running in map/reduce steps and imposed a *sorted & partitioned* movement of data between map and reduce steps [21]. These built-in semantics, ideal in some core use cases, could be pure overhead in many other scenarios and even undesirable in some. The observation here is the need for an API to describe the structure of arbitrary DAGs without adding unrelated semantics to that DAG structure.

Data-plane Customizability. Once the structure of distributed computation has been defined, there can be a variety of alternative implementations of the actual logic that executes in that structure. These could be algorithmic, e.g. different ways of partitioning the data or these could be related to using different hardware, e.g. using remote memory access (RDMA) where available. In the context of MapReduce, the built-in semantics of the engine makes such customizations difficult because they intrude in the implementation of the engine itself. Secondly, the monolithic structure of the tasks executing the MapReduce job on the cluster makes plugging in alternative implementations difficult. This motivates that data transformations and data movements that define the data plane need to be completely customizable. There is a need to be able to model different aspects of task execution in a manner that allows individual aspects of the execution, e.g. reading input, processing data etc. to be customized easily. Interviewing several members of the Hadoop community we confirmed that evolving existing engines (e.g., changing the shuffle behavior in MapReduce) is far from trivial.

While other frameworks such as [24, 15, 38], already support a more general notion of DAGs, they share the same limitation of MapReduce, built-in semantics and implementations of the data-plane. With Tez we provide a lower level abstraction, that enables such semantics and specialized implementations to be added on top of a basic shared scaffolding.

Late-binding Runtime Optimizations. Applications need to make late-binding decisions on their data processing logic for performance [13]. The algorithm, e.g. join strategies and scan mechanisms, could change based on dynamically observing data being read. Partition cardinality and work division could change as the application gets a better understanding of its data and environment. Hadoop clusters can be very dynamic in their usage and load characteristics. Users and jobs enter and exit the cluster continuously and have varying resource utilization. This makes it important for an application to determine its execution characteristics based on the current state of the cluster. We designed Tez to make this late-binding and on-line decision-making easier to implement, by enabling updates to key abstractions at runtime.

This concludes our overview of historical context and rationale for building Tez. We now turn to describing the high level architecture of Tez, and provide some insight into the key building blocks.

3. ARCHITECTURE

Apache Tez is designed and implemented with a focus on the issues discussed above, in summary: 1) expressiveness of the underlying model, 2) customizability of the data plane, and 3) facilitate runtime optimizations. Instead of building a general purpose execution engine, we realize the need for Tez to provide a *unifying framework for creating purpose-built engines that customize data processing for their specific needs*. Tez solves the common, yet hard problem of orchestrating and running a distributed data processing application on Hadoop and enables the application to focus on providing specific semantics and optimizations. There is a clear separation of concerns between the application layer and the Tez library layer. Apache Tez provides cluster resource negotiation, fault tolerance, resource elasticity, security, built-in performance optimizations and a shared library of ready to use components. The application provides custom application logic, custom data plane and specialized optimizations.

This leads to three key benefits: 1) amortized development costs (Hive and Pig completely rewrote their engines using the Tez libraries in about 6 months), 2) improved performance (we show in Section 6 up to 10× performance improvement while using Tez), and 3) enabling future pipelines that leverage multiple engines, to be run more efficiently because of a shared substrate.

Tez is composed of a set of core APIs that define the data processing and an orchestration framework to launch that on the cluster. Applications are expected to implement these APIs to provide the execution context to the orchestration framework. Its useful to think of Tez as a library to create a scaffolding representing the structure of the data flow, into which the application injects its custom logic (say operators) and data transfer code (say reading from remote machine disks). This design is both tactical and strategic. Long-term, this makes Tez remain application agnostic while in the short term, allows existing applications like Hive or Pig to leverage Tez without significant changes in their core operator pipelines. We will begin with describing the DAG API and Runtime API. These are the primary application facing interfaces used to describe the DAG structure of the application and the code to be executed at runtime. Next we explain support for applying runtime optimizations to the DAG via an event based control plane using VertexManagers and DataSourceInitializers. Finally, in Section 4 we describe the YARN based orchestration framework to execute the all of this on a Hadoop cluster. In particular, we will focus on the performance and production-readiness aspects of the implementation.

3.1 DAG API

The Tez DAG API is exposed to runtime engine builders as an expressive way to capture the structure of their computation in a concise way. The class of data processing application we focus on, are naturally represented as DAGs, where data proceeds from data sources towards data sinks, while being transformed in intermediate vertices. Tez focuses on acyclic graphs, and by assuming deterministic computation on the vertex and data routing on the edges, we enable re-execution based fault tolerance, akin to [24] and is further explained in Section 4.3. Modeling computations as a DAG is not new but hitherto most systems have typically designed DAG APIs in the context of supporting a higher level engine. Tez is designed to model this data flow graph as the main focus. Using well-known concepts of vertices and edges the DAG API enables a clear and concise description of the structure of the computation.

Vertex. A vertex in the DAG API represents transformation of data and is one of the steps in processing the data. This is where the core application logic gets applied to the data. Hence a vertex must be configured with a user-provided *processor* class that defines the logic to be executed in each task. One 'vertex' in the DAG is often executed in parallel across a (possibly massive) number of parallel tasks. The definition of a vertex controls such parallelism. Parallelism is usually determined by the need to process data that is distributed across machines or by the need to divide a large operation into smaller pieces. The task parallelism of a vertex may be defined statically during DAG definition but is typically determined dynamically at runtime.

Edge. An edge in the graph represents the logical and physical aspects of data movement between producer and consumer vertices.

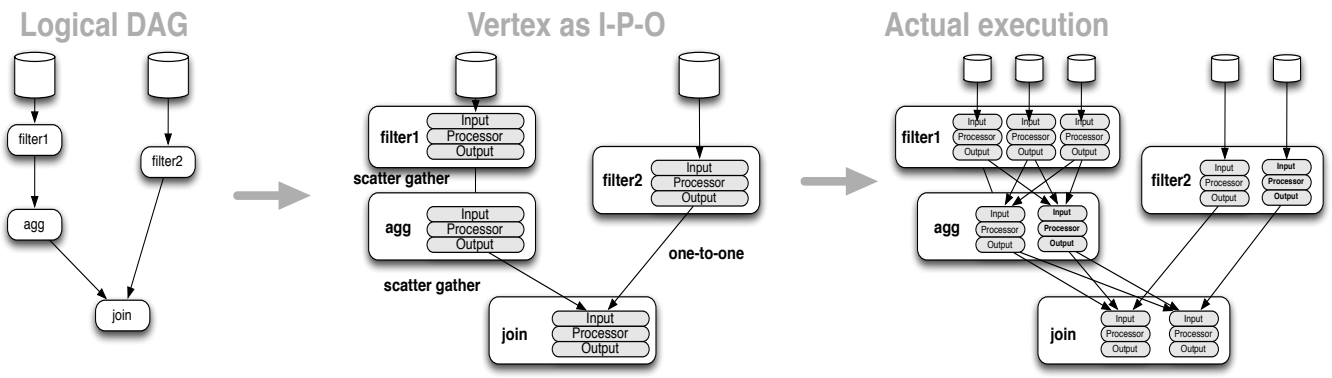


Figure 2: Expansion of the logical vertex DAG to a physical task DAG based on vertex parallelism and edge definition

- *Connection Pattern:* The logical aspect of an edge is the connection pattern between producer and consumer vertex tasks and their scheduling dependency. This enables the orchestration framework to route data from the output of the producer task to the correct input of the consumer task. This routing table must be specified by implementing a pluggable EdgeManagerPlugin API. Figure 3 shows 3 common connection patterns (one-to-one, broadcast, scatter-gather), that can be used to express most DAG connections and come built-in with the project. For cases where custom routing is needed, applications are allowed to define their own routing by providing their own implementation (we give a concrete example in Section 5.2).
- *Transport Mechanism:* The physical aspect of an edge is the storage or transport mechanism employed to move the data. This could be local-disk, or local/remote main-memory, etc. The actual data transfer operation of the edge is performed by a compatible pair of input and output classes that are specified for the edge. Compatibility is based on using the same data format and physical transport mechanisms. E.g. both operate on key-value pairs and operate on disks, or both operate on byte streams and use main memory. Tez comes with built-in inputs and outputs for common use cases as described in Section 4.1

Vertex parallelism and the edge properties can be used by Tez to expand the logical DAG to the real physical task execution DAG during execution as shown in Figure 2

Data Sources and Sinks. The DAG can be defined by creating vertices and connecting them via edges using the DAG API. Typically, the data flow will read initial input from some data sources and write final output to some data sinks. Data sources may be associated with a DataSourceInitializer that can be invoked at runtime to determine the optimal reading pattern for the initial input. E.g. in MapReduce parlance, this corresponds to 'split' calculation [27] where a split is a shard of distributed data that is read by a map task. The initial split calculation for map tasks can be performed using an initializer that considers the data distribution, data locality and available compute capacity to determine the number of splits and the optimal size of each split. Similarly, data sinks may be associated with a DataSinkCommitter that is invoked at runtime to commit the final output. The definition of commit may vary with the output type but is guaranteed to be done once, and typically involves making the output visible to external observers after successful completion.

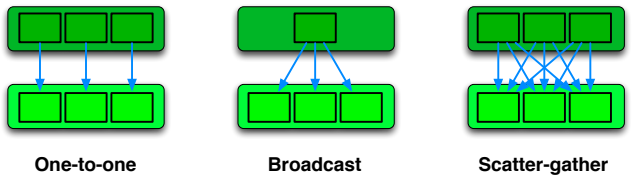


Figure 3: Edge properties: define movement of data between producers and consumers

```
1: DAG dag = DAG.create("WordCount");

2: Vertex tokenizerVertex = Vertex.create("Tokenizer", TokenProcessor.class)
                              .addDataSource("Input", HdfsInitializer.class);

3: Vertex summationVertex = Vertex.create("Summation", SumProcessor.class)
                              .addDataSink("Output", HdfsCommitter.class);

4: EdgeProperty edgeProperty = EdgeProperty.create(scatter_gather,
                              KeyValueShuffleWriter.class, KeyValueShuffleReader.class);

5: dag.addVertex(tokenizerVertex).addVertex(summationVertex)
       .addEdge(Edge.create(tokenizerVertex, summationVertex, edgeProperty);
```

Figure 4: Essence of the Tez API shown via pseudo-code for the canonical WordCount example

This manner of DAG assembly allows for pluggable and re-usable components. A common shared library of inputs and outputs can be re-used by different applications, thus only needing to supply the processor logic in a vertex. Conversely, the same DAG structure may be executed more optimally in a different hardware environment by replacing the inputs/outputs on the edges. Tez comes with an input/output library for data services built into Hadoop - HDFS and the YARN Shuffle Service. This enables Hadoop eco-system applications like Hive and Pig to quickly leverage Tez by implementing only their custom processors. Figure 4 shows a condensed view of describing a logical DAG using the API.

3.2 Runtime API

The DAG API defines the scaffolding structure of the data processing. The Runtime API is used to inject the actual application code that fills the scaffolding. Concretely, the Runtime API defines the interfaces to be implemented to create processor, input and output classes that are specified in the DAG above.

Inputs, Processor, Outputs. A vertex is a logical representation of a transformation step in the DAG. The actual transformations are applied by running tasks, for that vertex, on machines in the cluster. Tez defines each task as a composition of a set of inputs, a processor and a set of outputs (IPO). The processor is defined by the vertex for that task. The inputs are defined by the output classes of the incoming edges to that vertex. The outputs by the input classes of the outgoing edges from that vertex. This enables the processor to have a logical view of the processing, thus retaining the simplified programming model popularized in MapReduce. The inputs and outputs hide details like the data transport, partitioning of data and/or aggregation of distributed shards. The Runtime API is a thin wrapper to instantiate and interact with inputs, processors and outputs. After the IPO objects have been created, they are configured.

IPO Configuration. The framework configures IPOs via an opaque binary payload specified during DAG creation. This manner of binary payload configuration is a common theme to configure any application specific entity in Tez. This allows applications to instantiate their code using any mechanism of their choice. Not only can this be used for simple configuration but also for code injection (as exemplified in Section 5.4). After configuration, the processor is presented with all its inputs and outputs and asked to run. Thereafter, it's up to the processor, inputs and outputs to cooperate with each other to complete the task. The framework interacts with them via a context object to send and receive events about completion, update progress, report errors etc.

Data Plane Agnostic. Tez specifies no data format and in fact, is not part of the data plane during DAG execution. The actual data transfer is performed by the inputs and outputs with Tez only routing connection information between producers and consumers. When a producer task output generates data then it can send metadata about it, say its access URL and size, via Tez, to the consumer task input. Tez routes this metadata using the connection pattern encoded in the edge connecting the producer and consumer. Thus Tez adds minimal overhead on the data plane. This also makes Tez data format agnostic. The inputs, processor and outputs can choose their own data formats (e.g. bytes, records or key-value pairs etc.) as suited for the application, exemplified in Section 5.5

This novel IPO based approach to task composition allows for separation of concerns and makes the system pluggable. The same DAG structure can be instantiated with environment dependent IOs. E.g. different cloud environments can plug in IOs that are optimized for their storage subsystems. We will see in the next sections, how the IPOs can be dynamically configured during execution for even further runtime customizations.

3.3 Event Based Control Plane

The open architecture of the Tez orchestration framework requires a de-coupled control plane that allows a variety of entities to communicate control information with each other. In order to achieve this Tez has an event based control plane that is also exposed to the application. Software components generate events that get routed to receivers. By design, this is an asynchronous, non-blocking, push-based method of communication. Events are used for all communications, be it framework to framework, application to framework and vice versa, or application to application. As shown in Figure 5, a DataEvent is generated by the output of a producer task and contains output metadata (say a URL) for the consumer task to read the data. This event is received by the framework and routed to the input of the consumer task by utilizing the connection information specified by the edge. If a task input has

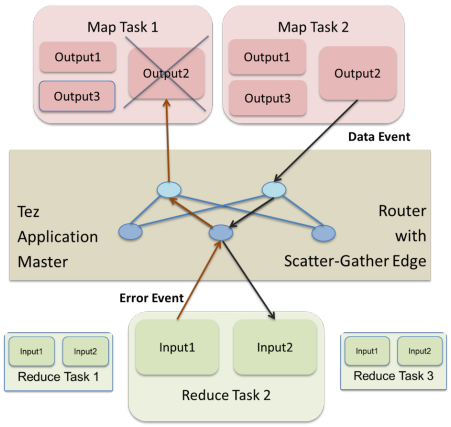


Figure 5: Events used to route metadata between application IOs and error notifications from applications to the framework

an error while reading its data then it can send an ErrorEvent to the framework. Based on such error events, Tez could re-execute the producer task to re-generate the data. Other events could be used to send statistics, progress etc. Event based communication also provides the flexibility to add more entities and communication channels without changing the interaction model or APIs. Tez only routes the events. Each event has an opaque binary payload that is interpreted by the sender and receiver to exchange control metadata. Events flow to and from tasks to the orchestrator on every task heartbeat. Event transfer latency depends on the heartbeat latency and processing latency at the orchestrator. These latencies increase in proportion to the size of the job as they depend on the number of concurrent connections and event load supported by the orchestrator. If control plane events lie on the data plane critical path then they would negatively affect application latency but if they are used only for data plane setup then Tez would not introduce any additional latency on the data plane for low latency applications.

3.4 Vertex Manager: dynamically adapting the execution

As motivated earlier, data processing clusters have variability in compute capacity or data distribution (where data is stored on physical nodes) that applications may consider to plan their work. Data dependent actions like sample based range partitioning or optimizations like partition pruning need the ability to change the DAG on the fly. It is not possible to encode all such current and future graph re-configurations statically, nor can this be done by Tez itself (as it requires too much domain knowledge). Thus Tez needs to allow the application to make such decisions at runtime and coordinate with Tez to dynamically adapt the DAG and its execution. This is enabled via the VertexManager abstraction, similar to [24].

Runtime Graph Re-configuration. When constructing the DAG, each vertex can be associated with a VertexManager. The VertexManager is responsible for vertex re-configuration during DAG execution. The orchestration framework contains various state machines that control the life-cycle of vertices, tasks etc. and the vertex state machine is designed to interact with the VertexManager during state transitions. The VertexManager is provided a context object that notifies it about state changes like task completions etc. Using the context object, the VertexManager can make changes to its own vertex's state. Among other things, the VertexManager can

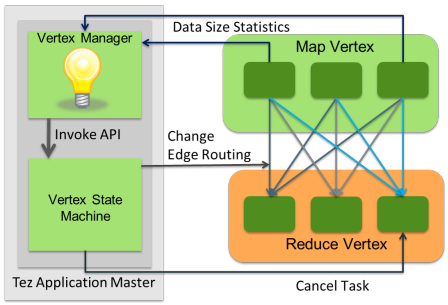


Figure 6: DAG reconfiguration by VertexManager to apply partition cardinality estimated at runtime

control the vertex parallelism, the configuration payloads of the inputs, processors and outputs, the edge properties and scheduling of tasks. As with other entities there is a VertexManager API that can be implemented by applications to customize the vertex execution. Using the same API, Tez comes with some built-in VertexManagers. If a VertexManager is not specified in the DAG, then Tez will pick one of these built-in implementations based on the vertex characteristics.

Automatic Partition Cardinality Estimation. As an example of a runtime optimization, we present a solution to a well-known problem in MapReduce about determining the correct number of tasks in the reduce phase. This number typically depends on the size of the data being shuffled from the mappers to the reducers and is accurately available only at runtime. Shuffle is the term used to describe the cross-network read and aggregation of partitioned input done prior to invoking the reduce operation. In Tez, the ShuffleVertexManager can be used to control the vertices that are reading shuffled data. The tasks producing the data to be shuffled, send data statistics to the ShuffleVertexManager using VertexManager events. As visualized in Figure 6, the ShuffleVertexManager gathers these statistics to calculate the total data size and estimate the correct number of reducers to read that data using a per-reducer desired data size heuristic. Since the number of reducers essentially represents the partition cardinality, this solution can be generalized to estimating the optimal number of partitions at runtime (e.g. partitions participating in a distributed join operation).

Scheduling Optimizations. VertexManagers also control the scheduling of tasks in their vertex. Typically, tasks should be started after their input data is ready. However, if the tasks can proceed meaningfully with partial input then they could be started out of order and use any free compute capacity. The shuffle operation mentioned above is an example of a case where partial inputs can be read by tasks pro-actively. This is an expensive data transfer across the network and starting early can help hide its latency by overlapping it with the completion of tasks that will produce the remaining input. Out of order scheduling can result in scheduling deadlocks in a resource constrained cluster where an out of order task ends up blocking one of its input tasks because it has occupied resources in the cluster. Tez has built-in deadlock detection and preemption to take care of such situations. It will use the DAG dependency to detect tasks running out of order and preempt them to resolve the deadlock.

3.5 Data Source Initializer

In Tez we have modeled data sources as first class entities in our design. The first step in a DAG usually involves reading initial input from data sources like distributed file systems, and typically is the largest in terms of resource consumption. Hence, a good or bad decision at this step can significantly improve or degrade performance. A data source in a DAG can be associated with a DataSourceInitializer that is invoked by the framework before running tasks for the vertex reading that data source. The initializer has the opportunity to use accurate information available at runtime to determine how to optimally read the input. Like the VertexManager, the initializer can also send and receive events from other entities. It also has access to cluster information via its framework context object. Based on these and other sources of information, the initializer can configure the task inputs or notify the vertex manager about vertex re-configurations (E.g. the optimal parallelism needed to process the input).

As an example, we will present a Hive dynamic partition pruning use case. It often happens that a data source will be read and subsequently joined on some key. If the join key space is known then we could only read a subset of the data that is relevant to the join. Sometimes this metadata is only available at runtime after inspecting the data in a different sub-graph of the DAG. Hive uses InputInitializer events to send this metadata from tasks in the other vertices to the initializer of the data source. The initializer uses that metadata to decide the relevant subset of data to read. This can lead to large performance gains depending on the join selectivity.

The above discussion has been a broad overview of the architecture and features in Tez. More details about the semantics around the API's and user defined entities is available in the API documentation on the project website [2]

4. IMPLEMENTATION AND PRACTICAL CONSIDERATIONS

We now turn to describing how the architecture of the previous section is instantiated in YARN, and discuss in more details efficiency and production-readiness aspects of Tez. From an engineering perspective this is where much of our effort was devoted, and what makes Tez a useful building block for data-processing engines.

4.1 Implementation in YARN

The Apache Tez project consists of 3 main parts:

- API library: This provides the DAG and Runtime APIs and other client side libraries to build applications
- Orchestration framework: This has been implemented as a YARN Application Master [32] (hereafter referred to as AM) to execute the DAG in a Hadoop cluster via YARN
- Runtime library: This provides implementations of various inputs and outputs that can be used out of the box.

Typical Tez Application Lifecycle. A Tez based application is written using the API library by constructing the DAG representing the application logic. Typically, higher level applications like Apache Pig construct DAGs on the fly by encoding their native language constructs into Tez DAGs. Since Tez is designed to operate in Hadoop clusters we have provided implementations of inputs and outputs to standard storage services present in all Hadoop clusters - HDFS [16] for reliable data storage and YARN Shuffle Service

[32] for temporary data storage. Applications that use only these services, need to implement just their processors to get up and running. Typically applications create a generic processor host that can be configured to execute DAG dependent operators. Tez inputs and outputs are based on the key-value data format for ease of use within the key-value dominated Hadoop ecosystem of projects like Apache Hive, Pig etc., and can be extended to other data formats. The DAG is then submitted to a YARN cluster using the Tez client library. YARN launches the Tez Application Master (AM - a per-application controller) to orchestrate the DAG execution. The DAG executed by the AM is typically a logical DAG that describes the data flow graph. The AM expands this graph to incorporate task parallelism per vertex. It does this using the input initializers and vertex managers specified in the DAG. The AM then requests YARN for resources to run the tasks from different vertices. YARN responds in the form of containers. A container is a unit of resource allocation on a cluster node. The AM launches tasks on these containers and routes control events. Tasks are typically executed in their dependency order and the DAG completes when all its tasks complete. The AM logs tracing and metadata information for monitoring and debugging.

By leveraging existing libraries and services from YARN and MapReduce, we have been able to quickly build on top of several man-years of production ready code for security and high volume network data shuffling; and integrate with the proven resource sharing and multi-tenancy model in YARN. Thus, applications built using Tez will benefit from all these without expending further effort.

4.2 Execution Efficiency

The YARN implementation of the orchestration framework is built with execution efficiency and performance in mind and incorporates well known ideas learned over the years in various distributed data processing systems.

Locality Aware Scheduling. Scheduling processing close to the data location is important for large-scale data-processing [21, 32]. Tez tries to run tasks close to their input data location. Location may be specified statically during DAG creation but is typically determined at runtime. The tasks that read from initial input data sources typically get locality information from their data sources while intermediate task locality is inferred from their source tasks and edge connections. E.g. tasks with scatter-gather inputs have no specific locality but may prefer to run close to the larger input data shards. 1-1 edges specify strict locality relationships between their source and destination tasks. Since getting perfect locality may not be guaranteed in a busy cluster, the framework automatically relaxes locality from node to rack and so on with delay scheduling [36] used to add a wait period before each relaxation.

Speculation. Large clusters can have heterogeneous hardware and varying loads and hardware-aging. This can lead to environment induced task slowdowns. Such slow tasks are termed stragglers and launching a clone of such tasks is typically used to mitigate their effects on latency [21]. Tez monitors task progress and tries to detect straggler tasks that may be running much slower than other tasks in the same vertex. Upon detecting such a task, a speculative attempt may be launched that runs in parallel with the original task and races it to completion. If the speculative attempt finishes first then it is successful in improving the completion time.

Container Reuse. Recall that the Tez AM runs tasks in containers allocated to it by YARN. When a task completes, the AM has an option to return the container to YARN and ask for another container with different capabilities or locality. However, each container allocation cycle has overheads associated with resource negotiation from YARN as well as launching the container process. This overhead can be minimized by re-using the container to run other pending tasks that match the resource allocation and locality of that container. When there are no such matching tasks, the Tez AM releases the idle containers back to YARN in return for new resources with different capabilities. In the Java world, this reuse has the additional benefit of giving the JVM optimizer a longer time to observe and optimize the hot code paths leading to further performance benefits [1].

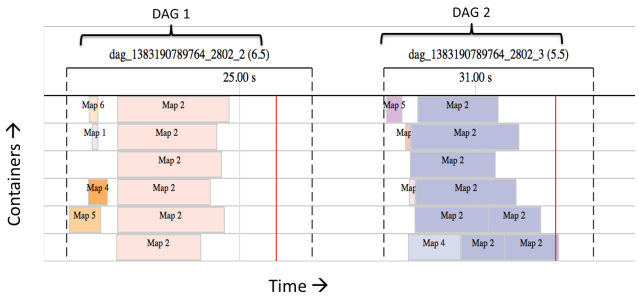


Figure 7: Execution trace of 2 DAGs executed in the same Tez Session. Containers are re-used by tasks within a DAG and across DAGs. Cross DAG re-use happens only in session mode.

Session. A session takes the concept of container reuse one step further. A Tez AM can be run in session mode in which it can run a sequence of DAGs submitted to it by the client. This allows tasks from multiple DAGs to reuse containers and leads to further efficiencies and performance gains. In addition, a session can be pre-warmed by requesting the AM to launch containers before the first DAG is ready to execute. These pre-warmed containers can execute some pre-determined code to allow JVM optimizations to kick in. This extends the benefits of container reuse to the first DAG that gets submitted to the session. E.g. Apache Hive and Pig use the session mode to run multiple drill-down queries in the same session for performance benefits. Tez sessions also enable iterative processing to be performed efficiently. Each iteration can be represented as a new DAG and submitted to a shared session for efficient execution using pre-warmed session resources. Figure 7 shows the trace of a Tez session with containers shared across tasks of multiple DAGs.

Shared Object Registry. Tez extends the benefits of container reuse to the application by providing an in-memory cache of objects that can be populated by a task and then re-used by subsequent tasks. The lifecycle of objects in the cache can be limited to a vertex, a DAG or the session and is managed by the framework. It can be used to avoid re-computing results when possible. E.g. Apache Hive populates the hash table for the smaller side of a map join in Hive parlance (broadcast join). Once a hash table has been constructed by a join task, other join tasks don't need to re-compute it and improve their performance.

4.3 Production Readiness

While performance and efficiency are important for a framework such as Tez, we cannot ignore the standard abilities that are prerequisites for a production ready and dependable framework for large scale data processing. A plethora of entities use technologies like Apache Hive, Pig and other commercial software like Cascading to run mission critical operations. If they are to confidently

build using Apache Tez, then items like fault tolerance, security and multi-tenancy become necessary requirements. Fortunately, Tez has been able to build on top of proven and tested platforms like Hadoop YARN and MapReduce and draws from their strengths for achieving some of these abilities. The YARN integration exemplifies the specialized code implemented in Tez that can be leveraged by higher-level engines using Tez.

Multi-Tenancy. Data processing clusters are becoming increasingly large and sharing their capital expenditure among multiple applications and users is essential from a capex point of view [32]. Applications must be written with such sharing and cooperative behavior in mind. The discrete task based processing model in Tez lends itself nicely to such cooperative behavior. Short lived ephemeral tasks allow resources to be periodically released by Tez applications so that they can be allocated to other users and applications as deemed appropriate by the cluster resource allocation policy. This also enables higher resource utilization by transferring resources from applications that don't need them to applications that do.

This is where engines that effectively deploy services daemons suffer from a drawback. Typically, the service daemons have to pre-allocate a large share of resources that cannot be shared with other applications. For better utilization, these daemons try to run multiple 'tasks' concurrently but that is not useful when there isn't enough load on the system, besides introducing the possibility of interference between concurrent tasks. With Tez, since each task runs in its own container process, the resource allocations are much finer grained. This improves utilization (by reducing allocated resources that are idle) and also provides process-based resource isolation (for CPU/memory etc.). This also provides resource elasticity to Tez applications in that they can scale up to utilize as much resources as the cluster can spare to speed up job execution time while gracefully degrading performance but still completing the job when resources are scarce. To be clear, this discussion about daemon based designs is in the specific context of ephemeral data processing jobs. There are many contexts like data storage, web services, PAAS applications etc. where a long running shared service provided by a daemon based engine is suitable.

Security. Security is real concern with the variety and volume of data stored in modern data processing clusters. Hadoop has built-in Kerberos and token based authentication and access control [32] and Tez natively integrates with the Hadoop security framework to provide the application with secure access. In addition to that, the inputs and outputs provided with Tez support encryption for data read across the network. Security is a real concern with the variety of data stored and concurrent access from multiple users. Being outside the data plane reduces the contribution of Tez in the threat surface of the application. The only interaction between Tez and the app is control metadata routed via events by Tez. This metadata is presented to Tez as an opaque binary payload and thus can be protected by the app by encryption or other techniques as deemed necessary. In the control plane, secure Hadoop provides Kerberos and token based authentication for applications to access storage or compute resources and Tez integrates with the secure API's exposed by Hadoop. Tez has some built-in input and output libraries for HDFS and local storage. In a secure Hadoop cluster, these libraries use HDFS token based authentication to access the data. In a secure cluster local data is written in the OS security content of the user and read via secure SSL channel provided by the YARN Shuffle Service.

Another aspect of security is isolation between tasks running on the same machine but belonging to different users. YARN provides this security isolation between containers by running the containers in the security context of the application user. Due to its fine-grained, ephemeral task model, Tez can leverage this container security by running tasks for single user in the containers of an application, thus guaranteeing user level isolation. This is much harder to achieve when using application engines that deploy service daemons. The daemons need to run tasks from different users in the same daemon process, making security isolation difficult or impossible. To work-around this, multiple instances of the service daemons need to be launched (one per user) and that may reduce resource utilization, as described above. We believe that the fine-grained, ephemeral task model of Tez makes it more suitable for secure and multi-tenant YARN clusters.

Fault Tolerance. Failures are a norm in clusters of commodity hardware. Failures can be on the compute nodes or the network. Tez provides robust fault tolerance against failures using task re-execution as a means of recovering from errors. When a task fails due to machine errors, it is re-executed on a different machine. Task re-execution based fault tolerance depends on deterministic and side-effect free task execution. Being side-effect free allows a task to be executed multiple times. Being deterministic, guarantees that if identical code is executed on identical input data then it will produce identical output data for each execution. These enable the system to safely re-execute tasks to recover from failures and data loss. Since the outputs are identical, the already completed consumer tasks of that output do not need to be re-executed. This limits the amount of re-execution and reduces the cost of failures.

Since Tez is not on the data plane, it exposes an InputReadError event that task inputs can use to notify Tez about loss of intermediate data. Using the DAG dependency information Tez can determine which task outputs produced the missing data and re-execute that task to regenerate the data. It may happen that the re-executed task also reports an input error. This would cause Tez to go up one more step in the DAG dependency and so on, until it has found stable intermediate data. The edge API allows for the specification of intermediate data resiliency such that Tez can be informed that a given edge data has been reliably stored, thus creating an barrier to cascading failures. Tez built-in input/output libraries leverage heuristics inherited from MapReduce for mitigating and recovering from network errors and cascading failures when shuffling large volumes of data. E.g. temporary network errors are retried with back-off before reporting an error event. Partially fetched data is cached and the consumer task stays alive until the remaining missing data is regenerated. The Tez AM periodically checkpoints its state. If the node, that is running the Tez AM, has a failure then YARN will restart the AM on another node and the AM can recover its state from the checkpoint data.

Tez tightly integrates with YARN to handle planned and unplanned cluster outages. It listens to notifications about machine loss or decommissioning and pro-actively re-executes the tasks that were completed on such machines. This decreases the chance that consumers of those task outputs will fail. Tez also understands actions taken by YARN such as preempting containers for capacity rebalancing or terminating badly behaving containers and responds to those actions appropriately.

Limitations. The current implementation of Tez is Java based and thus we are limited to JVM based applications right now. The Tez based MapReduce implementation has successfully executed non-JVM user code using MapReduce's approach of forking off non-Java code. However, a more native Tez support would need non-Java APIs for writing IPOs and executors to support them.

Work is in progress to have a portable text based representation of the DAG API to enable non-Java compilers that target Tez. Tez can only be used in a YARN based Hadoop cluster because the current scheduling implementation has been written for YARN. Our recent work to enable developer debugging capability has abstracted out the dependence on a cluster. Extensions of that work could enable Tez to utilize other systems for execution. The current fault tolerance model depends on the assumption that intermediate task outputs are localized to the machine on which the task ran. Thus intermediate data loss causes re-execution of the task on a different machine. This may not be true of all IOs. e.g. if data is being streamed directly over the network. Also, such network streaming may result in collapse of the connected streaming sub-graph, which would need extensions of the fault tolerance model to handle such correlated failures.

5. APPLICATIONS & ADOPTION

In this section we will outline projects that have been updated or prototyped to use the Tez framework to run on YARN. These projects represent a significant variety of application types and help show the applicability of the Tez APIs for modeling and building high-performance data processing applications.

5.1 Apache MapReduce

MapReduce is a simple yet powerful means of scalable data processing that can be credited with ushering in the era of inexpensive hyper-scale data processing. At its core, it is a simple 2 vertex connected graph. In Tez, it can be represented with a map vertex and a reduce vertex that are connected using a scatter-gather edge. Tez has a built-in MapProcessor and a ReduceProcessor that run in the respective Map and Reduce vertices and provide the MapReduce interface functionality. Thus MapReduce can be easily written as a Tez based application and, in fact, the Tez project comes with a built-in implementation of MapReduce. Any MapReduce based application can be executed without change using the Tez version of MapReduce by simply changing a MapReduce configuration on a YARN cluster.

5.2 Apache Hive

Apache Hive is one of the most popular SQL-based declarative query engines in the Hadoop ecosystem. It used to translate queries written in HiveQL (a SQL-like dialect) to MapReduce jobs and run them on a Hadoop cluster. Like other SQL engines, Hive translates the queries into optimized query trees. Often these translations to MapReduce were inefficient due to the restricted expressiveness of MapReduce. These trees translate directly to DAGs specified using the Tez DAG API. Thus they can be represented efficiently in Tez. In addition, Hive uses custom edges (written to the Tez API) to perform sophisticated joins that were hitherto very difficult to do. E.g. In a join variant called Dynamically Partitioned Hash Join; Hive uses a custom vertex manager to determine which subsets of data shards to join with each other and creates a custom edge that routes the appropriate shards to their consumer tasks. While these query planning improvements provide algorithmic performance gains, Hive benefits from the execution efficiencies (Section 4.2) to get significant performance benefits out of the box. This integration has been implemented by the Apache Hive community with Hive 0.13 being the first release of Hive to have Tez integration[7]. Further Tez-based optimizations (like dynamic partition pruning) have been released in Hive 0.14 with more work planned in future releases.

5.3 Apache Pig

Apache Pig provides a procedural scripting language (named PigLatin) that is designed to write complex batch processing ETL pipelines. The procedural nature of PigLatin allows the creation of complex DAGs with vertices having multiple outputs. In MapReduce, applications could write only 1 output and thus were forced to use creative workarounds like tagging the data or writing side-effect outputs. Being able to model multiple outputs explicitly via the Tez APIs allows the planning and execution code in Pig to be clean and maintainable. Pig supports joins with data-skew detection and this was earlier done by running different MapReduce jobs to read and sample the data, then create histograms based on the samples on the client machine and finally run another job that uses the histogram to read and partition the data. This complex workflow of jobs can now be represented as a sub-graph of any Pig DAG when using Tez. The samples are collected in a histogram vertex that calculates the histogram. The histogram is sent via an event to a custom vertex manager that re-configures the partition vertex to perform the optimal partitioning. Pig developers from Yahoo, Netflix, LinkedIn, Twitter and Hortonworks came together to implement this integration. Pig 0.14 is the first release of Pig to support Tez based execution in addition to MapReduce[8].

5.4 Apache Spark

Apache Spark [38] is a new compute engine that provides an elegant language integrated Scala API for distributed data processing. It specializes in machine learning but the API lends itself to mixed workloads and pipeline processing. Data distribution metadata is captured at the language layer in a concept called Resilient Distributed Dataset (RDD) [37] and this metadata is used during compilation to construct a DAG of tasks that perform the distributed computation. Spark comes with its own processing engine service to execute these tasks. We were able to encode the post-compilation Spark DAG into a Tez DAG and run it successfully in a YARN cluster that was not running the Spark engine service. User defined Spark code is serialized into a Tez processor payload and injected into a generic Spark processor that deserializes and executes the user code. This allows unmodified Spark programs to run on YARN using Spark's own runtime operators. Apache Hive and Pig were already designed to translate to MapReduce and the translation to Tez is an evolutionary step. Modeling a net new engine like Spark on YARN using Tez presents a strong proof of the generality and modeling power of the Tez framework. Tez sessions also enable Spark machine learning iterations to run efficiently by submitting the per-iteration DAGs to a shared Tez session. This work is an experimental prototype and not part of the Spark project[22].

5.5 Apache Flink

Apache Flink [5] is a new project in the Apache community with roots in the Stratosphere research project of the TU Berlin data management community. It is a parallel processing engine that provides programming APIs in Java and Scala, a cost-based optimizer for these APIs, as well as its own execution engine. Flink is another example of a new platform that could be integrated with YARN using the Tez framework instead of running it as a standalone service. The post-optimization DAG is translated to a Tez DAG for this integration. While Apache Hive and Pig work on key-value data formats and could use the built-in Tez inputs and outputs, Flink keeps intermediate data in a custom binary format. This format can be used to perform operations like group by etc. without much deserialization overhead. The pluggable and composable Tez task model allowed Flink to incorporate its runtime operators and binary format inside Tez tasks, thus allowing unmodified programs to run on

YARN using Flink's native runtime model. This work is currently in progress in the Apache Flink developer community[6].

5.6 Commercial Adoption

The customizability and performance focused design of Tez has resulted in rapid uptake from commercial software projects. Concurrent Inc. supports an open source language integrated API in Java, called Cascading, for distributed data processing. Cascading has been updated to run on YARN using the Tez framework with promising performance results. Scalding is a Scala dialect over Cascading that automatically gets the benefits of Tez via Cascading. Cascading 3.0 is currently available as a developer preview and integrates with Tez [3]. Datameer provides a visual analytics platform that uses Tez to run optimized analytics queries on YARN. It also uses Tez sessions to maintain a query service pool for fast response times in a secure, multi-tenant environment. Datameer 5.0 is the first release that uses Tez [33]. Release 8 of Syncsort's Hadoop product, DMX-h, shipped with an intelligent execution layer to enable transparent targeting of execution frameworks other than Mapreduce. Following this, they are in the process of integrating with Tez as one of their supported execution frameworks [11].

5.7 Deployments

Apache Tez has been deployed across multiple organizations and on a variety of cluster configurations. Most prominently, Yahoo! has deployed Tez on multiple clusters ranging from 100s to 1000s of nodes to run Hive and Pig with Tez [35]. LinkedIn has completed migration of all their Hadoop clusters to YARN and is running Hive and Pig with Tez. Microsoft Azure has deployed Hive with Tez as part of its cloud Hadoop offering [10]. Hortonworks has provided Tez as a part of its Hadoop distribution since April 2014 and is seeing rapid adoption of Tez by its install base. At the time of publication, nearly 100 Hortonworks customers have explored the capabilities of Tez.

The growing adoption of Tez we described in this section provides a qualitative metric of the project success. In the next section, we turn to the experiment results obtained as a result of these applications being integrated with Tez.

6. EXPERIMENTAL RESULTS

We devote this section to present several experiments, showcasing how Tez-based implementations of Hive, Pig and Spark on YARN outperform their original implementation on YARN. The experiments are derived from both standard benchmarks, and production workloads.

6.1 Hive 0.14 Performance Tests

Hive utilizes various features available in Tez, such as broadcast edges, runtime re-configuration and custom vertex managers, to achieve a better overall execution of the user's processing goals. In conjunction with Hive 0.14's Cost Based Optimizer, Tez enables the execution of bushy join plans which can take advantage of intermediate broadcast joins. The pluggable task model of Tez allows Hive to use custom vectorized operators throughout the processing. Custom edges are used to perform efficient Hive sort-merge-bucket joins. In Figure 8 we show the impact of these optimizations on a TPC-DS derived Hive workload, run at 30 terabytes scale on a 20 node cluster with 16 cores, 256Gb RAM and 6 x 4Tb drives per node. The Tez-based implementation substantially outperforms the traditional MapReduce based one.

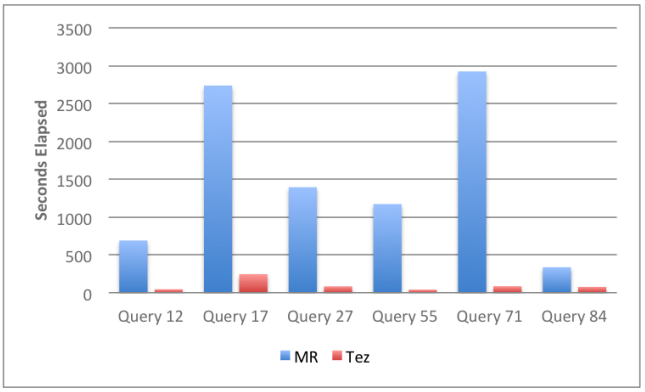


Figure 8: Hive: TPC-DS derived workload (30TB scale)

6.2 Yahoo Hive Scale Tests

In Figure 9 we show a comparative scale test of Hive on Tez, with a TPC-H derived Hive workload [35], at 10 terabytes scale on a 350 node research cluster with 16 cores, 24Gb RAM and 6 x 2Tb drives per node. This was presented at Hadoop Summit 2014, San Jose. This shows that Tez based implementation outperforms the MapReduce based implementation at large cluster scale.

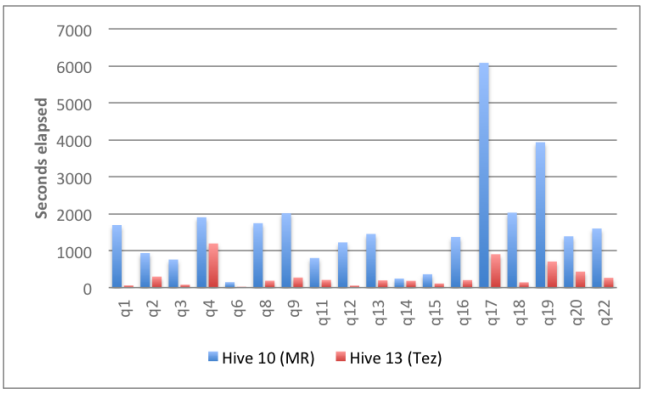


Figure 9: Hive: TPC-H derived workload at Yahoo (10TB scale)

6.3 Yahoo Pig Production Tests

At Yahoo!, Pig on Tez was tested on large production ETL pig jobs that run in the order of minutes to hours. To test different aspects of scale and features of the implementation, the pig scripts run had varying characteristics like terabytes of input, 100K+ tasks, complex DAGs with 20 to 50 vertices and doing a combination of various operations like group by, union, distinct, join, order by, etc. The tests were run on different production clusters where data resided and already running regular jobs with average utilization of 60-70%. The cluster had 4,200 servers, 46 PB HDFS storage and 90TB aggregate memory. Most data nodes were with 12/24G RAM, 2 x Xeon 2.40GHz, 6 x 2TB SATA on Hadoop 2.5, RHEL 6.5, JDK 1.7 There were performance improvements of 1.5 to 2x compared to MapReduce keeping all the configuration (memory, shuffle configuration, etc.) same as MapReduce. Figure 10 shows the results of these tests.

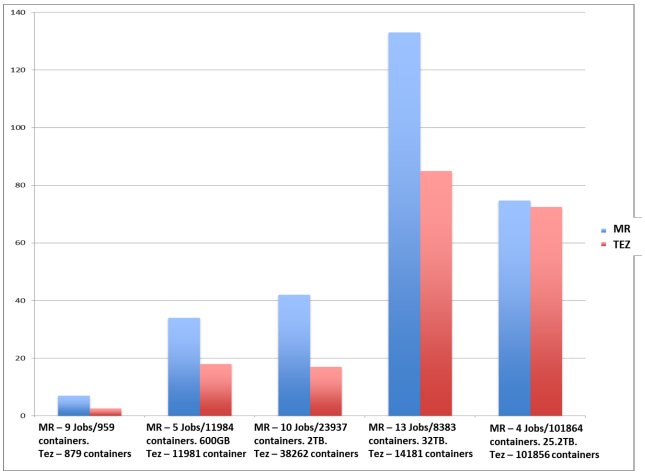


Figure 10: Pig workloads at Yahoo

6.4 Pig KMeans Iteration Tests

As noted in Section 4.2, Tez session and container-reuse features work in favor of fast iterative workloads, which require consecutive DAGs to execute over the same data-set. In Figure 11 we show performance improvements for a K-means iterative PIG script [20], run for 10, 50 and 100 iterations against a 10,000 row input dataset on a single node. This was presented at the Hadoop Summit 2014, San Jose.

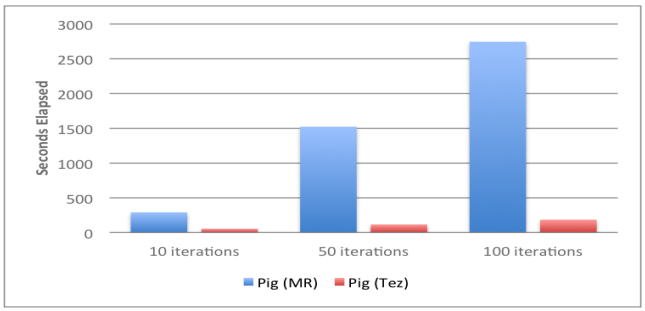


Figure 11: Pig iteration workload (k-means)

6.5 Spark Multi-Tenancy on YARN Tests

As explained in Section 4.3, Tez's ephemeral task based model is better for multi-tenancy and resource-sharing. This is demonstrated in Figure 12 by comparing service-based vs Tez-based implementations of Spark on YARN. The Tez based implementation releases idle resources that get assigned to other jobs that need them, thus speeding them up, as shown in Figure 13, while the service-based implementation holds on to resources for the life of the service. For the experiment, we have a 5-user concurrency test [22] of partitioning a TPC-H lineitem data-set along the L SHIPDATE column, on a 20 node cluster. The tests were run across data sets which correspond to 100 GB, 200 GB, 500 Gb and 1 TB warehouse scale factors. The cluster used to run this workload was identical to the Hive 0.14 benchmarks, having 16 cores, 256Gb of RAM and 6 x 4Tb disks per node.

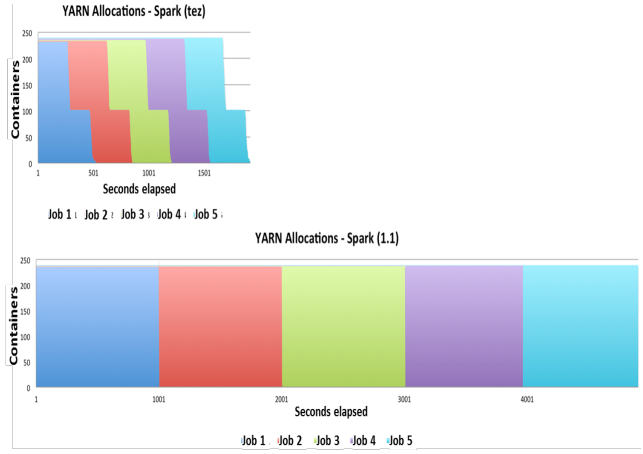


Figure 12: Sharing a cluster across 5 concurrent Spark jobs with Tez (identical x-axis time scales)

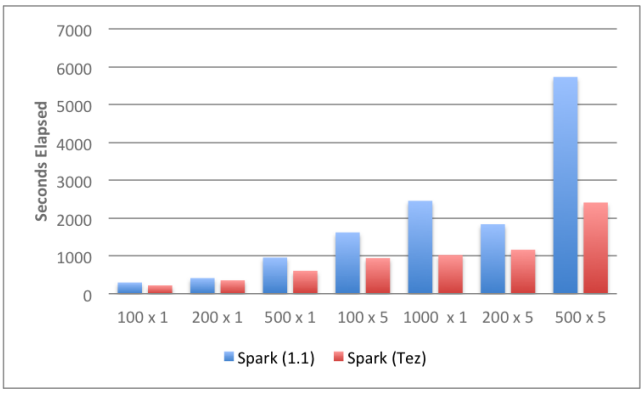


Figure 13: Spark Multi-Tenancy on YARN Latency Results

7. OPEN SOURCE AND FUTURE WORK

Apache Tez has been developed as an open source project under the Apache Software Foundation. It's a community driven project with contributors from Microsoft, Yahoo, Hortonworks, LinkedIn among others as well as individual enthusiasts. The source code for the project is available at http://git.apache.org/tez.git and the project website is at [2]

The open architecture of Tez and its fundamentally customizable design lends it to becoming a platform for experimentation and innovation. We believe that the current use cases built on Tez are only the initial steps of a longer journey. There is considerable interest in a variety of areas to improve and leverage Tez. Progressive query optimization which allows a complex and large query to be executed partially and optimized incrementally as the query proceeds. Apache Hive [30] and Apache Calcite [17] are working together on materialized views for speeding up common sub-queries. We want to provide deep integration with in-memory storage capabilities being added to HDFS [12] so that Tez applications can benefit from in-memory computing. Tez currently supports Java applications and extending it to support other languages would widen the scope of applications built using Tez. Another area of interest is tooling for debugging failure and performance performance bottlenecks. Increasingly, geographical distribution and legal/privacy requirements are making cross data-center job execution important [34]. Improving the Tez orchestration and API to model such jobs

may help in executing them efficiently. The above are only a few of the many possibilities in which Tez may be evolved or used by academic and commercial communities. Many runtime optimizations are also in the works. E.g. automatically choosing optimal data transport mechanisms like in-memory data for machine co-located tasks or using a reliable store for outputs of extremely long tasks so that their outputs are safeguarded against loss. An tactical idea is to create tooling that enables a full MapReduce workflow to be stitched into a single Tez DAG. This would enable legacy MapReduce workflows to easily use the MapReduce im

8. RELATED WORK

Apache Tez has been fortunate to learn from the development and experiences of similar systems such as Dryad, Hyracks and Nephele [24, 15, 14]. All of them share the concept of modeling data processing as DAGs with vertices representing application logic and edges or channels representing data transfer. Tez makes this more fine-grained by adding the concepts of inputs, processor and outputs to formally define the tasks executing the DAGs, leading to clear separation of concerns and allowing pluggable task composition. All of them participate to varied extents in the data plane and define some form of data format, which allows applications to define custom formats that derive from the base definition. All of them define on-disk, over-network and in-memory communication channels. Tez, on the other hand, does not define any data format and is not part of the data plane at all. On a similar note, Hyracks defines an operator model for execution that allows it to understand the data flow better for scheduling. Tez treats processors as black boxes so that the application logic can be completely decoupled from the framework. Nephele is optimized for cloud environments where it can elastically increase or decrease resources and choose appropriate virtual machines. Tez also enables resource elasticity by acquiring and releasing resources in YARN. Dryad and Tez share the concept of vertex managers for dynamic graph re-configurations. Tez takes this concept a step further by formalizing an API that allows the managers to be written without knowing the internals of the framework and also defining an event based communication mechanism that enables application code in tasks to communicate with application code in vertex managers in order to actuate the re-configurations. In addition, Tez adds the concept of input initializers to formally model primary data sources and apply runtime optimizations while reading them. Dryad schedules tasks when all the inputs of the tasks are ready to be consumed, to prevent scheduling deadlocks. Tez allows out of order execution for performance reasons and has built-in preemption to resolve scheduling deadlocks. Overall, Tez differs from these systems in its modeling capabilities and the design goal of being a library to build engines rather than being an engine by itself. MapReduce is, of course, the incumbent engine in the Hadoop ecosystem. Tez subsumes the MapReduce APIs such that it is possible to write a fully functional MapReduce application using Tez.

Dremel [26] is a processing framework for interactive analysis of large data sets based on multi-level execution trees that is optimized for aggregation queries and has motivated systems like Presto [9] and Apache Drill [23]. These, and other SQL query engines like Impala [25] or Apache Tajo [18], differ from Tez by being engines optimized for specific processing domains whereas Tez is a library to build data processing applications. Spark [38] is a new general purpose data processing engine. It exposes a Resilient Distributed Dataset (RDD) [37] based computation model that eventually gets executed on an in-memory storage and compute engine. Tez, again differs being a library and not a general purpose engine. Tez does not provide any storage service but applications can use existing in-memory stores, e.g. HDFS memory storage [12], to get the advantage of in-memory computing. The Spark notion of using RDDs as a means of implicitly capturing lineage dependency between steps of processing can be related to capturing that same dependency explicitly via defining the DAG using Tez APIs.

An important category of systems to compare against are other frameworks to build YARN-applications. The two most relevant in this space are Apache REEF [19] and Apache Twill [31]. These systems focus on a much broader class of applications (including services), than Tez, and thus provide a lower-level API. Tez focuses on supporting data-flow driven applications, and thus consciously chooses to provide a structured DAG-based control-flow.

9. CONCLUSIONS

Today, Hadoop is a booming ecosystem for large-scale data processing, blessed with an ever growing set of application frameworks, providing diverse abstractions to process data. We recognize that this is invaluable, yet we highlight substantial concerns of fragmentation and repeated work, as each application framework solves similar fundamental problems over and over again.

To address this issue we present Apache Tez, an *open-source framework designed to build data-flow driven processing engines*. Tez provides a scaffolding and libraries to facilitate the design and implementation of DAG-centric data processing applications, and focuses on re-use, while balancing customizability of the performance critical data plane. Tez makes a conscious effort to enable dynamic optimizations, such as partition pruning. Besides these key architectural choices, what sets Tez apart from other attempts of unifying frameworks is a sizeable open-source community, that is pushing Tez towards becoming the framework of choice for building DAG-orientied data processing engines. As of today, the most popular projects (Hive, Pig and Cascading) have integrated with Tez. We demonstrated experimentally that the Tez-based incarnations of these systems deliver substantial performance benefits beyond the qualitative argument of leveraging common functionalities.

We argue that the standardization we are promoting can foster even faster innovation, and enable integration plays that would be otherwise cumbersome (e.g., pipelines made up of multiple application frameworks). Tez's customizability and open-source community makes it an ideal playground for research, as novel ideas can be tested, integrated, and gain real-world impact with minimal overhead.

Acknowledgements

Apache Tez is an open source community driven project with contributions gratefully accepted from numerous individuals and organizations. In particular we would like to call out Rajesh Balamohan for keeping a watchful eye on performance and Tassapol Athiapinya and Yesha Vora for testing and system validation. We would like to thank members of other project communities who have helped in adopting and demonstrating the value of Tez. Notably, Gunther Hagleitner and Vikram Dixit for Apache Hive; Rohini Palaniswamy, Cheolsoo Park, Daniel Dai, Olga Natkovich, Mark Wagner and Alex Bain for Apache Pig; Chris Wensel for Cascading; Oleg Zhurakousky for Apache Spark; Kostas Tzoumas and Stephan Ewen for Apache Flink. We are also grateful to Yahoo and Hortonworks for providing experimentation infrastructure. We hope that the innovation platform provided by Tez will lead to further contributions from many more.

10. REFERENCES

[1] Understanding just-in-time compilation and optimization. https://docs.oracle.com/cd/E15289_01/doc.40/e15058/underst_jit.htm.

[2] Apache tez project website, 2014. http://tez.apache.org.

[3] Cascading integration with tez project, 2014. http://www.cascading.org/2014/09/17/cascading-3-0-wip-now-supports-apache-tez.

[4] Cascading: Project website, 2014. http://www.cascading.org.

[5] Flink: apache incubator project (formerly stratosphere), 2014. http://flink.incubator.apache.org/.

[6] Flink integration with tez project jira, 2014. http://issues.apache.org/jira/browse/FLINK-972.

[7] Hive integration with tez project jira, 2014. http://issues.apache.org/jira/browse/HIVE-4660.

[8] Pig integration with tez project jira, 2014. http://issues.apache.org/jira/browse/PIG-3446.

[9] Presto: Distributed sql query engine for big data, 2014. http:/prestodb.io.

[10] What's new in the hadoop cluster versions provided by hdinsight?, 2014. http://azure.microsoft.com/en-us/documentation/articles/hdinsight-component-versioning/.

[11] New syncsort big data software removes major barriers to mainstream apache hadoop adoption, 2015. http://www.syncsort.com/en/About/News-Center/Press-Release/New-Syncsort-Big-Data-Software.

[12] A. Agarwal. https://issues.apache.org/jira/browse/hdfs-5851, 2014. https://issues.apache.org/jira/browse/HDFS-5851.

[13] S. Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 137–142, New York, NY, USA, 2010. ACM.

[14] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *SOCC*, 2010.

[15] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[16] D. Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[17] Calcite developer community (formerly Optiq). Calcite: Apache incubator project, 2014. https://github.com/apache/incubator-calcite.

[18] H. Choi, J. Son, H. Yang, H. Ryu, B. Lim, S. Kim, and Y. D. Chung. Tajo: A distributed data warehouse system on large clusters. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1320–1323. IEEE, 2013.

[19] B.-G. Chun, T. Condie, C. Curino, C. Douglas, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, J. Rosen, et al. Reef: Retainable evaluator execution framework. *Proceedings of the VLDB Endowment*, 6(12):1370–1373, 2013.

[20] Daniel Dai and Rohini Palaniswamy. Pig on Tez: Low latency ETL, 2014. http://www.slideshare.net/Hadoop_Summit/pig-on-tez-low-latency-etl-with-big-data/25.

[21] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51, 2008.

[22] Gopal Vijayaraghavan and Oleg Zhurakousky. Improving Spark for Data Pipelines with Native YARN Integration, 2014. http://bit.ly/1IkZVvP.

[23] M. Hausenblas and J. Nadeau. Apache drill: interactive ad-hoc analysis at scale. *Big Data*, 1(2):100–104, 2013.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS*, 2007.

[25] M. Kornacker, A. Behm, V. Bittorf, and et al. Impala: A modern, open-source sql engine for hadoop. In *CIDR*, 2015.

[26] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, Sept. 2010.

[27] A. C. Murthy, C. Douglas, M. Konar, O. O'Malley, S. Radia, S. Agarwal, and V. KV. Architecture of next generation apache hadoop mapreduce framework. Technical report, Tech. rep., Apache Hadoop, 2011.

[28] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[29] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: Friends or foes? *Commun. ACM*, 53(1):64–71, Jan. 2010.

[30] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive – a warehousing solution over a map-reduce framework. In *PVLDB*, 2009.

[31] Twill developer community. Twill: Apache incubator project, 2014. http://twill.incubator.apache.org/.

[32] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *SOCC*, 2013.

[33] P. Voss. The challenge to choosing the "right" execution engine, 2014. http://bit.ly/1wXP9Wp.

[34] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese. Wanalytics: Analytics for a geo-distributed data-intensive world. In *CIDR*, 2015.

[35] Yahoo Hadoop Platforms Team. Yahoo Betting on Apache Hive, Tez, and YARN, 2014. http://yahoodevelopers.tumblr.com/post/85930551108/yahoo-betting-on-apache-hive-tez-and-yarn.

[36] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[38] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.