REEF: Retainable Evaluator Execution Framework

Markus Weimer[a], **Yingda Chen**[a], **Byung-Gon Chun**[c], **Tyson Condie**[b], **Carlo Curino**[a], **Chris Douglas**[a], **Yunseong Lee**[c], **Tony Majestro**[a], **Dahlia Malkhi**[f], **Sergiy Matusevych**[a], **Brandon Myers**[e], **Shravan Narayanamurthy**[a], **Raghu Ramakrishnan**[a], **Sriram Rao**[a], **Russell Sears**[d], **Beysim Sezgin**[a], and **Julia Wang**[a]

[a] Microsoft

[b] UCLA

[c] Seoul National University

[d] Pure Storage

[e] University of Washington

[f] VMware

Abstract

Resource Managers like Apache YARN have emerged as a critical layer in the cloud computing system stack, but the developer abstractions for leasing cluster resources and instantiating application logic are very low-level. This flexibility comes at a high cost in terms of developer effort, as each application must repeatedly tackle the same challenges (e.g., fault-tolerance, task scheduling and coordination) and re-implement common mechanisms (e.g., caching, bulk-data transfers). This paper presents REEF, a development framework that provides a control-plane for scheduling and coordinating task-level (data-plane) work on cluster resources obtained from a Resource Manager. REEF provides mechanisms that facilitate resource re-use for data caching, and state management abstractions that greatly ease the development of elastic data processing work-flows on cloud platforms that support a Resource Manager service. REEF is being used to develop several commercial offerings such as the Azure Stream Analytics service. Furthermore, we demonstrate REEF development of a distributed shell application, a machine learning algorithm, and a port of the CORFU [4] system. REEF is also currently an Apache Incubator project that has attracted contributors from several instititutions.[1]

Keywords

Big Data; Distributed Systems; Database; High Performance Computing; Machine Learning

---

[1] http://reef.incubator.apache.org

1. INTRODUCTION

Apache Hadoop has become a key building block in the new generation of scale-out systems. Early versions of analytic tools over Hadoop, such as Hive [44] and Pig [30] for SQL-like queries, were implemented by translation into MapReduce computations. This approach has inherent limitations, and the emergence of resource managers such as Apache YARN [46], Apache Mesos [15] and Google Omega [34] have opened the door for newer analytic tools to bypass the MapReduce layer. This trend is especially significant for iterative computations such as graph analytics and machine learning, for which MapReduce is widely recognized to be a poor fit. In fact, the website of the machine learning toolkit Apache Mahout [40] explicitly warns about the slow performance of some of its algorithms when run on Hadoop MapReduce.

Resource Managers are a first step in re-factoring the early implementations of MapReduce into a common scale-out computational fabric that can support a variety of analytic tools and programming paradigms. These systems expose cluster resources—in the form of machine slices—to higher-level applications. Exactly how those resources are exposed depends on the chosen Resource Manager. Nevertheless, in all cases, higher-level applications define a single *application master* that elastically acquires resources and executes computations on them. Resource Managers provide facilities for staging and bootstrapping these computations, as well as coarse-grained process monitoring. However, runtime management —such as runtime status and progress, and dynamic parameters—is left to the application programmer to implement.

This paper presents the Retainable Evaluator Execution Framework (REEF), which provides runtime management support for task monitoring and restart, data movement and communications, and distributed state management. REEF is devoid of a specific programming model (e.g., MapReduce), and instead provides an application framework on which new analytic toolkits can be rapidly developed and executed in a resource managed cluster. The toolkit author encodes their logic in a Job Driver—a centralized work scheduler —and a set of Task computations that perform the work. The core of REEF facilitates the acquisition of resources in the form of Evaluator runtimes, the execution of Task instances on Evaluators, and the communication between the Driver and its Tasks. However, additional power of REEF resides in its ability to facilitate the development of reusable data management services that greatly ease the burden of authoring the Driver and Task components in a large-scale data processing application.

REEF is, to the best of our knowledge, the first framework that provides a re-usable control-plane that enables systematic reuse of resources and retention of state across arbitrary tasks, possibly from different types of computations. This common optimization yields significant performance improvements by reducing I/O, and enables resource and state sharing across different frameworks or computation stages. Important use cases include pipelining data between different operators in a relational pipeline and retaining state across iterations in iterative or recursive distributed programs. REEF is an (open source) Apache Incubator

project to increase contributions of artifacts that will greatly reduce the development effort in building analytical toolkits on Resource Managers.

The remainder of this paper is organized as follows. Section 2 provides background on Resource Manager architectures. Section 3 gives a general overview of the REEF abstractions and key design decisions. Section 4 describes some of the applications developed using REEF, one being the Azure Stream Analytics Service offered commercially in the Azure Cloud. Section 5 analyzes REEF's runtime performance and showcases its benefits for advanced applications. Section 6 investigates the relationship of REEF with related systems, and Section 7 concludes the paper with future directions.

2. RISE OF THE RESOURCE MANAGERS

The first generation of Hadoop systems divided each machine in a cluster into a fixed number of slots for hosting map and reduce tasks. Higher-level abstractions such as SQL queries or ML algorithms are handled by translating them into MapReduce programs. Two main problems arise in this design. First, Hadoop clusters often exhibited extremely poor utilization (on the order of 5 – 10% CPU utilization at Yahoo! [17]) due to resource allocations being too coarse-grained.[2] Second, the MapReduce programming model is not an ideal fit for some applications, and a common workaround on Hadoop clusters is to schedule a "map-only" job that internally instantiates a distributed program for running the desired algorithm (e.g., machine learning, graph-based analytics) [38, 1, 2].

These issues motivated the design of a second generation Hadoop system, which includes an explicit resource management layer called YARN.[3] Additional examples of resource managers include Google Omega [34] and Apache Mesos [15]. While structurally different, the common goal is to directly lease cluster resources to higher-level computations, or jobs. REEF is designed to be agnostic to the particular choice of resource manager, while providing support for obtaining resources and orchestrating them on behalf of a higher-level computation. In this sense, REEF provides a logical/physical separation between applications and the resource management layer. For the sake of exposition, we focus on obtaining resources from YARN in this paper.[4]

Figure 1 shows a high-level view of the YARN architecture, and Figure 2 contains a table of components that we describe here. A typical YARN setup would include a single Resource Manager (RM) and several Node Manager (NM) installations; each NM typically manages the resources of a single machine, and periodically reports to the RM, which collects all NM reports and formulates a global view of the cluster resources. The periodic NM reports also provides a basis for monitoring the overall cluster health at the RM, which notifies relevant applications when failures occur.

A YARN job is represented by an Application Master (AM), which is responsible for orchestrating the job's work on allocated *containers* i.e., a slice of machine resources (some

---

[2]Hadoop MapReduce tasks are often either CPU or I/O bound, and slots represent a fixed ratio of CPU and memory resources.
[3]YARN: Yet Another Resource Negotiator
[4]Comparing the merits of different resource management layers is out of scope for this paper. REEF is primarily relevant to what happens with allocated resource, and not how resources are requested.

amount of CPU, RAM, disk, etc.). A client submits an AM package—that includes a shell command and any files (i.e., binary executables, configu-rations) needed to execute the command—to the RM, which then selects a single NM to host the AM. The chosen NM creates a shell environment that includes the file resources, and then executes the given shell command. The NM monitors the AM for resource usage and exit status, which the NM includes in its periodic reports to the RM. At runtime, the AM uses an RPC interface to request containers from the RM, and to ask the NMs that host its containers to launch a desired program. Returning to Figure 1, we see two AM instances running, each with allocated containers executing a job-specific task.

2.1 Example: Distributed Shell on YARN

To further set the stage, we briefly explain how to write an application directly on YARN i.e., without REEF. The YARN source code contains a simple example implementation of a distributed shell (DS) application. Within that example is code for submitting an AM package to the RM, which proceeds to launch a distributed shell AM. After starting, the AM establishes a periodic heartbeat channel with the RM using a YARN provided client library. The AM uses this channel to submit requests for containers in the form of resource specifications: such as container count and location (rack/machine address), and hardware requirements (amount of memory/disk/cpu). For each allocated container, the AM sets up a launch context—i.e., file resources required by the executable (e.g., shell script), the environment to be setup for the executable, and a command-line to execute—and submits this information to the NM hosting the container using a YARN provided client library. The AM can obtain the process-level status of its containers from the RM or more directly with the host NM, again using a YARN provided client library. Once the job completes (i.e., all containers complete/exit), the AM sends a completion message to the RM, and exits itself.

The YARN distribution includes this distributed shell program as an exercise for interacting with its protocols. It is around 1300 lines of code. A more complete distributed shell application might include the following features:

- Provide the result of the shell command to the client.

- More detailed error information at the AM and client.

- Reports of execution progress at the AM and client.

Supporting this minimal feature set requires a runtime at each NM that executes the given shell command, monitors the progress, and sends the result (output or error) to the AM, which aggregates all results and sends the final output to the client. REEF is our attempt to capture such control-flow code, that we believe will be common to many YARN applications, in a general framework. In Section 4.1 we will describe a more feature complete version of this example developed on REEF in about half (530) the lines of code.

3. REEF

Resource managed applications leverage leased resources to execute massively distributed computations; here, we focus on data analytics jobs that instantiate compute tasks, which process data partitions in parallel. We surveyed the literature [53, 16, 8, 5, 11, 52] for

common mechanisms and design patterns, leading to the following common components within these architectures.

- A centralized per-job scheduler that observes the runtime state and assigns tasks to resources e.g., MapReduce task slots [11].

- A runtime for executing compute tasks and retaining state in an organized fashion i.e., contexts that group related object state.

- Communication channels for monitoring status and sending control messages.

- Configuration management for passing parameters and binding application interfaces to runtime implementations.

REEF captures these commonalities in a framework that allows application-level logic to provide appropriate implementations of higher-level semantics e.g., deciding which resources should be requested, what state should be retained within each resource, and what task-level computations should be scheduled on resources. The REEF framework is defined by the following key abstractions.

- Driver: application code that implements the resource allocation and Task scheduling logic.

- Evaluator: a runtime environment on a container that can retain state within Contexts and execute Tasks (one at a time).

- Context: a state management environment within an Evaluator, that is accessible to any Task hosted on that Evaluator.

- Task: the work to be executed in an Evaluator.

Figure 3 further describes REEF in terms of its runtime infrastructure and application framework. The figure shows an application Driver with a set of allocated Evaluators, some of which are executing application Task instances. The Driver Runtime manages events that inform the Driver of the current runtime state. Each Evaluator is equipped with a Context for capturing application state (that can live across Task executions) and Services that provide library solutions to general problems e.g., state checkpointing, group communication among a set of participating Task instances. An Environment Adapter insulates the REEF runtime from the underlying Resource Manager layer.[5] Lastly, REEF provides messaging channels between the Driver and Task instances—supported by a highly optimized event management toolkit (Section 3.1.2)—for communicating runtime status and state, and a configuration management tool (Section 3.1.3) for binding application logic and runtime parameters. The remainder of this section provides further details on the runtime infrastructure components (Section 3.1) and on the application framework (Section 3.2).

3.1 Runtime Infrastructure

The Driver Runtime hosts the application control-flow logic implemented in the Driver module, which is based on a set of asynchronous event-handlers that react to runtime events

---

[5]REEF is able to expose Resource Manager specific interfaces (e.g., for requesting resources) to application Driver modules.

e.g., resource allocations, task executions and failures. The Evaluator executes application tasks implemented in the Task module, and manages application state in the form of Contexts. The Environment Adapter deals with the specifics of the utilized resource management service. Lastly, Services add extensibility to the REEF framework by allowing isolated mechanisms to be developed and incorporated into an application's logic. This section further describes these runtime infrastructure components.

3.1.1 Environment Adapter—REEF is organized in a way that factors out many of the environment specific details into an Environment Adapter layer (see Figure 3), making the code base easy to port to different resource managers. The primary role of the Environment Adapter is to translate Driver actions (e.g., requests for resources) to the underlying resource manager protocol. We have implemented three such adapters:

1. **Local Processes:** This adapter leverages the host operating system to provide process isolation between the Driver and Evaluators. The adapter limits the number of processes active at a given time and the resources dedicated to a given process. This environment is useful for debugging applications and examining the resource management aspects of a given application or service on a single node.

2. **Apache YARN:** This adapter executes the Driver Runtime as a YARN Application Master [46]. Resource requests are translated into the appropriate YARN protocol, and YARN containers are used to host Evaluators.

3. **Apache Mesos:** This adapter executes the Driver Runtime as a "framework" in Apache Mesos [15]. Resource requests are translated into the appropriate Mesos protocol, and Mesos executors are used to host Evaluators.

Creating an Environment Adapter involves implementing a couple of interfaces. In practice, most Environment Adapters require additional configuration parameters from the application (e.g. credentials). Furthermore, Environment Adapters expose the underlying Resource Manager interfaces, which differ in the way that resources are requested and monitored. REEF provides a generic abstraction to these low-level interfaces, but also allows applications to bind directly to them for allocating resources and dealing with other subtle nuances e.g., resource preemption.

3.1.2 Event Handling—We built an asynchronous event processing framework called Wake, which is based on ideas from SEDA [49], Rx [26] and the Click modular router [19]. As we will describe in Section 3.2.1, the Driver interface is comprised of handlers that contain application code that react to events. Wake allows the Driver Runtime to tradeoff between cooperative *thread sharing* that synchronously invokes these event handlers in the same thread, and asynchronous *stages*, where events are queued for execution inside of an independent thread pool. Using Wake, the Driver Runtime has been designed to prevent blocking from long-running network requests and application code. In addition to handling local event processing, Wake also provides remote messaging facilities built on top of Netty [43]. We use this for a variety of purposes, including full-duplex control-plane messaging and a range of scalable data movement and group communication primitives. The latter are used every day to process millions of events in the Azure Streaming Service (see Section

4.4). Lastly, we needed to guarantee message delivery to a logical Task that could physically execute on different Evaluators e.g., due to a prior failure. Wake provides the needed level of indirection by addressing Tasks with a logical identifier, which applications bind to when communicating among Tasks.

3.1.3 Tang Configuration—Configuring distributed applications is well-known to be a difficult and error prone task [48, 31]. In REEF, configuration is handled through dependency injection, which is a software design pattern that binds dependencies (e.g., interfaces, parameter values, etc.) to dependent objects (e.g., class implementations, instance variables, constructor arguments, etc.). Google's Guice [13] is an example of a dependency injection toolkit that we used in an early version of REEF. The Guice API is based on binding patterns that link dependencies (e.g., application Driver implementations) to dependents (e.g., the REEF Driver interface), and code annotations that identify injection targets (e.g., which class constructors to use for parameter injection). The dependency injection design pattern has a number of advantages: client implementation independence, reduction of boilerplate code, more modular code, easier to unit test. However, it alone did not solve the problem of mis-configurations, which often occurred when instantiating application Driver, Context, or Task implementations on some remote container resources, where it was very difficult to debug.

This motivated us to develop our own dependency injection system called Tang, which restricts dynamic bind patterns.[6] This restriction allows Tang configurations to be strongly typed and easily verified for correctness through static analysis of bindings; prior to instantiating client modules on remote resources, thus allowing Tang to catch mis-configuration issues early and provide more guidance into the problem source. More specifically, a Tang specification consists of binding patterns that resolve REEF dependencies (e.g., the interfaces of a Driver and Task) to client implementations. These binding patterns are expressed using the host language (e.g., Java, C#) type system and annotations, allowing unmodified IDEs such as Eclipse or Visual Studio to provide configuration information in tooltips, auto-completion of configuration parameters, and to detect a wide range of configuration problems (e.g., type checking, missing parameters) as you edit your code. Since such functionality is expressed in the host language, there is no need to install additional development software to get started with Tang. The Tang configuration language semantics were inspired by recent work in the distributed systems community on CRDTs (Commutative Replicated Data Types) [35] and the CALM (Consistency As Logical Monotonicity) conjecture [3]. Due to space issues, we refer the reader to the online documentation for further details (see http://reef.incubator.apache.org/tang.html).

3.1.4 Contexts—Retaining state across task executions is central to the REEF design, and to the support for iterative data flows that cache loop invariant data or facilitate delta-based computations e.g., Naiad [27] and Datalog. Moreover, we also needed the option to clean up state from prior task executions, which prompted the design of stackable contexts in the

[6]Injection of dependencies via runtime code, or what Guice calls "provider methods."

Evaluator runtime. Contexts add structure to Evaluator state, and provide the Driver with control over what state gets passed from one task to the next, which could cross a computational stage boundary. For example, assume we have a hash-join operator that consists of a build stage, followed by a probe stage. The tasks of the build stage construct a hash-table—on the join column(s) of dataset A—and stores it in the root context that will be shared with the tasks of the probe stage, which performs the join with dataset B by looking up matching A tuples in the hash-table. Let us further assume that the build stage tasks require some scratch space, which is placed in a (child) scratch context. When the build stage completes, the scratch context is discarded, leaving the root context, and the hash-table state, for the probe stage tasks. For REEF applications, Contexts add fine-grained (task-level) mutable state management, which could be leveraged for building a DAG scheduler (like Dryad [16], Tez [33], Hyracks [8]), where vertices (computational stages) are given a "localized" context for scratch space, and use the "root" context for passing state.

3.1.5 Services—The central design principle of REEF is in factoring out core functionalities that can be re-used across a broad range of applications. To this end, we allow users to deploy services as part of the Context definition. This facilitates the deployment of distributed functionalities that can be referenced by the application's Driver and Tasks, which in turn eases the development burden of these modules. For example, we provide a name-based communication service that allows developers to be agnostic about reestablishing communication with a Task that was re-spawned on a separate Evaluator; this service works in concert with Wake's logical Task addressing. Services are configured through Tang, making them easy to compose with application logic.

3.2 Application Framework

We now describe the framework used to capture application logic i.e., the code written by the application developer. Figure 4 presents a high-level control-flow diagram of a REEF application. The control channels are labeled with a number and a description of the interaction that occurs between two entities in the diagram. We will refer to this figure in our discussion by referencing the control-flow channel number. For instance, the client (top left) initiates a job by submitting an Application Master to the Resource Manager (**control-flow 1**). In REEF, an Application Master is configured through a Tang specification, which requires (among other things) bindings for the Driver implementation. When the Resource Manager launches the Application Master (**control-flow 2**), the REEF provided Driver Runtime will start and use the Tang specification to instantiate the Driver components i.e., event-handlers described below. The Driver can optionally be given a channel to the client (**control-flow 7**) for communicating status and receiving commands e.g., via an interactive application.

3.2.1 Driver—The Driver is responsible for scheduling the task-level work of an application. For instance, a Driver that schedules a DAG of data-processing elements— common to many data-parallel runtimes [52, 16, 39, 8, 5, 53]—would launch (per-partition) tasks that execute the work of individual processing elements in the order of data dependencies. However, unlike most data-parallel runtimes[7], resources for executing such tasks must first be allocated from the Resource Manager. This added dimension increases

the scheduler complexity, which motivated the design of the REEF Driver to adopt an Reactive Extensions (Rx) [26] API[8] that consists of asynchronous handlers that react to events triggered by the runtime. We categorize the events, that a Driver reacts to, along the following three dimensions:

1. **Runtime Events:** When the Driver Runtime starts, it passes a start event to the Driver, which must react by either requesting resources (**control-flow 3**)—using a REEF provided request module that mimics the underlying resource management protocol—or by setting an alarm with a callback method and future time. Failure to do one of these two steps will result in the automatic shutdown of the Driver. In general, an automatic shutdown will occur when, at any point in time, the Driver does not have any resource allocations, nor any outstanding resource requests or alarms. Lastly, the Driver may optionally listen for the stop event, which occurs when the Driver Runtime initiates its shutdown procedure.

2. **Evaluator Events:** The Driver receives events for Evaluator allocation, launch, shutdown and failure. An *allocation* event occurs when the resource manager has granted a resource, from an outstanding request, to the Driver. The Evaluator allocation event API contains methods for configuring the initial Context state (e.g., files, services, object state, etc.), and methods to launch the Evaluator on the assigned resource (via **control-flow 4**), or release it (deallocate) back to the Resource Manager, triggering a *shutdown* event. Furthermore, Evaluator allocation events contain resource descriptions that provide the Driver with information needed to constrain state and assign tasks e.g., based on data-locality. A *launch* event is trigged when confirmation of the Evaluator bootstrap is received at the Driver. The launch event includes a reference to the initial Context, which can be used to add further sub-Context state (described in Section 3.1.4), and to launch a sequence of Task executions (one at a time) (via **control-flow 5**). A failure at the Evaluator granularity is assumed not to be recoverable (e.g., due to misconfiguration or hardware faults), and as a result, the relevant resource is automatically deallocated, and a *failure* event—containing the exception state—is passed to the Driver. On the other hand, Task events (discussed below) are assumed to be recoverable, and do not result in an Evaluator deallocation, allowing the Driver to recover from the issue; for example, an out-of-memory exception might prompt the Driver to configure the Task differently e.g., with a smaller buffer.

3. **Task Events:** All Evaluators periodically send status updates that include information about its Context state, running Services and the current Task execution status to the Driver Runtime (**control flow 6**). The Task execution status is surfaced to the Driver in the form of events: launch, message, failed, and completion. The *launch* event API contains methods for terminating or suspending the Task execution, and a method for sending messages—in the form of opaque byte arrays—to the running Task (via **control-flow 6**). Messages sent by the Driver are immediately pushed to the relevant Task to minimize latency. Task

---

[7]Today, exceptions include Tez [33] and Spark [52].
[8]Supported by Wake, which was described in Section 3.1.2.

implementations are also able to send messages (opaque byte arrays) back to the Driver, which are piggy-backed on the (periodic) Evaluator status updates. Furthermore, when a Task completes, the Driver is passed a *completion* event that includes a byte array "return value" of the Task main method (described below). We further note that REEF can be configured to limit the size of these messages in order to avoid memory pressure. Lastly, Task failures result in an event that contains the exception information, but (as previously stated) do not result in the deallocation of the Evaluator hosting the Task failure.

3.2.2 Task—A Task is a piece of application code that contains a main method, which will be invoked by the Evaluator. The application-supplied Task implementation has access to its configuration parameters and the Evaluator state, which is exposed as Contexts. The Task also has access to any services that the Driver may have started on the given Evaluator; for example, a Task could deposit its intermediate data in a buffer manager service so that it can be processed by a subsequent Task running on the same Evaluator.

A Task ends when its main method returns with an optional return value, which REEF presents to the Driver. The Evaluator catches any exceptions thrown by the Task and includes the exception state in the failure event passed to the Driver. A Task can optionally implement a handle for receiving messages sent by the Driver. These message channels can be used to instruct the Task to suspend or terminate its execution in a graceful way. For instance, a suspended Task could return its checkpoint state that can be used to resume it on another Evaluator. To minimize latency, all messages asynchronously sent by the Driver are immediately scheduled by Wake to be delivered to the appropriate Task i.e., REEF does not wait for the next Evaluator "heartbeat" interval to transfer and deliver messages. Wake could be configured to impose a rate limitation, but we have not explored that approach in this initial version, nor have we encountered such a bottleneck in our applications.

3.3 Implementation—REEF's design supports applications in multiple languages; it currently supports Java and C#. Both share the core Driver Runtime Java implementation via a native (C++) bridge, therefore sharing advancements of this crucial runtime component. The bridge forwards events between Java and C# application Driver implementations. The Evaluator is implemented once per language to avoid any overhead in the performance-critical data path.

Applications are free to mix and match Driver side event handlers in Java and C# with any number of Java and C# Evaluators. To establish communications between Java and C# processes, Wake is implemented in both languages. Tang is also implemented in both languages, and supports configuration validation across the boundary; it can serialize the configuration data and dependency graph into a neutral form, which is understood by Tang in both environments. This is crucial for the early error detection in a cross-language applications. For instance, a Java Driver receives a Java exception when trying to submit an ill-configured C# Task *before* attempting to launch the Task on a remote Evaluator.

To the best of our knowledge, REEF is the only distributed control flow framework that provides this deep integration across such language boundaries. Figure 5 gives an overview

of the effort involved in the development of REEF, including its cross-language support.[9] About half of the code is in Wake and Tang, while the other half is in the REEF core runtime. Interestingly, both Tang and Wake are bigger in C# than in Java. In the case of Wake, this is largely due to the extensive use of the Netty Java library, which is not available in C#. For Tang, its Java implementation relies heavily on reflection and leverages the runtime leniency of the Java type system; a luxury that a more rigid and expressive type system like the C# runtime does not permit.

3.4 Discussion—REEF is an active open-source project that started in late 2012. Over the past two years, we have refined our design based on feedback from many communities. The initial prototype of REEF's application interface were based on the Java Concurrency Library. When the Driver made a request for containers, it was given a list of objects representing allocated evaluators wrapped in Java Futures. This design required us to support a pull-based API, whereby the client could request the underlying object, even though the container for that object was not yet allocated, turning it into blocking method call. Extending the Future interface to include callbacks somewhat mitigated this issue. Nevertheless, writing distributed applications, like a MapReduce runtime, against this pull-based API was brittle; especially in the case of error handling e.g., exceptions thrown in arbitrary code interrupted the control-flow in a manner that was not always obvious, instead of being pushed to a specific (e.g., Task) error event-handler that has more context. As a result, we rewrote the REEF interfaces around an asynchronous event processing (push-based) model implemented by Wake, which greatly simplified both the REEF runtime and application-level code. For example, under the current event processing model, we have less of a need for maintaining bookkeeping state e.g., lists of Future objects representing outstanding resource requests. Wake also simplified performance tuning by allowing us to dedicate Wake thread pools to heavily loaded event handlers, without changes to the underlying application (handler) code.

4. APPLICATIONS

This section describes several applications built on REEF, ranging from basic applications to production level services. We start with an interactive distributed shell to further illustrate the life-cycle of a (basic) REEF application. Next, we highlight the benefits of developing on REEF with a novel class of machine learning research enabled by the REEF abstractions. We then conclude with a description of two real-world applications that leverage REEF to deploy on YARN, emphasizing the ease of development on REEF. The first is a Java version of CORFU [4], which is a distributed log service. The second is Azure Streaming Analytics, which is a publicly available service deployed on the Azure Cloud Platform. Additional REEF applications and tutorials can be found at http://reef.incubator.apache.org.

4.1 Distributed Shell

We illustrate the life-cycle of a REEF application with a simple interactive distributed shell, modeled after the YARN example described in Section 2.1. Figure 6 depicts an execution of

---

[9]All numbers where computed on the Apache REEF git repository found at http://git.apache.org, commit fa353fdabc8912695ce883380fa962baea2a20fb

this application on two Evaluators (i.e., on two machines) that execute Tasks running a desired shell command. During the course of this execution, the Evaluators enter different states defined by time-steps t_1, t_2, t_3, t_4 e.g., in time-step t_2, both Evaluators are executing Task 1. The lines in the figure represent control flow interactions and are labeled with a numbering scheme (e.g., i1 for interaction 1) that we refer to in our description below.

The application starts at the Client, which submits the Distributed Shell Driver (DSD) to the Resource Manager (RM) for execution; this interaction is labeled (i1). The RM then launches the Driver Runtime as an Application Master. The Driver Runtime bootstrap process establishes a bidirectional communication channel with the Client (i3) and sends a start event to the DSD, which requests two containers (on two separate machines) with the RM (i2). The RM will eventually send container allocation notifications to the Driver Runtime, which sends allocation events to the DSD. The DSD uses those events to submit a root Context—defining the initial state on each Evaluator—to the Driver Runtime, which uses the root Context configuration to launch the Evaluators in containers started by the Node Managers.

The Evaluator bootstrap process establishes a bidirectional connection to the Driver Runtime (i4). At time t_1, the Evaluator informs the Driver Runtime that it has started and that the root Context is active. The Driver Runtime then sends two active context events to the DSD, which relays this information to the Client via (i3). The Client is then prompted for a shell command. An entered command is sent via (i3) and eventually received by the DSD in the form of a client message event. The DSD uses the shell command in that message to configure Task 1, which is submitted to the Driver Runtime for execution on both Evaluators. The Driver Runtime forwards the Task 1 configuration via (i4) to the Evaluators, which execute an instance of Task 1 in time-step t_2. Note that Task 1 may change the state in the root Context. When Task 1 completes in time-step t_3, the Evaluator informs the Driver Runtime via (i4). The DSD is then passed a completed task event containing the shell command output, which is sent to the client via (i3). After receiving the output of Task 1 on both Evaluators, the Client is prompted for another shell command, which would be executed in a similar manner by Task 2 in time-step t_4.

Compared to the YARN distributed shell example described in Section 2.1, our implementation provides cross-language support (we implemented it in Java and C#), is runnable in all runtimes that REEF supports, and presents the client with an *interactive* terminal that submits subsequent commands to retained Evaluators, avoiding the latency of spawning new containers. Further, the REEF distributed shell exposes a RESTful API for Evaluator management and Task submission implemented using a REEF HTTP Service, which takes care of tedious issues like finding an available port and registering it with the Resource Manager for discovery.

Even though the core distributed shell example on REEF is much more feature rich, it comes in at less than half the code (530 lines) compared to the YARN version (1330 lines).

4.2 Distributed Machine Learning

Many state-of-the-art approaches to distributed machine learning target abstractions like Hadoop MapReduce [10, 40]. Part of the attraction of this approach is the transparent handling of failures and other elasticity events. This effectively shields the algorithm developers from the inherently chaotic nature of a distributed system. However, it became apparent that many of the policy choices and abstractions offered by Hadoop are not a great fit for the iterative nature of machine learning algorithms [50, 1, 47]. This lead to proposals of new distributed computing abstractions specifically for machine learning [7, 52, 23, 24, 22]. Yet, policies for resource allocation, bootstrapping, and fault-handling remain abstracted way through a high-level domain specific language (DSL) [7, 52] or programming model [23, 24, 22].

In contrast, REEF offers a lower-level programming abstraction that can be used to take advantage of algorithmic optimizations. This added flexibility sparked a line of ongoing research that integrates the handling of failures, resource starvation and other elasticity challenges directly into the machine learning algorithm. We have found a broad range of algorithms can benefit from this approach, including linear models [28], principal component analysis [21] and Bayesian matrix factorization [6]. Here, we highlight the advantages that a lower-level abstraction like REEF offers for learning linear models, which are part of a bigger class of Statistical Query Model algorithms [18].

4.2.1 Linear Models—The input to our learning method is a dataset D of examples (x_i, y_i) where $x_i \in R^d$ denotes the features and $y_i \in R$ denotes the label of example i. The goal is to find a linear function $f_w(x_j) = \langle x_j, w \rangle$ with $w \in R^d$ that predicts the label for a previously unseen example. This goal can be cast as finding the minimizer $\hat{w}$ for the following optimization problem:[10]

$$\hat{w} = \underset{w}{argmin} \sum_{x,y \in D} l\left(f_w\left(x\right), y\right) = \underset{w}{argmin} \sum_{x,y \in D} l\left(\langle x, w \rangle, y\right)$$

Here, $l(f, y)$ is the loss function the model $\hat{w}$ is to minimize, e.g. the squared error $l\left(f, y\right) = \frac{1}{2}(f - y)^2$. This function is typically convex and differentiable in f and therefore the optimization problem (1) is convex and differentiable in w, and therefore can be minimized with a simple gradient-based algorithm.

The core gradient computation of the algorithm decomposes per example. This allows us to partition the dataset D into k partitions $D_1, D_2, ..., D_k$ and compute the gradient as the sum of the per-partition gradients. This property gives rise to a simple parallelization strategy: assign each Evaluator a partition D_i and launch a Task to compute the gradient on a per-partition basis. The per-partition gradients are aggregated (i.e., summed up) to a global gradient, which is used to update the model w. The new model is then broadcast to all Evaluator instances, and the cycle repeats.

---

[10]Note that we omit the regularizer which, despite its statistical importance, does not affect the distribution strategy.

4.2.2 Elastic Group Communications—In parallel work, we designed an elastic group communications library as a REEF Service that exposes Broadcast and Reduce operators familiar to Message Passing Interface (MPI) [14] programmers. It can be used to establish a communication topology among a set of leaf Task participants and a root Task. The leaf Tasks are given a Reduce operator to send messages to the root Task, which can aggregate those messages and use a Broadcast operator to send a message to all leaf Tasks. If the Reduce operation is associative, then a tree topology is established, with internal nodes performing the pre-aggregation steps. The Service also offers synchronization primitives that can be used to coordinate bulk-synchronous processing (BSP) [45] rounds. Crucially, the Service delegates topology changes to the application Driver, which can decide how to react to the change, and instruct the Service accordingly. For example, the loss of a leaf Task can be simply ignored, or repaired synchronously or asynchronously.[11] And the loss of the root Task can be repaired synchronously or asynchronously. The application is notified when asynchronous repairs have been made.

In an elastic learning algorithm, the loss of leaf Tasks can be understood as the loss of partitions D_i in the dataset. We can interpret these faults as being a sub-sample of the data, in the absence of any statistical bias that this approach could introduce. This allows us to tolerate faults algorithmically, and avoid pessimistic fault-tolerance policies enforced by other systems e.g., [52, 1, 11, 23, 24]. The performance implications are further elaborated in Section 5.2, and in greater detail in [28].

4.3 CORFU on REEF

CORFU [4] is a distributed logging tool providing applications with consistency and transactional services at extremely high throughput. There are a number of important use cases which a shared, global log enables:

- It may be used for driving remote checkpoint and recovery.

- It exposes a log interface with strict total-ordering and can drive replication and distributed locking.

- It may be leveraged for transaction management.

Importantly, all of these services are driven with no I/O bottlenecks using a novel paradigm that separates control from the standard leader-IO, which prevails in Paxos-based systems. In a nutshell, internally a CORFU log is striped over a collection of *logging units*. Each unit accepts a stream of logging requests at wire-speed and sequentializes their IO. In aggregate, data can be streamed in parallel to/from logging-units at full cluster bisection bandwidth. There are three operational modes, in-memory, non-atomic persist, and atomic-persist. The first logs data only in memory (replicated across redundant units for "soft" fault tolerance). The second logs data opportunistically to stable storage, with optional explicit sync barriers. The third persists data immediately before acknowledging appends. A soft-state *sequencer* process regulates appends in a circular fashion across the collection of stripes. A CORFU

---

[11]The loss of an internal node in a tree topology can be modeled as the loss of a set of leaf Task nodes.

master controls the configuration, growing and shrinking the stripe-set. Configuration changes are utilized both for failure recovery and for load-rebalancing.

The CORFU architecture perfectly matches the REEF template. CORFU components are implemented as task modules, one for the sequencer, and one for each logging-unit. The CORFU master is deployed in a REEF Driver, which provides precisely the control and monitoring capabilities that the CORFU master requires. For example, when a logging unit experience a failure, the Driver is informed, and the CORFU master can react by deploying a replacement logging unit and reconfiguring the log. In the same manner, the CORFU master interacts with the log to handle sequencer failures, to react when a storage unit becomes full, and for load-rebalancing.

An important special failure case is the CORFU master itself. For applications like CORFU, it is important that a master does not become a single point of failure. REEF provides Service utilities for triggering checkpointing and for restarting a Driver from a checkpoint. The CORFU master uses these hooks to backup the configuration-state it holds onto the logging units themselves. Should the master fail, a recovery CORFU Driver is deployed by the logging units.

In this way, REEF provides a framework that decouples CORFU's resource deployment from its state, allowing CORFU to be completely elastic for fault tolerance and load-management.

Using CORFU from REEF—A CORFU log may be used from other REEF jobs by linking with a CORFU client-side library. A CORFU client finds (via CORFULib) the CORFU master over a publicized URL. The master informs the client about direct ports for interacting with the sequencer and the logging-units. Then, CORFULib interacts with the units to drive operations like log-append and log-read directly over the interconnect.

CORFU as a REEF service—Besides running as its own application, CORFU can also be deployed as a REEF Service. The Driver side of this Service subscribes to the events as described above, but now in addition to the other event handlers of the application. The CORFU and application event handlers compose to form the Driver and jointly implement the control-flow of the application, each responsible for a subset of the Evaluators. This greatly simplifies the deployment of such an application, as CORFU then shares the event life-cycle with it and does not need external coordination.

4.4 Azure Stream Analytics

Azure Stream Analytics (ASA) is a fully managed stream processing service offered in the Microsoft Azure Cloud. It allows users to setup resilient, scalable queries over data streams that could be produced in "real-time." The service hides many of the technical challenges from its users, including machine faults and scaling to millions of events per second. While a description of the service as a whole is beyond the scope here, we highlight how ASA uses REEF to achieve its goals.

ASA implements a REEF Driver to compile and optimize–taking user budgets into consideration–a query into a data-flow of processing stages, similar to [30, 44, 33, 16, 53, 8, 5, 52]. Each stage is parallelized over a set of partitions i.e., an instance of a stage is assigned to process each partition in the overall stage input. Partitioned data is pipelined from producer stages to consumer stages according to the (compiled) data-flow. All stages must be started before query processing can begin on input data streams. The Driver uses the stage data-flow to formulate a request for resources; specifically, an Evaluator is requested per-stage instance. A Task is then launched on each Evaluator to execute the stage instance work on an assigned partition. It is highly desirable that this bootstrap process happens quickly to aid experimentation.

At runtime, an ASA Task is supported by two REEF Services, which aided in shortening the development cycle. The first is a communications Service built on Wake for allowing Tasks to send messages to other Tasks based on a logical identifier, which is independent to the Evaluator on which they execute, making Task restart possible on alternate Evaluator locations. The communication Service is highly optimized for low-latency message exchange, which ASA uses to communicate streaming partitions between Tasks. The second is the checkpointing Service that provides each Task with an API for storing intermediate state to stable storage, and an API to fetch that state e.g., on Task restart.

ASA is a production-level service that has had very positive influence on recent REEF developments. Most notably, REEF now provides mechanisms for capturing the Task-level log files—on the containers where the Task instances executed—to a location that can be viewed (postmortem) locally. Another recent development is an embedded HTTP server as a REEF Service that can be used to examine log files and execution status at runtime. These artifacts were motivated during the development and initial deployment phases of ASA. Further extensions and improvements are expected as more production-level services (already underway at Microsoft) are developed on REEF.

4.5 Summary

The applications described in this section underscore our original vision of REEF as being:

1. A flexible framework for developing distributed applications on Resource Manager services.

2. A standard library of reusable system components that can be easily composed (via Tang) into application logic.

Stonebraker and Cetintemel argued that the "one size fits all model" is no longer applicable to the database market [36]. We believe this argument naturally extends to "Big Data" applications. Yet, we also believe that there exists standard mechanisms common to many such applications. REEF is our attempt to provide a foundation for the development of that common ground in open source.

5. EVALUATION

Our evaluation focuses on microbenchmarks (Section 5.1) that examine the overheads of REEF for allocating resources, bootstrapping Evaluator runtimes, and launching Task

instances; we then report on a task launch overhead comparison with Apache Spark. Section 5.2 then showcases the benefits of the REEF abstractions with the elastic learning algorithm (from Section 4).

Experimental setup

We report experiments in three environments. The first is using the local process runtime. The second is based on YARN version 2.6 running on a cluster of 35 machines equipped with 128GB of RAM and 32 cores; each machine runs Linux and Java 1.7. We submit one job at a time to an empty cluster to avoid job scheduling queuing effects. Third, we leveraged Microsoft Azure to allocate 25 D4 instances (8 cores, 28 GB of RAM and 400 GB of SSD disk each) in an experiment that compares the overheads of REEF to Apache Spark [52].

5.1 Microbenchmark

Key primitive measurements—Figure 7 shows the time it takes to dispatch a local Wake Event, bootstrap an Evaluator, and launch a Task. There are roughly three orders of magnitude difference in time between these three actions. This supports our intuition that there is a high cost to reacquiring resources for different Task executions. Further, Wake is able to leverage multi-core systems in its processing of fine-grained events, achieving a throughput rate that ranges from 20-50 million events per second per machine.

Overheads with short-lived Tasks—In this experiment, the Driver is configured to allocate a fixed number of Evaluators and launch Tasks that sleep for one second, and then exit. This setup provides a baseline (ideal) job time interval (i.e., #Tasks * one second) that we can use to assess the combined overhead of allocating and bootstrapping Evaluators, and launching Tasks. Figure 8 evaluates this setup on jobs configured with various numbers of Evaluators and Tasks. The combined overhead is computed from $\frac{actual\ runtime}{ideal\ runtime} - 1$, where:

$$ideal\ runtime = \frac{\#\text{Tasks} * task\ execution\ time}{\#\text{Evaluators}}.$$

The figure shows that as we run more Tasks per Evaluator, we amortize the cost of communicating with YARN and launching Evaluators, and the overall job overhead decreases. This is consistent with the earlier synthetic measurements that suggest spawning tasks is orders of magnitudes faster than launching Evaluators. Since job parallelism is limited to the number of Evaluators, jobs with more Evaluators suffer higher overheads but finish faster.

Comparison with Apache Spark—Next, we leveraged 25 D4 Microsoft Azure instances (the third experimental setup) to run a similar experiment comparing to Apache Spark. Out of the total 200 cores available, we allocated 300 YARN containers, each with 1GB of available memory. In addition, each application master was allocated 4GB of RAM. The experiment begins by instantiating a task runtime (an Evaluator in the REEF case, and an Executor in the Spark case) on each container. The respective application masters then

begin to launch a series of tasks, up to a prescribed number. The combined overhead is computed as above.

Before reporting results for this experiment, we first describe the differences in the overheads for launching tasks. In Spark, launching a task requires transferring a serialized closure object with all of its library dependencies to the Spark Executor, which caches this information for running subsequent tasks of the same type i.e., stage. In REEF, library dependencies are transferred upfront i.e., when the Evaluator is launched. This highlights a key difference in the REEF design, which assumes complete visibility into what tasks will run on an Evaluator. Thus, the overhead cost of launching a Task in REEF boils down to the time it takes to package and transfer its Tang configuration.

Figure 9 reports on the overheads of REEF and Apache Spark for jobs that execute a fixed number of tasks configured to sleep for 100ms before exiting. The total running time is reported in Figure 9a and the percentage of time spent on overhead work (i.e., total running time normalized to ideal running time) is in Figure 9b. In all cases, the overhead in REEF is less than Spark. In both systems, the overheads diminish as the job size (i.e., number of tasks) increases. On the lower end of the job size spectrum, Spark overheads for transferring task information (e.g., serialized closure and library dependencies) are much more pronounced; larger jobs benefit from the caching this information on the Executor. At larger job sizes, both system overheads converge to about the same (percentage) amount.

Evaluator/Task allocation and launch time breakdown—Here we dive deeper into the time it takes to allocate resources from YARN, spawn Evaluators, and launching Tasks. Figure 10 shows these times (as a stacked graph) for a job that allocates 256 Evaluators. The red and green portions are very pessimistic estimates of the REEF overhead in starting an Evaluator on a Node Manager and launching a Task on a running Evaluator, respectively. The majority of the time is spent in container allocation (blue portion) i.e., the time from container request submission to the time the allocation response is received by the Driver; this further underscores the need to minimize such interactions with YARN by retaining Evaluators for recurring Task executions.

The time to launch an Evaluator on an allocated container is shown by the red portion, which varies between different Evaluators. YARN recognizes when a set of processes (from its perspective) share files (e.g., code libraries), and only copies such files once from the Application Master to the Node Manager. This induces higher launch times for the first wave of Evaluators. Later scheduled Evaluators launch faster, since the shared files are already on the Node Manager from earlier Evaluator executions; recall, we are scheduling 256 Evaluators on 35 Node Managers. Beyond that, starting a JVM and reporting back to the Driver adds about 1-2 seconds to the launch time for all Evaluators. The time to launch a Task (green portion) is fairly consistent (about 0.5 seconds) across all Evaluators.

5.2 Resource Elastic Machine Learning

In this section, we evaluate the elastic group communications based machine learning algorithm described in Section 4.2. The learning task is to learn a a linear logistic regression mode using a Batch Gradient Descent (BGD) optimizer. We use two datasets for the

experiments, both derived from the splice dataset described in [1]. The raw data consists of strings of length 141 with 4 alphabets (A, T, G and C).

Dataset A contains a subset of 4 million examples sampled from splice was used to derive binary features that denote the presence or absence of n-grams at specific locations of the string with $n = [1,4]$. The dimensionality of the feature space is 47,028. This dataset consists of 14GB of data.

Dataset B contains the entire dataset of 50 million examples was used to derive the first 100,000 features per the above process. This dataset consists of 254GB of data.

Algorithm—We implemented the BGD algorithm described in Section 4.2.1 on top of the elastic group communications Service described in Section 4.2.2. The Driver assigns a worker Task to cache and process each data partition. Each worker Task produces a gradient value that is reduced to a global gradient on the root Task using the Reduce operator. The root Task produces a new model that is Broadcast to the worker Tasks. The job executes in iterations until convergence is achieved.

Developing on REEF—REEF applications can be easily moved between Environment Adapters (3.1.1). We used this to first develop and debug BGD using the Local Process adapter. We then moved the application to YARN with only a single configuration change. Figure 11 shows the convergence rate of the algorithm running on Dataset A on the same hardware in these two modes: "Local" denotes a single cluster machine. In the YARN mode, 14 compute Tasks are launched to process the dataset. The first thing to note is that the algorithm performs similarly in both environments. That is, in each iteration, the algorithm makes equivalent progress towards convergence. The main difference between these two environments is in the start-up cost and the response time of each iteration. YARN suffers from a higher start-up cost due to the need to distribute the program, but makes up for this delay during execution, and converges about 14 minutes earlier than the local version. Considering the 14x increased hardware, this is a small speedup that suggests to execute the program on a single machine which REEF's Environment Adapters made it easy to discover.

Elastic BGD—Resource Managers typically allocate resources as they become available. Traditional MPI-style implementations wait for all resources to come online before they start computing. In this experiment, we leverage the Elastic Group Communications Service to start computing as soon as the first Evaluator is ready. We then add additional Evaluators to the computation as they become available. Figure 12 plots the progress in terms of the objective function measured on the full dataset over time for both elastic and non-elastic versions of the BGD job. The line labeled *Non-elastic BGD* waits for *all* Evaluators to come online before executing the first iteration of the learning algorithm. The line labeled *Elastic BGD* starts the execution as soon as the first Evaluator is ready, which occurs after the data partition is cached. New Evaluators are incorporated into the computation at iteration boundaries.

We executed these two strategies on an idle YARN cluster, which means that resource requests at the Resource Manager were immediately granted. Therefore, the time taken for

an Evaluator to become ready was in (1) the time to bootstrap it, and (2) the time to execute a Task that loaded and cached data in the root Context. As the Figure shows, the elastic approach is vastly preferable in an on-demand resource managed setting. In effect, elastic BGD (almost) finishes by the time the non-elastic version starts.

While this application-level elasticity is not always possible, it is often available in machine learning where each machine represents a partition of the data. Fewer partitions therefore represent a smaller sample of the data set. And models obtained on small samples of the data can provide good starting points [9] for subsequent iterations on the full data.

Algorithmic fault handling—We consider machine failure during the execution. We compare three variants: (1) No failure; (2) ignoring the failure and continuing with the remaining data and (3) our proposal: use a first-order Taylor approximation of the missing partitions' input until the partitions come back online. Figure 13 shows the objective function over iterations. Our method shows considerable improvement over the baselines. Surprisingly, we even do better than the no failure case. This can be explained by the fact that the use of the past gradient has similarities to adding a momentum term which is well-known to have a beneficial effect [32].

Scale-out—Figure 14 shows the iteration time for varying scale-up factors. It grows logarithmically as the the data scales linearly (each partition adds approximately 1GB of data). This is positive and expected, as our Reduce implementation uses a binary aggregation tree; doubling the Evaluator count adds a layer to the tree.

6. RELATED WORK

REEF provides a simple and efficient framework for building distributed systems on Resource Managers like YARN [46] and Mesos [15]. REEF replaces software components common across many system architectures [39, 33, 52, 5, 8, 16, 53] with a general framework for developing the specific semantics and mechanisms in a given system e.g., data-parallel operators, an explicit programming model, or domain-specific language (DSL). Moreover, REEF is designed to be extensible through its Service modules, offering applications with library solutions to common mechanisms e.g., group communication, data shuffle, or a more general RDD [52]-like abstraction, which could then be exposed to other higher-level programming models (e.g., MPI).

With its support for state caching and group communication, REEF greatly simplifies the implementation of iterative data processing models such as those found in GraphLab [23], Twister [12], Giraph [38], and VW [1]. REEF can also be leveraged to support stream processing systems such as Storm [25] and S4 [29] on managed resources, as demonstrated with Azure Streaming Analytics (Section 4.4). Finally, REEF has been designed to facilitate hand-over of data across frameworks, short-circuiting many of the HDFS-based communications and parsing overheads incurred by state-of-the-art systems.

The Twill project [42] and REEF both aim to simplify application development on top of resource managers. However, REEF and Twill go about this in different ways. Twill simplifies programming by exposing a developer abstraction based on Java Threads that

specifically targets YARN, and exposes an API to an external messaging service (e.g., Kafka [20]) for its control-plane support. On the other hand, REEF provides a set of common building blocks (e.g., job coordination, state passing, cluster membership) for building distributed applications, virtualizes the underlying Resource Manager layer, and has a custom built control-plane that scales with the allocated resources.

Slider [41] is a framework that makes it easy to deploy and manage long-running static applications in a YARN cluster. The focus is to adapt existing applications such as HBase and Accumulo [37] to run on YARN with little modification. Therefore, the goals of Slider and REEF are different.

Tez [33] is a project to develop a generic DAG processing framework with a reusable set of data processing primitives. The focus is to provide improved data processing capabilities for projects like Hive, Pig, and Cascading. In contrast, REEF provides a generic layer on which diverse computation models, like Tez, can be built.

7. SUMMARY AND FUTURE WORK

We embrace the industry-wide architectural shift towards decoupling resource management from higher-level applications stacks. In this paper, we propose a natural next step in this direction, and present REEF as a scale-out computing fabric for resource managed applications. We started by analyzing popular distributed data-processing systems, and in the process we isolated recurring themes, which seeded the design of REEF. We validated these design choices by building several applications, and hardened our implementation to support a commercial service in the Microsoft Azure Cloud.

REEF is an ongoing project and our next commitment is towards providing further building-blocks for data processing applications. Specifically, we are actively working on a checkpoint service for fault-tolerance, a bulk-data transfer implementation that can "shuffle" massive amounts of data, an improved low-latency group communication library, and an abstraction akin to RDDs [51], but agnostic to the higher-level programming model. Our intention with these efforts is to seed a community of developers that contribute further libraries (e.g., relational operators, machine learning toolkits, etc.) that integrate with one another on a common runtime. In support of this goal, we have set up REEF as an Apache Incubator project. Code and documentation can be found at http://reef.incubator.apache.org. The level of engagement both within Microsoft and from the research community reinforces our hunch that REEF addresses fundamental pain-points in distributed system development.

Acknowledgements

REFERENCES

1. Agarwal A, Chapelle O, Dudík M, Langford J. A reliable effective terascale linear learning system. CoRR. 2011 abs/1110.4198.

2. Ahmed A, Aly M, Gonzalez J, Narayanamurthy S, Smola AJ. Scalable inference in latent variable models. WSDM '12. 2012

3. Alvaro P, Conway N, Hellerstein J, Marczak WR. Consistency analysis in bloom: a calm and collected approach. CIDR. 2011:249–260.

4. Balakrishnan M, Malkhi D, Davis JD, Prabhakaran V, Wei M, Wobber T. Corfu: A distributed shared log. ACM Transactions on Computer Systems (TOCS). 2013; 31(4):10.

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

6. Beutel A, Weimer M, Narayanan V, Tom Minka YZ. Elastic distributed bayesian collaborative filtering. NIPS workshop on Distributed Machine Learning and Matrix Computations. 2014

7. Borkar V, Bu Y, Carey MJ, Rosen J, Polyzotis N, Condie T, Weimer M, Ramakrishnan R. Declarative systems for large-scale machine learning. TCDE. 2012; 35(2)

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

9. Bousquet O, Bottou L. The tradeoffs of large scale learning. Advances in Neural Information Processing Systems. 2007:161–168.

10. Chu C-T, Kim SK, Lin Y-A, Yu Y, Bradski GR, Ng AY, Olukotun K. Map-reduce for machine learning on multicore. Advances in Neural Information Processing Systems. 2006

11. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun. ACM. 2008; 51

12. Ekanayake J, Li H, Zhang B, Gunarathne T, Bae S-H, Qiu J, Fox G. Twister: a runtime for iterative mapreduce. HPDC. 2010

13. Google. Guice. https://github.com/google/guice

14. Gropp, W.; Huss-Lederman, S.; Lumsdaine, A.; Lusk, E.; Nitzberg, B.; Saphir, W.; Snir, M. MPI - The Complete Reference: Volume 2, The MPI-2 Extensions. MIT Press; Cambridge, MA, USA: 1998.

15. Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, AD.; Katz, R.; Shenker, S.; Stoica, I. NSDI. USENIX Association; 2011. Mesos: A platform for fine-grained resource sharing in the data center.; p. 22-22.

16. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: distributed data-parallel programs from sequential building blocks. Eurosys. 2007

17. Kavulya, S.; Tan, J.; Gandhi, R.; Narasimhan, P. An analysis of traces from a production mapreduce cluster.. Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10; Washington, DC, USA. IEEE Computer Society; 2010. p. 94-103.

18. Kearns M. Efficient noise-tolerant learning from statistical queries. J. ACM. 1998; 45(6):983–1006.

19. Kohler E, Morris R, Chen B, Jannotti J, Kaashoek MF. The click modular router. ACM Transactions on Computer Systems (TOCS). 2000; 18(3):263–297.

20. Kreps J, Narkhede N, Rao J. Kafka: A distributed messaging system for log processing. NetDB. 2011

21. Kumar A, Karampatziakis N, Mineiro P, Weimer M, Narayanan V. Distributed and scalable pca in the cloud. BigLearn NIPS Workshop. 2013

22. Li M, Andersen DG, Park JW, Smola AJ, Ahmed A, Josifovski V, Long J, Shekita EJ, Su B-Y. Scaling distributed machine learning with the parameter server. Proc. OSDI. 2014:583–598.

23. Low, Y.; Gonzalez, J.; Kyrola, A.; Bickson, D.; Guestrin, C.; Hellerstein, JM. GraphLab: A New Parallel Framework for Machine Learning.. Conference on Uncertainty in Artificial Intelligence (UAI); Catalina Island, California. 2010.

24. Malewicz, G.; Austern, MH.; Bik, AJ.; Dehnert, JC.; Horn, I.; Leiser, N.; Czajkowski, G. Pregel: a system for large-scale graph processing.. Proceedings of the ACM SIGMOD International Conference on Management of data, SIGMOD '10; New York, NY, USA. ACM; 2010. p. 135-146.

25. Marz, N. Storm: Distributed and fault-tolerant realtime computation. http://storm.apache.org

26. Meijer E. Your mouse is a database. Commun. ACM. 2012; 55(5):66–73.

27. Murray, DG.; McSherry, F.; Isaacs, R.; Isard, M.; Barham, P.; Abadi, M. Naiad: A timely dataflow system.. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13; New York, NY, USA. ACM; 2013. p. 439-455.

28. Narayanamurthy S, Weimer M, Mahajan D, Condie T, Sellamanickam S, Keerthi SS. Towards resource-elastic machine learning. BigLearn NIPS Workshop. 2013

29. Neumeyer L, Robbins B, Nair A, Kesari A. S4: Distributed stream computing platform. ICDMW. 2010

30. Olston, C.; Reed, B.; Srivastava, U.; Kumar, R.; Tomkins, A. Pig latin: a not-so-foreign language for data processing.. Proceedings of the ACM SIGMOD international conference on Management of data, SIGMOD '08; New York, NY, USA. ACM; 2008. p. 1099-1110.

31. Rabkin, A. Ph.D. Dissertation. UC Berkeley: 2012. Using program analysis to reduce misconfiguration in open source systems software..

32. Rumelhart DE, Hinton GE, Williams RJ. Learning internal representations by error propagation. Technical report, DTIC Document. 1985

33. Saha B, Shah H, Seth S, Vijayaraghavan G, Murthy A, Curino C. Apache tez: A unifying framework for modeling and building data processing applications. SIGMOD. 2015; 2015

34. Schwarzkopf M, Konwinski A, Abd-El-Malek M, Wilkes J. Omega: flexible, scalable schedulers for large compute clusters. EuroSys. 2013:351–364.

35. Shapiro M, Preguiça NM. Designing a commutative replicated data type. CoRR. 2007 abs/0710.1784.

36. Stonebraker, M.; Cetintemel, U. One size fits all: An idea whose time has come and gone.. Proceedings of the 21st International Conference on Data Engineering, ICDE '05; Washington, DC, USA. IEEE Computer Society; 2005. p. 2-11.

37. The Apache Software Foundation. Apache Accumulo. http://accumulo.apache.org/

38. The Apache Software Foundation. Apache Giraph. http://giraph.apache.org/

39. The Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org

40. The Apache Software Foundation. Apache Mahout. http://mahout.apache.org

41. The Apache Software Foundation. Apache Slider. http://slider.incubator.apache.org/

42. The Apache Software Foundation. Apache Twill. http://twill.incubator.apache.org/

43. The Netty project. Netty. http://netty.io

44. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R. Hive – a warehousing solution over a map-reduce framework. PVLDB. 2009

45. Valiant LG. A bridging model for parallel computation. Commun. ACM. 1990; 33(8):103–111.

46. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E. Apache hadoop yarn: Yet another resource negotiator. SOCC. 2013

47. Weimer M, Rao S, Zinkevich M. A convenient framework for efficient parallel multipass algorithms. LCCC. 2010

48. Welsh, M. What I wish systems researchers would work on. http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html

49. Welsh, M.; Culler, D.; Brewer, E. SIGOPS. Vol. 35. ACM; 2001. Seda: an architecture for well-conditioned, scalable internet services.; p. 230-243.

50. Ye, J.; Chow, J-H.; Chen, J.; Zheng, Z. Stochastic gradient boosted distributed decision trees.. Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM '09; New York, NY, USA. ACM; 2009. p. 2061-2064.

51. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin M, Shenker S, Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI. 2012

52. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. HotCloud. 2010

53. Zhou J, Bruno N, Wu M-C, Larson P-A, Chaiken R, Shakib D. Scope: Parallel databases meet mapreduce. VLDB Journal. 2012; 21(5)
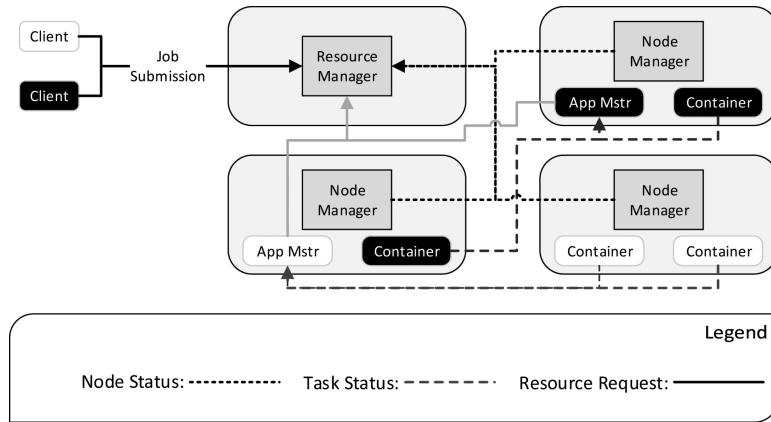
Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

Figure 1.

Example YARN Architecture showing two clients submitting jobs to the Resource Manager (RM), which launches two client Application Master (AM) instances on two Node Managers (NM). Each AM requests containers from the RM, which allocates the containers based on available resources reported from its NMs. If supported by the application, tasks running on containers report status to the respective AM.

Component (abbr.)	Description
Resource Manager (RM)	A service that leases cluster resources to applications.
Node Manager (NM)	Manages the resources of a single compute entity (e.g., machine). Reports the status of managed machine resources to the RM
Application Master (AM)	Implements the application control-flow and resource allocation logic.
Container	A single unit of resource allocation e.g., some amount of CPU/RAM/Disk.

Figure 2.
Glossary of components (and abbreviations) described in this section, and used throughout the paper.
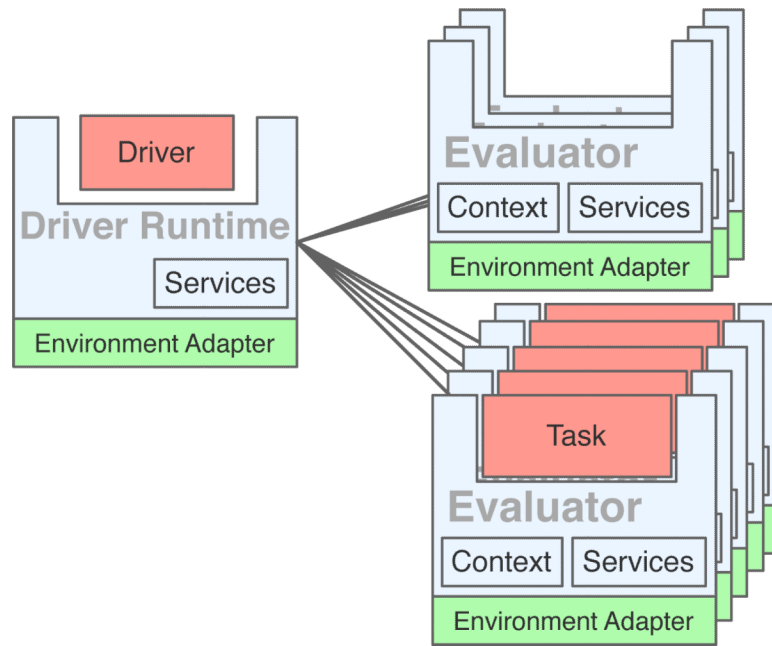
Figure 3.
An instance of REEF in terms of its application framework (Driver and Task) and runtime infrastructure components (Evaluator, Driver Runtime, Environment Adapter).
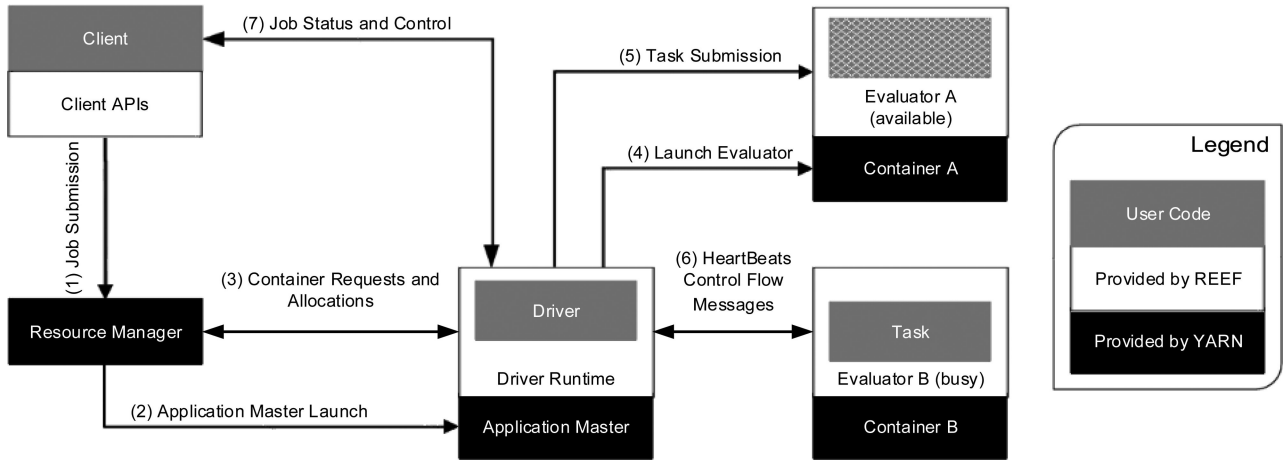
Figure 4.

High-level REEF control-flow diagram—running within an example YARN environment—that captures an application with two Evaluator instances, one of which is running a Task. Each control channel is labeled with a number and description of the interaction that occurs between the two entities.

	C#	Java	CPP	Total
Tang	10,567	6,976	0	17,543
Wake	7,749	4,681	0	12,430
REEF	13,136	15,118	1,854	30,108
Services	0	5,319	0	5,319
Total	31,452	32,094	1854	65,400

Figure 5.
Lines of code by component and language
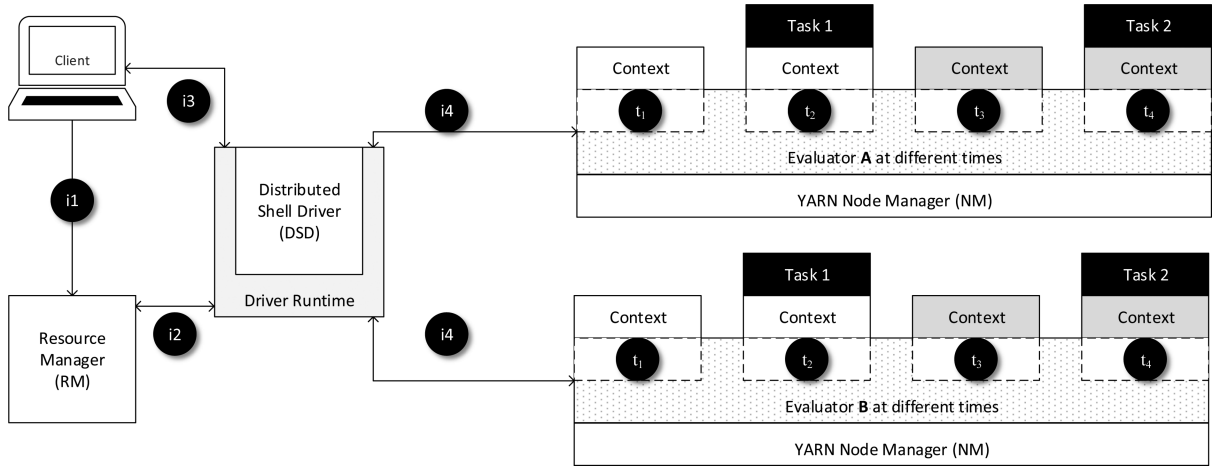
Figure 6.
A Client executing the distributed shell job on two Evaluators **A** and **B**. The Evaluators execute shell commands—submitted by the Client—in Task 1 and Task 2 at time instances t_2 and t_4.

	Wake Event	REEF Task	YARN Container
Time(ns)	1	30,000	1.2E7

Figure 7.
Startup times for core REEF primitives
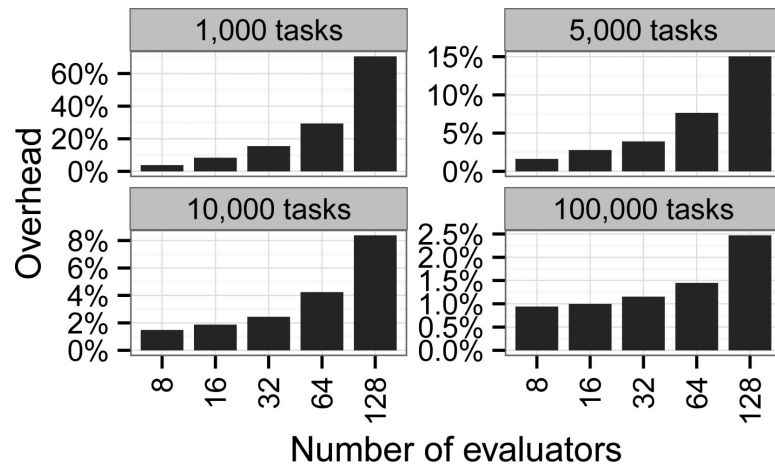
Figure 8.
Combined (REEF + YARN) overheads for jobs with short-lived (1 second) tasks
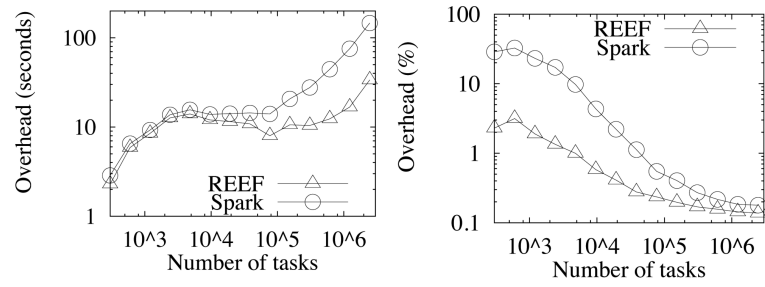
(a) Absolute running time (y-axis) of jobs with varying numbers of tasks (x-axis).

(b) Computed overheads (y-axis) of jobs with varying numbers of tasks (x-axis).

Figure 9.

Overheads of REEF and Apache Spark for jobs with short-lived (100ms) tasks.
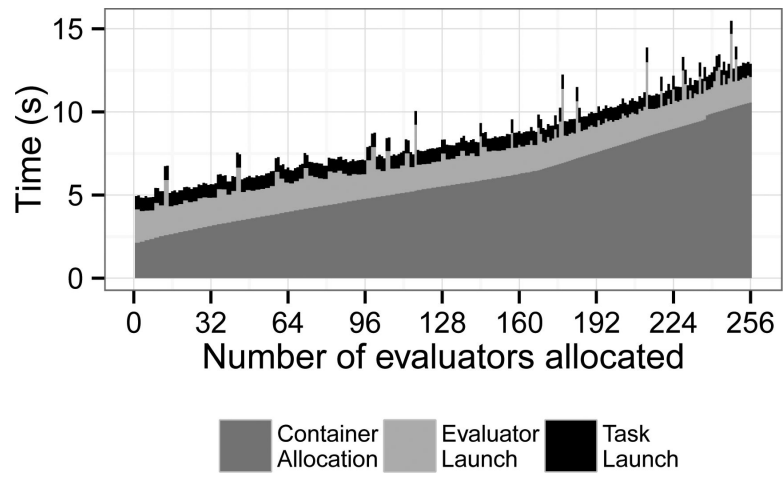
Figure 10.
Evaluator/Task allocation and launch time breakdown
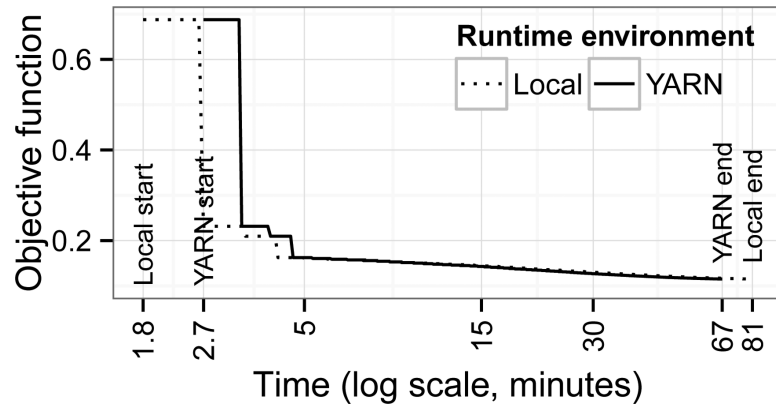
Figure 11.
Objective function over time for Dataset A when executing locally and on a YARN cluster
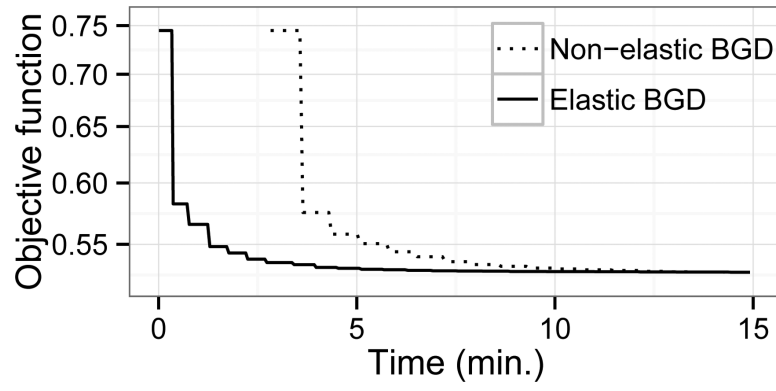
Figure 12.
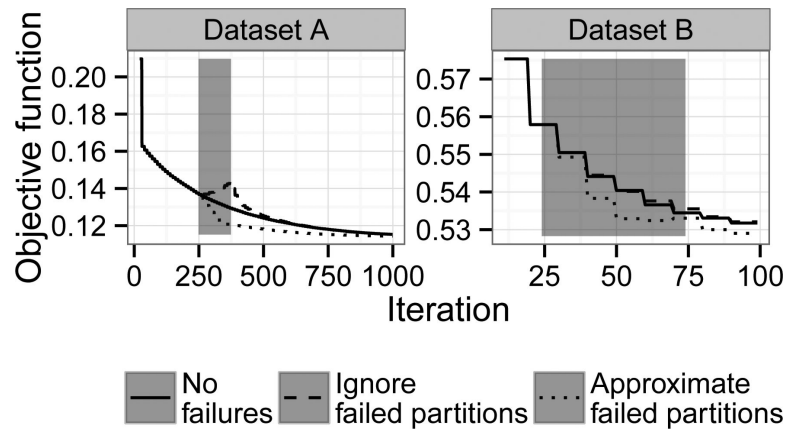Ramp-up experiment on Dataset B

Figure 13.
Learning progress over iterations with faulty partitions. Grey areas between iterations 250..375 for Dataset A and 25..75 for Dataset B indicate the period of induced failure.
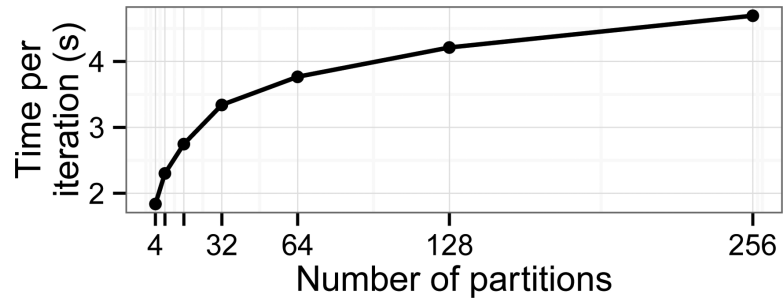
Figure 14.
Scale-out iteration time with partitions of 1GB.