Apache REEF: Retainable Evaluator Execution Framework

5

BYUNG-GON CHUN, Seoul National University
TYSON CONDIE, University of California at Los Angeles
YINGDA CHEN, Alibaba
BRIAN CHO, Facebook
ANDREW CHUNG, Carnegie Mellon University
CARLO CURINO and CHRIS DOUGLAS, Microsoft
MATTEO INTERLANDI, University of California at Los Angeles
BEOMYEOL JEON, University of Illinois at Urbana-Champaign
JOO SEONG JEONG, GYEWON LEE, and YUNSEONG LEE, Seoul National University
TONY MAJESTRO, Microsoft
DAHLIA MALKHI, VMware
SERGIY MATUSEVYCH, Microsoft
BRANDON MYERS, University of Iowa
MARIIA MYKHAILOVA and SHRAVAN NARAYANAMURTHY, Microsoft
JOSEPH NOOR, University of California at Los Angeles
RAGHU RAMAKRISHNAN and SRIRAM RAO, Microsoft
RUSSELL SEARS, Pure Storage
BEYSIM SEZGIN, Microsoft
TAEGEON UM, Seoul National University
JULIA WANG and MARKUS WEIMER, Microsoft
YOUNGSEOK YANG, Seoul National University

This manuscript is an extension of the conference version appearing in the Proceedings of the 36th ACM SIGMOD 2015 [54]. We present a more detailed description of the REEF architecture and design. We add the high-availability support of the REEF Driver and its evaluation. We also add two new REEF use cases: a distributed cache and a distributed machine-learning framework.

At Seoul National University, this work was supported by the MSIT (Ministry of Science and ICT), Korea, under the SW Starlab support program (IITP-2017-R0126-17-1093) supervised by the IITP (Institute for Information & Communications Technology Promotion). Additionally, this work was supported at UCLA through grants NSF IIS-1302698 and CNS-1351047, and U54EB020404 awarded by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) through funds provided by the trans-NIH Big Data to Knowledge (BD2K) initiative (www.bd2k.nih.gov).

Authors' addresses: B.-G. Chun (corresponding author), J. Jeong, G. Lee, Y. Lee, T. Um, and Y. Yang, Computer Science and Engineering Department, Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul 08826, South Korea; emails: bgchun, joosjeong, gyewonlee, yunseong, taegeonum, yyang@snu.ac.kr; T. Condie, M. Interlandi, and J. Noor, Computer Science Department, University of California, Los Angeles, 4732 Boelter Hall, Los Angeles, CA 90095, USA; emails: tcondie@cs.ucla.edu, m.interlandi@gmail.com, jnoor@cs.ucla.edu; C. Curino, C. Douglas, T. Majestro, S. Matusevych, M. Mykhailova, S. Narayanamurthy, R. Ramakrishnan, S. Rao, B. Sezgin, J. Wang, and M. Weimer, Microsoft, 1 Microsoft Way, Redmond, WA 98052, USA; emails: ccurino, cdoug, tmajest, sergiym, mamykhai, shravan, raghu, sriramra, beysims, Qiuhe.Wang, mweimer@microsoft.com; Y. Chen, Alibaba, 1511 6th Ave, Seattle, WA 98101, USA; email: ydchen@gmail.com; B. Cho, Facebook, 1 Hacker Way, Menlo Park, CA 94025, USA; email: bcho@fb.com; A. Chung, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA; email: afchung@andrew.cmu.edu; B. Jeon, Department of Computer Science, University of Illinois at Urbana-Champaign, Thomas M. Siebel Center, 201 North Goodwin Ave, Urbana, IL 61801, USA; email: bj2@illinois.edu; D. Malkhi, VMWare, 3425 Hillview Ave, Palo Alto, CA 94304, USA; email: dahliamalkhi@gmail.com; B. Myers, Computer Science Department, The University of Iowa, 14 MacLean Hall, Iowa City, IA 52242, USA; email: brandon-d-myers@uiowa.edu; R. Sears, Pure Storage, 650 Castro St #400, Mountain View, CA 94041, USA; email: sears@purestorage.com.

ACM 0734-2071/2017/10-ART5 $15.00
https://doi.org/10.1145/3132037

Resource Managers like YARN and Mesos 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 reimplement common mechanisms (e.g., caching, bulk-data transfers). This article 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 reuse for data caching and state management abstractions that greatly ease the development of elastic data processing pipelines on cloud platforms that support a Resource Manager service. We illustrate the power of REEF by showing applications built atop: a distributed shell application, a machine-learning framework, a distributed in-memory caching system, and a port of the CORFU system. REEF is currently an Apache top-level project that has attracted contributors from several institutions and it is being used to develop several commercial offerings such as the Azure Stream Analytics service.

CCS Concepts: • **Computer systems organization → Distributed architectures**; • **Software and its engineering → Distributed systems organizing principles**; • **Information systems → Data management systems**;

Additional Key Words and Phrases: Data processing, resource management

ACM Reference format:
Byung-Gon Chun, Tyson Condie, Yingda Chen, Brian Cho, Andrew Chung, Carlo Curino, Chris Douglas, Matteo Interlandi, Beomyeol Jeon, Joo Seong Jeong, Gyewon Lee, Yunseong Lee, Tony Majestro, Dahlia Malkhi, Sergiy Matusevych, Brandon Myers, Mariia Mykhailova, Shravan Narayanamurthy, Joseph Noor, Raghu Ramakrishnan, Sriram Rao, Russell Sears, Beysim Sezgin, Taegeon Um, Julia Wang, Markus Weimer, and Youngseok Yang. 2017. Apache REEF: Retainable Evaluator Execution Framework. *ACM Trans. Comput. Syst.* 35, 2, Article 5 (October 2017), 31 pages.
https://doi.org/10.1145/3132037

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 [51] and Pig [35] 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 [53], Apache Mesos [19], and Google Omega [39] 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 [47] explicitly warns about the slow performance of some of its algorithms on Hadoop MapReduce.

Resource Managers are a first step in refactoring 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 computational status and progress, and dynamic parameters—is left to the application programmer to implement.

This article presents Apache REEF (Retainable Evaluator Execution Framework), which provides runtime management support for computational 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 his or her logic in a `Driver`—the 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 reusable control plane that enables systematic reuse of resources and retention of state across tasks, possibly from different types of computations, by filling the gap between Resource Managers and applications. 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 dataflow and retaining state across iterations in iterative or recursive distributed programs. REEF is an (open-source) Apache top-level project with many contributions of artifacts that greatly reduce the development effort in building analytical toolkits on Resource Managers.

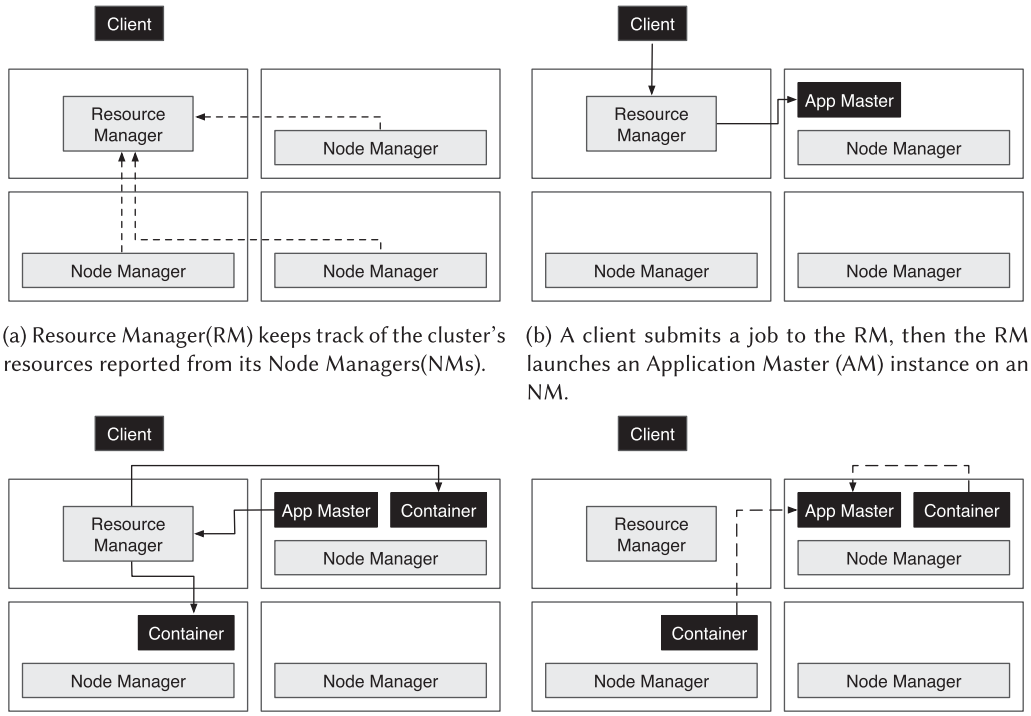
The remainder of this article is organized as follows. Section 2 provides background on Resource Manager architectures. Section 3 gives a general overview of the REEF abstractions and key design decisions. Section 4 describes some of the applications developed using REEF, one being the Azure Stream Analytics 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 article 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 exhibit extremely poor utilization (on the order of 5%–10% CPU utilization at Yahoo! [21]) due to resource allocations being too coarse grained.[1] Second, the MapReduce programming model is not an ideal fit for certain applications. 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) [1, 2, 45].

These issues motivated the design of a second-generation Hadoop system, which included an explicit resource management layer called YARN [53].[2] Additional examples of Resource Managers include Google Omega [39], Apache Mesos [19], and Kubernetes [5]. 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 article. Comparing the merits of different resource management layers is out of scope for this article, because

[1]Hadoop MapReduce tasks are often either CPU or I/O bound, and slots represent a fixed ratio of CPU and memory resources.
[2]YARN: Yet Another Resource Negotiator.



(a) Resource Manager(RM) keeps track of the cluster's resources reported from its Node Managers(NMs).

(b) A client submits a job to the RM, then the RM launches an Application Master (AM) instance on an NM.

(c) When the AM requests containers to the RM, then containers are allocated by the RM in the available nodes.

(d) If supported by the application, tasks running on containers report status to the AM.

Fig. 1. Illustration of an application running on YARN.

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)	Handles the application control flow and resource negotiation with the RM.
Container	A single unit of resource allocation, e.g., some amount of CPU/RAM/Disk.

Fig. 2. Glossary of components (and abbreviations) described in this section and used throughout the article.

REEF is primarily relevant to what happens with allocated resources, and not how resources are requested.

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 provide a basis for monitoring the overall cluster health via the RM, which notifies relevant applications when failures occur.

A YARN application is represented by an Application Master (AM), which is responsible for orchestrating the job's work on allocated *containers,* that is, a slice of machine resources (some amount of CPU, RAM, disk, etc.). A client submits an AM package—which includes a shell command and any files (i.e., binary executables, configurations) 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 ask the NMs that host its containers to launch a desired program. Returning to Figure 1, we see an AM instance running with two allocated containers, each of which executes a job-specific task.

2.1 Example: Distributed Shell on YARN

To further set the stage, we briefly explain how to write a distributed shell application directly on YARN, that is, without REEF. The YARN source code distribution 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 containing relevant files required by the executable (e.g., shell script), the environment to be set up for the executable, and a command line to execute. The AM then 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 (around 1,300 lines of code) as an exercise for interacting with its protocols. A more complete distributed shell application might include the following features:

- Providing 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. In Section 4.1, we will describe a more feature-complete version of this example developed on REEF in less than half (530) the lines of code. The key contributor to this lines-of-code reduction happens by capturing the control-flow code, common to Resource Manager applications, in the REEF framework.

3 REEF ARCHITECTURE

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 [7, 10, 15, 20, 60, 61] for common mechanisms and design patterns, leading to the following common components within these architectures:

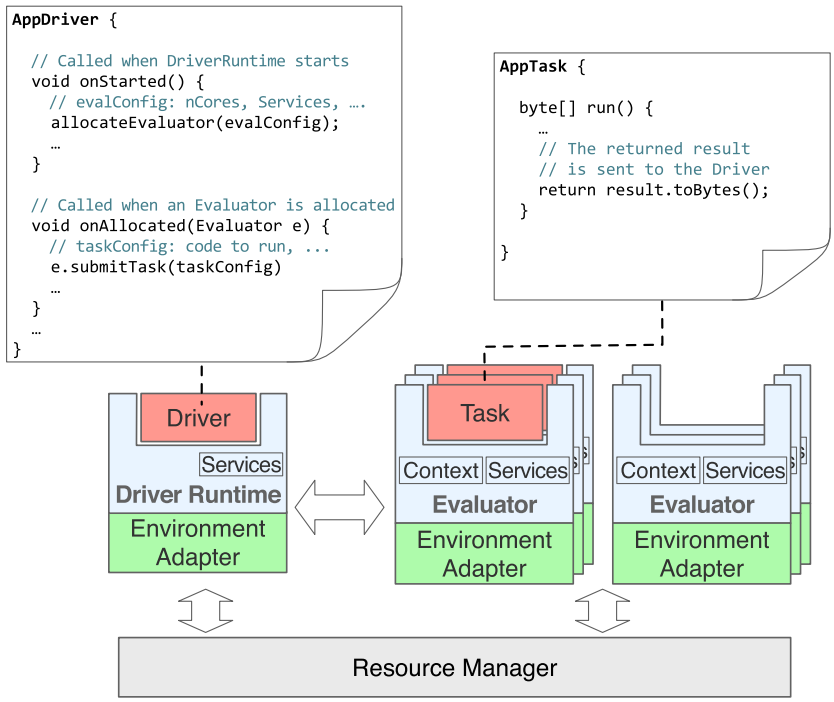


Fig. 3. An instance of REEF in terms of its application framework (`Driver` and `Task`) and runtime infrastructure components (`Evaluator`, `Driver Runtime`, `Environment Adapter`).

- A centralized per-job scheduler that observes the runtime state and assigns tasks to resources, for example, MapReduce task slots [15]
- A runtime for executing compute tasks and retaining state in an organized fashion, that is, contexts that group related object states
- Communication channels for monitoring status and sending control messages
- Configuration management for passing parameters and binding application interfaces to runtime implementations

Apache REEF captures these features in a framework that allows application-level logic to focus on appropriate implementations of higher-level semantics, such as 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 provides the following key abstractions to developers:

- `Driver`: application code that implements the resource allocation and `Task` scheduling logic
- `Task`: the work to be executed in an `Evaluator`
- `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`

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 is a shim layer that insulates the REEF runtime from the underlying Resource Manager layer. Lastly, REEF provides messaging channels between the Driver and Task instances—supported by a highly optimized event management toolkit (Section 3.4.1)—for communicating runtime status and state, and a configuration management tool (Section 3.4.2) 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).

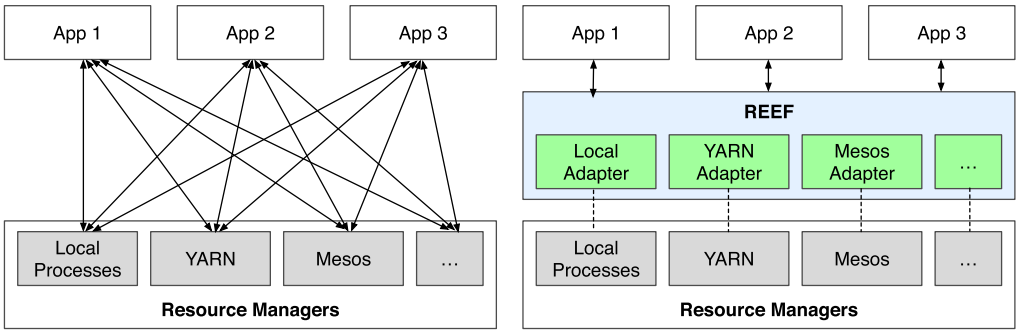
3.1 Runtime Infrastructure

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 (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 new mechanisms to be developed and incorporated into an application's logic. This section further describes these runtime infrastructure components.

3.1.1 Evaluators. REEF retains resource containers across tasks to avoid resource allocation and scheduling costs. An Evaluator is the abstraction to capture retainable containers and is the runtime for Tasks. There is a 1:1 mapping between Evaluators and underlying resource containers. An Evaluator runs one Task at a time, but it may run many Tasks throughout its lifetime.

3.1.2 Contexts. Retaining state across task executions is central to the REEF design and critical to supporting iterative dataflows that cache loop-invariant data or to facilitate delta-based computations (e.g., Naiad [32] and BigDatalog [41]). Moreover, the need to clean up state from prior task executions prompted the design of stackable contexts in the 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, potentially crossing 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, storing 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, 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. Contexts add such fine-grained (task-level) mutable state management, which could be leveraged for building a DAG scheduler (like Dryad [20], Tez [38], Hyracks [10]), where vertices (computational stages) are given a "localized" context for scratch space, using the "root" context for passing state.

3.1.3 Services. The central design principle of REEF is in factoring out core functionalities that can be reused 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



(a) Without REEF, developers should write boiler-plate code multiple times in order to run their applications on different resource managers.

(b) REEF's environment adapters allow developers to run their applications on different resource managers with simple configuration changes.

Fig. 4. Integration of applications to various Resource Managers without and with REEF.

developers to be agnostic about re-establishing communication with a `Task` that was respawned on a separate `Evaluator`.

3.1.4 Environment Adapter. REEF factors out many of the Resource-Manager-specific details into an `Environment Adapter` layer (Figure 4), 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 `Evaluators`) to the underlying Resource Manager protocol. We have implemented three such adapters:

- **Local Processes:** REEF has its own runtime that leverages the host operating system to provide process isolation between the `Driver` and `Evaluators`. The runtime limits the number of processes active on a single node 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.
- **Apache YARN:** This adapter executes the `Driver Runtime` as a YARN Application Master [53]. Resource requests are translated into the appropriate YARN protocol, and YARN containers are used to host `Evaluators`. The adapter translates `Driver` resource requests into requests for YARN containers. When notified of successfully allocated containers, the adapter exposes them to the `Driver` as allocation events. In response to a `Context` or `Task` launch request from the `Driver`, the adapter configures and launches an `Evaluator` on the appropriate YARN container. Further, the adapter manages the status reports of all running YARN containers and notifies the `Driver` of respective `Evaluator` status reports through events such as allocation, failure, and completion.
- **Apache Mesos:** This adapter executes the `Driver Runtime` as a "framework" in Apache Mesos [19]. Resource requests are translated into the appropriate Mesos protocol, and Mesos executors are used to host `Evaluators`. The adapter locally starts a new process to host the `Driver Runtime`, passing user-configured parameters including the IP address of the Mesos Master. Once the `Driver Runtime` has been bootstrapped, the adapter registers itself as a Mesos Framework to start receiving resource offers. The adapter uses `Driver` resource requests to obtain resources during Mesos offer exchanges, which are allocated in the form of Mesos Executors (i.e., the equivalent of YARN containers). When the Mesos Master notifies it of successfully launched Mesos Executors, the adapter exposes them to the `Driver` as allocation events. The `Driver` can proceed to launch a `Context` or `Task`, which the adapter uses

to configure and launch Evaluator instances that run on Mesos Executors. As in YARN, the adapter receives status reports of all running Mesos Executors and notifies the Driver of respective Evaluator status reports (e.g., failure and completion).

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).

3.2 Application Framework

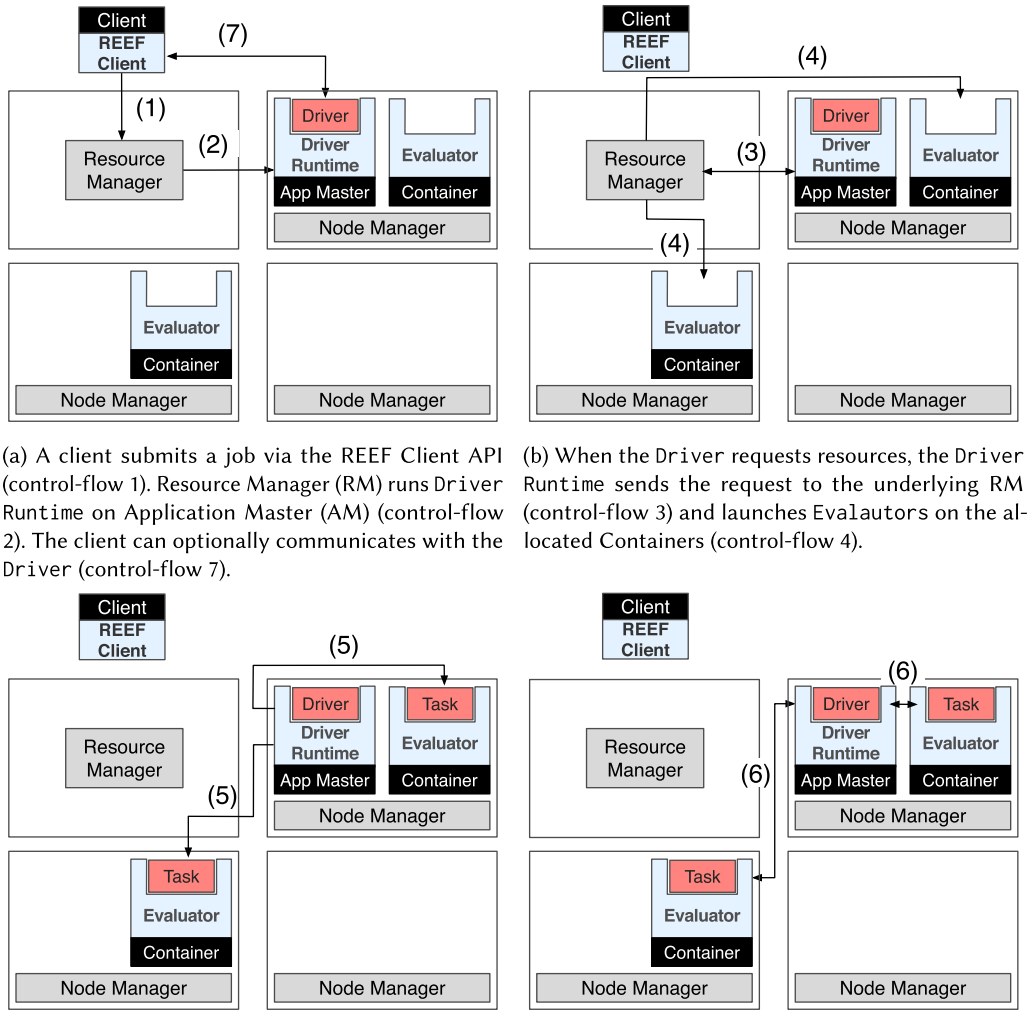
We now describe the framework used to capture application logic (i.e., the code written by the application developer). Figure 5 presents a high-level control-flow diagram of a REEF application. The control channels are labeled with a number. We will refer to this figure in our discussion by referencing 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 (Section 3.4.2), which requires bindings for the Driver implementation. When the Resource Manager launches the Application Master (**control-flow 2**), the REEF Driver Runtime will start and use the Tang specification to instantiate the Driver components. The Driver can optionally be given a channel to the client (**control-flow 7**) for communicating status and receiving commands via an interactive application.

3.2.1 Driver. The Driver implements event handlers that define the resource acquisition policies and (task-level) work scheduling of the application. For instance, a Driver that schedules a DAG of data processing elements—common to many data-parallel runtimes [7, 10, 20, 46, 60, 61]—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,[3] 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 a Reactive Extensions (Rx) [31] API[4] 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, a start event is passed to the Driver, which it must react to by either requesting resources (**control-flow 3**)—using a REEF-provided request module that mimics the underlying resource management protocol—or 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 to the Driver. The Evaluator allocation event API contains methods for configuring the initial Context state (e.g., files, services, object state, etc.), launching the

[3]Today, exceptions include Tez [38] and Spark [60].
[4]Supported by Wake, which we describe in Section 3.4.1.



(a) A client submits a job via the REEF Client API (control-flow 1). Resource Manager (RM) runs `Driver Runtime` on Application Master (AM) (control-flow 2). The client can optionally communicates with the `Driver` (control-flow 7).

(b) When the `Driver` requests resources, the `Driver Runtime` sends the request to the underlying RM (control-flow 3) and launches `Evalautors` on the allocated Containers (control-flow 4).

(c) The `Driver` submits `Tasks` to the running `Evaluators` (control-flow 5).

(d) The `Evaluators` and `Tasks` communicate with the `Driver` (control-flow 6).

Fig. 5. High-level REEF control-flow diagrams—running within an example YARN environment—that capture an application with two `Evaluator` instances, each of which is running a `Task`.

`Evaluator` on the assigned resource (via **control-flow 4**), and releasing it (deallocating) 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 triggered 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 3.1.2) and launch a sequence of `Task` executions (via **control-flow 5**). A failure of the `Evaluator` is assumed not to be recoverable (e.g., due to misconfiguration or hardware faults), and thus the relevant resource is automatically deallocated. A *failure* event containing the exception state is passed to the `Driver`.

On the other hand, Task events 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 including 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 four distinct events: launch, message, failed, and completion. The *launch* event API contains methods for terminating or suspending the Task execution and a method for sending opaque byte array messages to the running Task (via **control-flow 6**). Messages sent by the Driver are immediately pushed to the relevant Task to minimize latency. Task implementations are also able to send messages (opaque byte arrays) back to the Driver, which are piggy-backed on the 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. 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 do not result in the deallocation of the Evaluator hosting the failed Task.

3.2.2 Task. A Task is a piece of application code that contains a main method to 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 to be delivered to the appropriate Task. REEF does not wait for the next Evaluator "heartbeat" interval to transfer and deliver messages.

3.3 Driver High Availability

REEF handles Evaluator and Task failures by incorporating recovery logic at Driver. However, to make the entire application highly available, we also need to handle Driver failure events, which has not been tackled in other systems. To support long-running applications, REEF provides Driver high availability (HA) such that the crash of the Driver process does not cause the entire application to fail.

We currently support Driver HA only on the REEF YARN runtime by taking advantage of YARN's Resource Manager High Availability (RMHA) feature.[5] When RMHA is enabled, applications can preserve containers across application retries such that the failure of a YARN AM *only* results in a resubmission of an AM by the RM. An AM crash does not destroy the containers

[5]Note that Driver HA can only be supported if the underlying Resource Manager supports preservation of containers on a Driver failure and allows a new instance of the Driver to be associated with the containers requested by the previous Driver.

associated with the AM right away, and the outstanding containers are preserved for a certain duration. However, YARN does not reassociate the containers with the newly submitted AM. With REEF `Driver` HA, the `Evaluators` running on the containers call back to the new AM hosting `Driver`, and thus `Driver` is able to reassociate the `Evaluators`. REEF's `Driver` HA feature allows application developers to focus on their core application logic by dealing with these low-level details.

3.3.1 Driver-Side Design. The `Driver` keeps track of the `Evaluator` recovery process via a finite state machine. The recovery process always begins in the NOT_RESTARTED state. The `Driver` starts the recovery process by determining whether it is a resubmitted instance through information provided by the runtime environment. In YARN, we check resubmission by parsing the container ID of the AM container in which the `Driver` is running. If the `Driver` is a resubmitted instance, the process enters the RESTART_BEGAN state. Previous `Evaluators` and their statuses are then queried from the RM. The application is notified if any `Evaluator` has failed during the restart process, and the process enters the RESTART_IN_PROGRESS state as the `Driver` waits for evaluators to report back.

In anticipation of `Evaluators` reporting back, the `Driver` also keeps soft-state for all its expected `Evaluators`. An expected `Evaluator` starts out in the EXPECTED state. When the `Driver` receives a recovery heartbeat from the said `Evaluator`, the `Evaluator` moves to the REPORTED state. The `Evaluator` is subsequently added to the set of managed `Evaluators` in the `Driver` and moves to the REREGISTERED state. The handlers for `Evaluator` context and task are then invoked, and the `Evaluator` finally moves to the PROCESSED state. The `Driver` only transitions its state to the final RESTART_COMPLETED state if it expects no more `Evaluators` to report back or if a configurable recovery timeout has passed. Once the recovery timeout has been reached, all `Evaluators` that are still in the EXPECTED state are marked as EXPIRED and will be ignored and shut down if they report back to the `Driver` afterward. The reports of such expired `Evaluators` are entirely transparent to the application.

In order for the `Driver` to inform the client of which `Evaluator` failed on restart, it would need to keep external state on the `Evaluators`. The `Driver` performs this bookkeeping by recording the ID of the `Evaluator` allocated by the RM to the `Driver`. The `Driver` removes the `Evaluator` ID when an evaluator is released. The current implementation utilizes the cluster distributed file system (DFS) to perform `Evaluator` ID tracking.

3.3.2 Evaluator-Side Design. `Evaluators` periodically send heartbeats back to the `Driver`. In the event of a `Driver` failure, such heartbeats will fail. After passing a threshold of heartbeat failures, the `Evaluator` will enter a *recovery* mode where it assumes that the `Driver` has failed. Under the recovery mode, the `Evaluator` will try to perform an HTTP call routed by the YARN RM to a well-known endpoint set up by its associated `Driver`. This endpoint would provide the remote identification of the running `Driver` instance. In the YARN runtime, this endpoint can be derived from the RM host port and the application ID of the YARN application. Once the `Driver` has successfully been restarted, the endpoint will become available, and the `Evaluators` will be able to recover the remote identification of the new `Driver` instance and re-establish its heartbeats. As an alternative to the `Driver` setting up a special endpoint for HA, we are also looking into utilizing the YARN Service Registry to recover the remote identification of the `Driver`.

3.4 Low-Level Building Blocks

In building REEF, we factored out two low-level building blocks: event handling and configuration management. These blocks simplified developing key REEF abstractions.

3.4.1 Event Handling. We built an asynchronous event processing framework called Wake, which is based on ideas from SEDA [57], Rx [31], and the Click modular router [23]. As we describe in Section 3.2.1, the `Driver` interface is composed of handlers that contain application code that reacts to events. Wake allows the `Driver Runtime` to trade off between cooperative *thread sharing*, which 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 [50]. 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 4.5). 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`.

3.4.2 Configuration Management. Configuring distributed applications is well known to be a difficult and error-prone task [36, 56]. 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 [17] 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, and more modular code that is easier to unit test. However, it alone did not solve the problem of misconfigurations, which often occurred when instantiating application `Driver`, `Context`, or `Task` implementations on remote container resources that are 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, Tang will catch misconfiguration 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 and autocompletion of configuration parameters, and to detect a wide range of configuration problems (e.g., type checking, missing parameters). 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) [40] 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.apache.org/tang.html).

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

	C#	Java	C++	Total
Tang	17,740	12,770	0	30,510
Wake	6,125	8,248	0	14,373
REEF	26,326	38,976	3041	68,343
Services	16,428	20,961	0	37,389
Total	66,619	80,955	3,041	150,615

Fig. 6. Lines of code by component and language.

3.5 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.

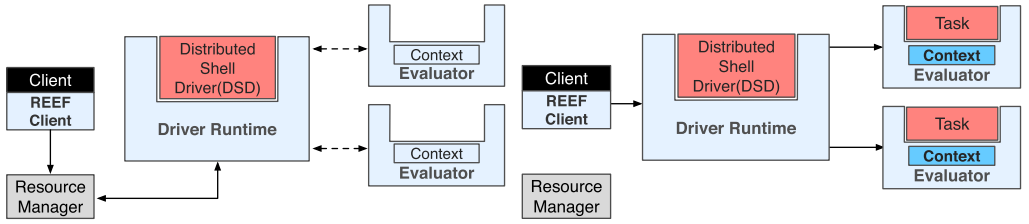
Applications are free to mix and match `Driver` side event handlers in Java and C# with any number of Java and C# `Evaluators`. To establish communications between Java and C# processes, Wake is implemented in both languages. Tang is also implemented in both languages and supports configuration validation 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 application. 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 6 gives an overview of the effort involved in the development of REEF, including its cross-language support.[7]

3.6 Discussion

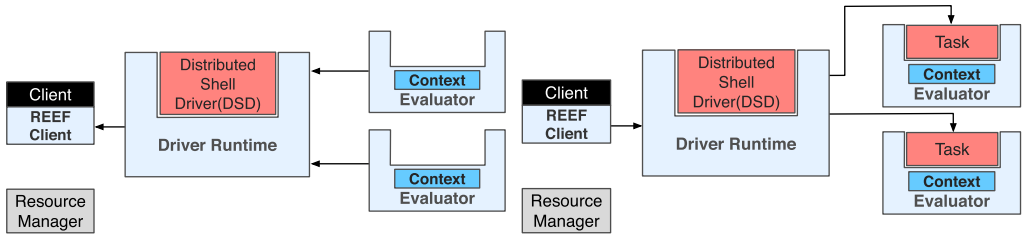
REEF is an active open-source project that started in late 2012. Over the past several years, we have refined our design based on feedback from many communities. The initial prototype of REEF's application interface was based on the Java Concurrency Library. When the `Driver` made a request for containers, 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. For example, exceptions thrown in arbitrary code interrupted the control flow in a manner that was not always obvious as opposed to 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. This 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.

[7]We computed lines of code on Apache REEF release 0.15.0.



(a) A REEF Client submits a job to start Distributed Shell Driver.

(b) When the Client sends a command through an HTTP request to the `Driver`, `Tasks` are launched on `Evaluators`.



(c) When `Tasks` complete, the results are sent back to the `Driver`. The `Evaluators` maintain the state in their `Contexts`.

(d) If the Client submits another command, new `Tasks` are submitted to the existing `Evaluators` with their state accessible via `Contexts`.

Fig. 7. A REEF 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`.

4 APPLICATIONS

This section describes several applications built on Apache REEF, ranging from basic applications to production-level services. We start with an interactive distributed shell to further illustrate the life cycle of a 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 three 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 [6], a distributed log service. The second is Surf, a distributed in-memory caching service. The third is Azure Streaming Analytics, a publicly available service deployed on the Azure Cloud Platform.

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 7 depicts an execution of this application on two `Evaluators` that execute `Tasks` running a desired shell command. During the course of this execution, the `Evaluators` enter different states. The lines in the figure represent control-flow interactions.

The application starts at the Client, which submits the Distributed Shell Driver (DSD) to the RM for execution. The RM then launches the `Driver Runtime` as an Application Master. The `Driver Runtime` bootstrap process establishes a bidirectional communication channel with the Client and sends a start event to the DSD, which requests two containers (on two separate machines) with the RM. 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. 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. The Client is then prompted for a shell command. An entered command is sent 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 to the Evaluators, which execute an instance of Task 1 (Figure 7(b)). Note that Task 1 may change the state in the root Context. When Task 1 completes, the Evaluator informs the Driver Runtime. The DSD is then passed a completed task event containing the shell command output, which is sent to the client. 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 (Figure 7(d)).

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 (1,330 lines) thanks to REEF's reusable control plane.

4.2 Distributed Machine Learning

Many state-of-the-art approaches to distributed machine learning target abstractions like Hadoop MapReduce [13, 47]. 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 [1, 55, 58]. This led to proposals of new distributed computing abstractions specifically for machine learning [9, 27–29, 60]. Yet, policies for resource allocation, bootstrapping, and fault handling remain abstracted through a high-level domain-specific language (DSL) [9, 60] or programming model [27–29].

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 that a broad range of algorithms can benefit from this approach, including linear models [33], principal component analysis [25], and Bayesian matrix factorization [8]. 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 [22].

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. The problem can be cast as an optimization problem of a loss function $l(f, y)$. This function is typically

convex and differentiable in f, and thus the optimization problem is convex and differentiable in w. Therefore, it 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 $D_1, D_2, \ldots, D_k$ and compute the gradient as the sum of the per-partition gradients. This property gives rise to a simple parallelization strategy: assign each `Evaluator` partition D_i and launch a `Task` to compute the gradient on a per-partition basis. The per-partition gradients are aggregated 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.

4.2.2 Elastic Group Communications. For machine learning, we built an elastic group communications library as a REEF `Service` that exposes Broadcast, Reduce, Scatter, and Gather operators familiar to Message Passing Interface (MPI) [18] 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 Reduce and Gather operators to send messages to the root `Task`, which can aggregate those messages and use Broadcast and Scatter operators to send a message to all leaf `Tasks`. Operation details, such as the Scatter order of data items or the implementation of the Reduce function, are all adjustable and provided as APIs.

The communication structure of nodes can be established as a tree topology with internal nodes performing the preaggregation steps, given that the Reduce operation is associative. Other topologies like a flat topology, where all nonroot Tasks are leaf nodes, are also possible. The `Service` offers synchronization primitives that can be used to coordinate bulk-synchronous processing (BSP) [52] 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 ignored or repaired (synchronously or asynchronously). The loss of an internal Task cannot be ignored and must be repaired, although the method of restoration is configurable, either by replacing the dead Task with another existing Task or by blocking the computation until a new Task has spawned to take the dead Task's place. In case of asynchronous repairs, the application is notified when repairs have been finished.

In an elastic learning algorithm, the loss of leaf `Tasks` can be understood as the loss of partitions in the dataset. We can interpret these faults as being a subsample 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 [1, 15, 28, 29, 60]. The performance implications are further elaborated in Section 5.2 and in greater detail in [33].

4.2.3 Dolphin, a Machine-Learning Framework. REEF's event-driven mechanism and dependency injection system allow a framework that supports fine-grained control over various components as well as pluggable implementations of custom algorithms. Dolphin is a machine-learning framework that utilizes these features and services. Dolphin provides multistage BSP execution of distributed machine-learning algorithms, where each worker is represented by a single REEF `Task`. A stage is defined as a set of iterations of repeated computation and communication. For each stage, workers communicate with a master node, implemented as another `Task`, via the elastic group communications library, updating the model accordingly (see Section 4.2.1). A stage ends when a certain convergence threshold is met and is followed by either another stage or job termination. The REEF `Driver` is implemented to handle this stage logic, submitting `Tasks` per data partition and managing faulty `Evaluators`. When a stage transition occurs, the `Driver` checks that `Tasks` have completed without any errors and then submits new `Tasks` for the next stage.

Such stage sequences make it possible for complex jobs to be divided into simple steps instead of having complicated control flows. Moreover, such abstraction goes well with REEF's `Evaluator` design, as `Evaluators` are retainable throughout the whole job; Dolphin will use existing resources as much as it can to reduce resource allocation time by simply submitting subsequent `Tasks` on the same `Evaluators` instead of replacing `Evaluators` with new ones. On the other hand, there is no such concept of retaining containers in YARN. In order to implement the same feature of utilizing available resources on YARN without additional allocation, all Dolphin stages will need to be packed into a single container from the start, leading to crude synchronization schemes and vague stage abstractions.

Dolphin also supports distributed asynchronous model training using a parameter server [27, 42]. The parameter server is deployed as a REEF service, and thus can run independently with a `Task` that is on the same `Evaluator`. The parameter server provides a key-value store for globally shared model parameters and processes parameter update messages from workers. Using the Wake event handling mechanism of event handlers, parameter updates are received from the workers in an asynchronous manner and are applied to the global model using multiple threads concurrently. The workers, which take responsibility for receiving the updated model and sending back gradients to the server, can continue to compute gradients without waiting for their latest updates to be added to the server's parameters. Training deep neural networks is one of the many machine-learning applications that can achieve excellent performance with a parameter server [12, 14]. Dolphin provides a deep learning module, where each worker trains a neural network model and shares parameters of the model using the parameter server.

4.3 CORFU on REEF

CORFU [6] is a distributed logging tool providing applications with consistency and transactional services at extremely high throughput. There are a number of important use cases for a shared, global log, such as:

- Driving remote checkpoint and recovery
- Exposing a log interface with strict total ordering to drive replication and distributed locking
- Enabling transaction management

Importantly, all of these services are driven without I/O bottlenecks by using a novel paradigm that separates control from the standard leader I/O, prevalent 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 I/O. In aggregate, data can be streamed in parallel to/from logging units at full cluster bisection bandwidth. There are three operational modes: in-memory, nonatomic 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 *master* controls the configuration and grows/shrinks the stripe set as needed. Configuration changes are utilized both for failure recovery and for load rebalancing.

The CORFU architecture 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 the control and monitoring capabilities that the CORFU master requires. For example, when a logging unit experiences 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, react when a storage unit becomes full, and rebalance loads.

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 back up 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 with respect to 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. 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 earlier, 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 and does not need external coordination.

4.4 Surf

Surf is a distributed in-memory caching tier for big data analytics, built on REEF. By caching data residing in distributed file systems (e.g., HDFS) in memory via Surf, analytics jobs can avoid I/O delays. For example:

- Placing a dataset that is periodically read in Surf to avoid disk I/O delays
- Replicating a hot dataset across Surf nodes to avoid I/O delays due to network hotspots
- Writing temporary data directly to Surf to avoid disk I/O and replication network I/O delays of distributed file systems

Surf is deployed as a long-running service. A Surf deployment is configured to cache a cluster's distributed file system. Surf also contains an HDFS-compatible client library that talks to the deployed cache; analytics frameworks can use Surf transparently with this library by simply replacing existing paths with Surf paths. For example, reads and writes to HDFS can be cached through Surf by replacing "hdfs://namenode/" with "surf://driver/."

Surf is designed to maximize analytics cluster utilization by elastically scaling the memory allocated to its cache. Increasing the cache is desirable to keep a working set in memory as needed, while decreasing the cache allows concurrently running analytics jobs to use the released memory. This approach is in contrast to other cache systems (HDFS cache [4], Tachyon [26]), which require a cluster operator to dedicate a static amount of memory on each machine for caching.

Surf's architecture fits well with REEF's abstractions. The `Driver` plays the role of manager and point of contact for the system. Its main roles are (1) elastically deploying `Evaluators` and `Tasks`, (2) receiving client read and write operations and forwarding them to the correct `Tasks`, (3) managing and updating file metadata, and (4) enforcing replication policies. A long-running `Task` is started on each `Evaluator`, which satisfies data read and write operations as given by the

Driver. The Task transfers data between client and the backing distributed file system, caches the data locally, and sends status update heartbeats to the Driver.

Surf's implementation relies heavily on the following REEF components:

Wake Event Handling: Surf Tasks must deal with multiple data read/write operations with clients and backing file systems. Surf makes use of Wake's stage implementation to queue and execute these operations concurrently without adding additional threading complexity.

Tang Configuration: Surf is built to support additional backing file systems. This configuration is done using Tang. Tang's dependency injection approach allows Surf's core logic to remain simple since it is agnostic to the particular file system implementation. Tang's static analysis checks make sure that deployment configurations are free of misconfigured types or missing parameters.

Driver-Side Events: The Driver supports elastic scale-out and scale-in by sending Evaluator allocation and shutdown requests. Evaluator state management is easily encapsulated from the data caching logic by listening to the resulting events regarding Evaluator allocation, launch, shutdown, and failure.

Environment Adapter: Surf resource requests are translated by the Environment Adapter, allowing it to support elasticity in all REEF-supported Resource Manager environments.

Developing Surf also led to improvements in REEF. REEF heartbeats from Evaluator to Driver were adjusted to allow triggering an immediate send, in addition to periodic heartbeats. This change came about because Evaluators that reach a metrics threshold for scale-out must immediately contact the Driver. If not, we risk caching operations waiting for scale-out operations to conclude. In addition, REEF's YARN allocation method was modified based on experience deploying Surf to a large-scale cluster. We observed that Surf's startup time was around hundreds of seconds for allocating tens of Evaluators. It turned out that REEF was sending YARN resource requests one at a time, causing round-trip latency overheads to add up. By modifying REEF to batch multiple resource requests to YARN, hundreds of Evaluators can now be allocated in about 1 second. This improvement has allowed more efficient use of REEF on the YARN cluster.

4.5 Azure Stream Analytics

Azure Stream Analytics (ASA) is a fully managed stream processing service offered in the Microsoft Azure Cloud. It allows users to set up 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 a query into a dataflow of processing stages, taking user budgets into consideration, similar to [7, 10, 20, 35, 38, 51, 60, 61]. Each stage is parallelized over a set of partitions; that is, 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 dataflow. All stages must be started before query processing can begin on input data streams. The Driver uses the stage dataflow to formulate a request for resources; specifically, an Evaluator is requested on a 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 of 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

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

Fig. 8. Startup times for core REEF primitives.

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 a very positive influence on REEF developments. Most notably, REEF provides mechanisms for capturing the Task-level log files—on the containers where the Task instances executed—to a location that can be viewed locally postmortem. REEF also provides 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.

4.6 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 [43]. 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.

5 EVALUATION

Our evaluation focuses on microbenchmarks (Section 5.1) that examine the overheads of Apache 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 showcases the benefits of the REEF abstractions with the elastic learning algorithm (from Section 4).

Unless we mention, the default experiment environment 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 Ubuntu Linux and OpenJDK Java 1.7.

5.1 Microbenchmark

Key primitive measurements: Figure 8 shows the time it takes to dispatch a local Wake Event, launch a Task, and bootstrap an Evaluator. 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 multicore systems in its processing of fine-grained events, achieving a throughput rate that ranges from 20 million to 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 1 second and exit. This setup provides a baseline (ideal) job time interval (i.e., #Tasks $*$ 1 second) that we can use to assess the combined overhead of allocating and bootstrapping Evaluators and launching Tasks. Figure 9 evaluates this setup on jobs configured with various numbers of Evaluators and Tasks. We compute the

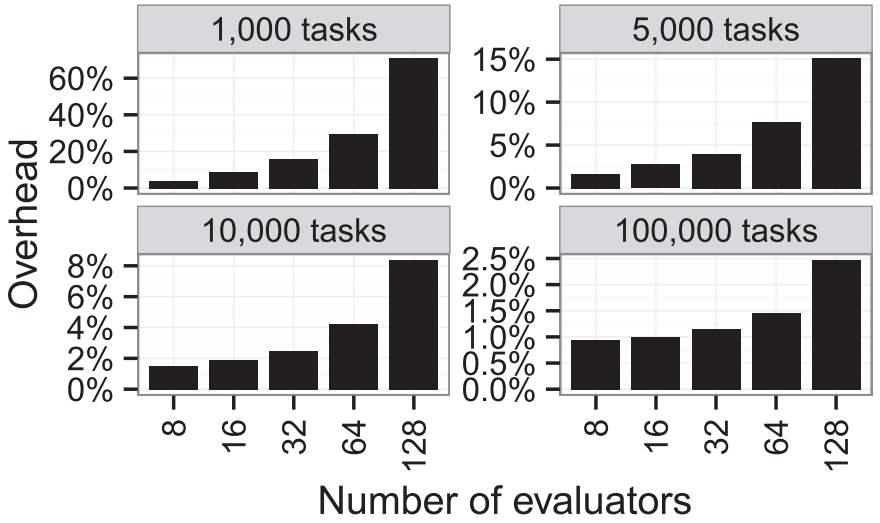


Fig. 9. Combined (REEF + YARN) overheads for jobs with short-lived (1-second) tasks.

combined overhead as

$$combined\ overhead = \left(\frac{actual\ runtime}{ideal\ runtime} - 1\right) * 100,$$

where

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

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.

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 for a job that allocates 256 `Evaluators`. The light gray and black 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 (gray portion), 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 light gray 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 to 2 seconds to the launch time for all `Evaluators`. The time to launch a `Task` (black portion) is fairly consistent, about 0.5 seconds, across all `Evaluators`.

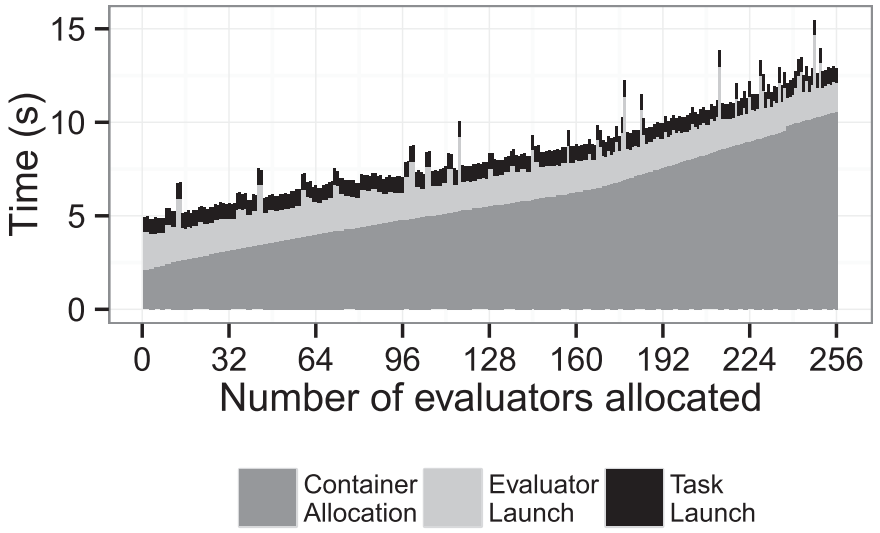


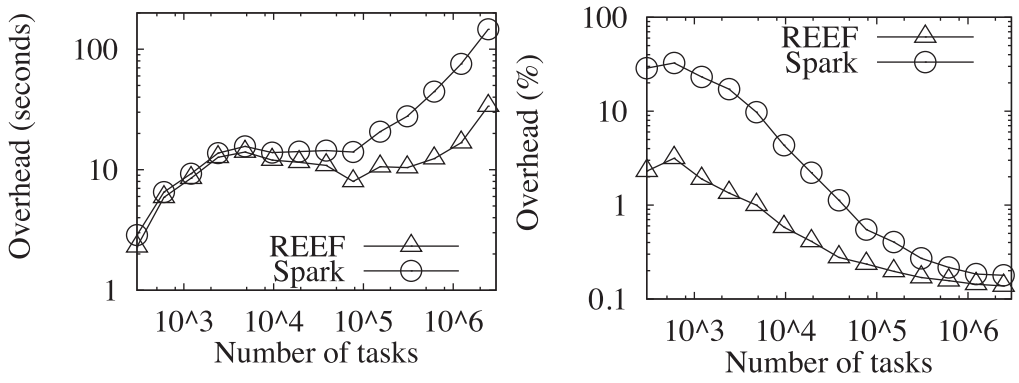
Fig. 10. Evaluator/Task allocation and launch time breakdown.

Comparison with Apache Spark: Next, we compare REEF with Spark 1.2.0 to understand task execution overheads better in 25 D4 Microsoft Azure instances. Out of the total 200 cores available, we allocated 300 YARN containers, each with 1GB of available memory. We execute tasks that do not incur CPU contention between containers. In addition, each application master was allocated 4GB of RAM. The experiment begins by instantiating a task runtime (an `Evaluator` in the REEF case, 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 earlier.

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. In REEF, library dependencies are transferred 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 11 reports on the overheads of REEF and Spark for jobs that execute a fixed number of tasks configured to sleep for 100ms before exiting. We reduce the sleep time to 100ms to stress `Driver`. The total running time is reported in Figure 11(a), and the percentage of time spent on overhead work (i.e., total running time normalized to ideal running time) is in Figure 11(b). 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 caching this information on the `Executor`. At larger job sizes, both system overheads converge to about the same.

Driver high availability: We evaluate how the `Driver` HA feature performs with two experiments. Both experiments were run in the .NET runtime on 20 Azure HDInsight instances. Figure 12 measures the time to recover when a certain number of `Evaluators` (up to 128 `Evaluators`) run before a `Driver` failure. We plot the average of the time to recover over five runs. The trend is



(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).

Fig. 11. Overheads of REEF and Spark for jobs with short-lived (100ms) tasks.

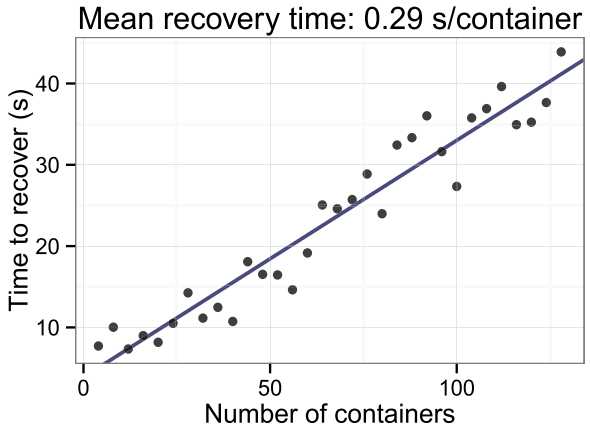


Fig. 12. Time for `Evaluators` to reregister with the `Driver` varying the number of running `Evaluators` before a `Driver` failure.

mostly linear with respect to the number of `Evaluators` up to 128 `Evaluators` as expected; the mean time to recover each `Evaluator` is 0.29 seconds.

Figure 13 measures the number of `Evaluators` registered with the `Driver` at a given point in time, starting with zero `Evaluator` in the beginning to having 128 registered running `Evaluators` at steady state. We intentionally trigger a `Driver` failure in between to observe the recovery behavior. We can see that the `Driver` recovers all 128 `Evaluators` successfully in approximately 40 seconds.

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 linear logistic regression model 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 four alphabets (A, T, G, and C).

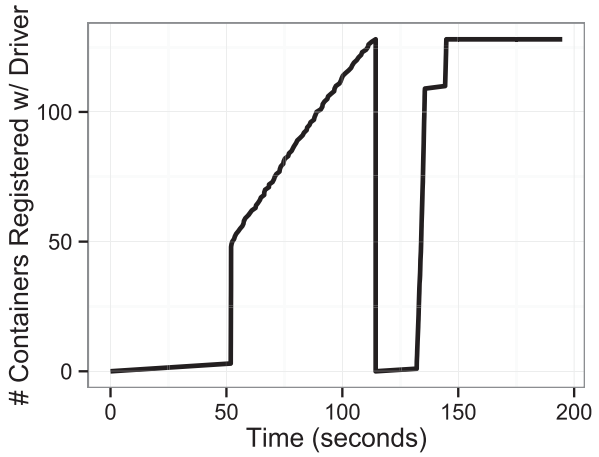


Fig. 13. The number of Evaluators that are registered with the Driver over time, with a Driver failure in the middle.

Dataset A contains a subset of 4 million examples sampled from splice, 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, used to derive the first 100,000 features per the aforementioned 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 (Section 3.1.4). 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 14 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 toward convergence. The main difference between these two environments is in the startup cost and the response time of each iteration. YARN suffers from a higher startup 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 14× increased hardware, this is a small speedup that suggests executing 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 15 plots the progress in terms of the objective function measured on the full dataset over time for both elastic and nonelastic versions of the BGD job. The line labeled

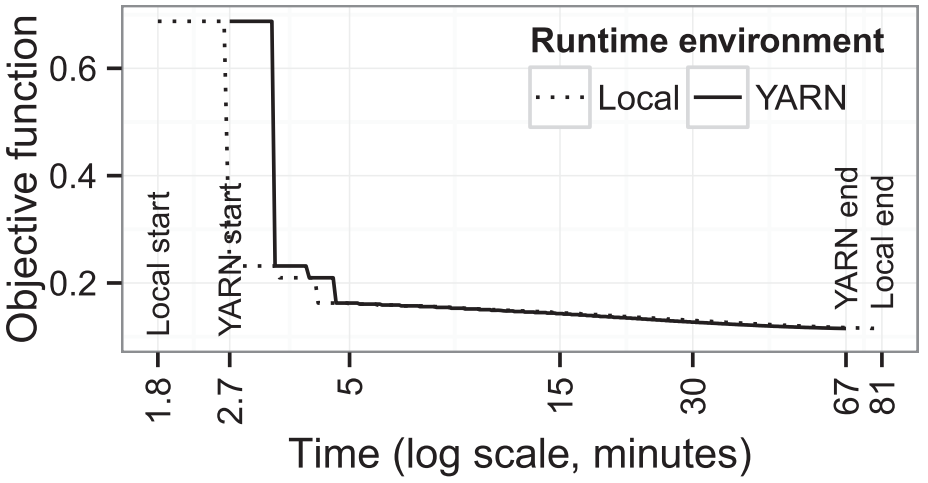


Fig. 14. Objective function over time for Dataset A when executing locally and on a YARN cluster.

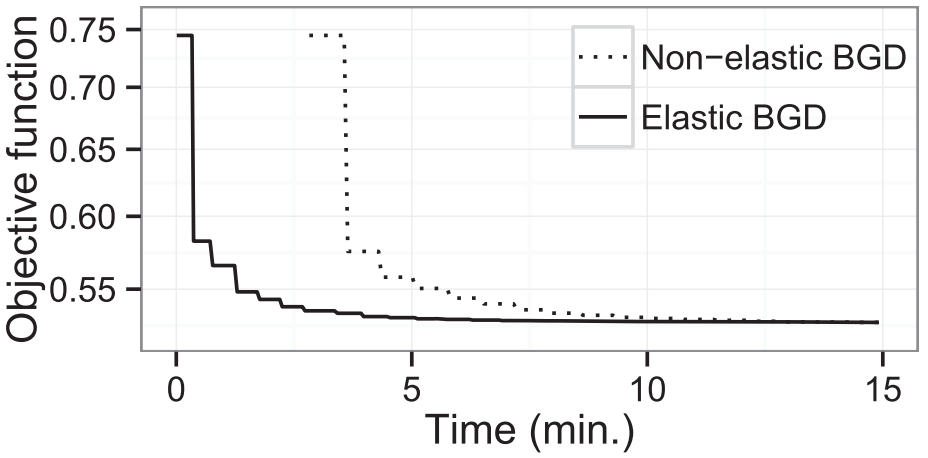


Fig. 15. Ramp-up experiment on Dataset B.

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 nonelastic 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

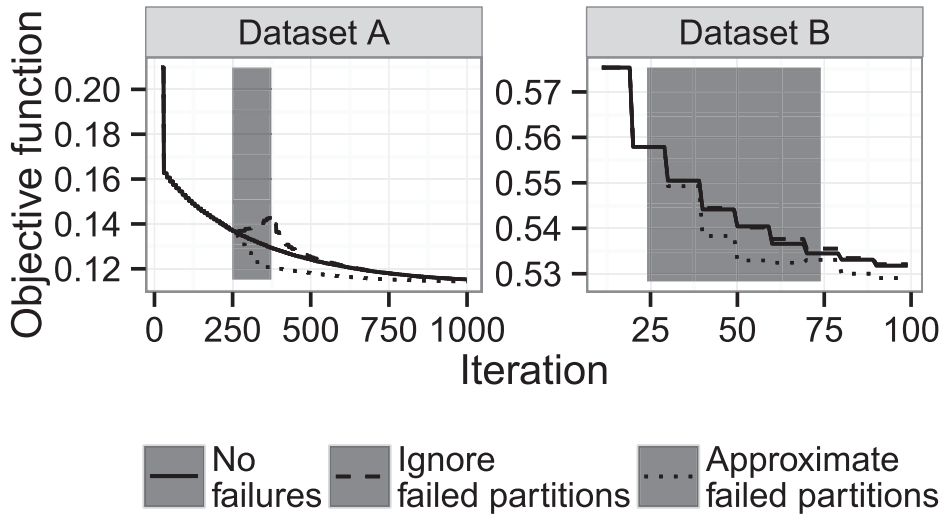


Fig. 16. Learning progress over iterations with faulty partitions. Gray areas between iterations 250..375 for Dataset A and 25..75 for Dataset B indicate the period of induced failure.

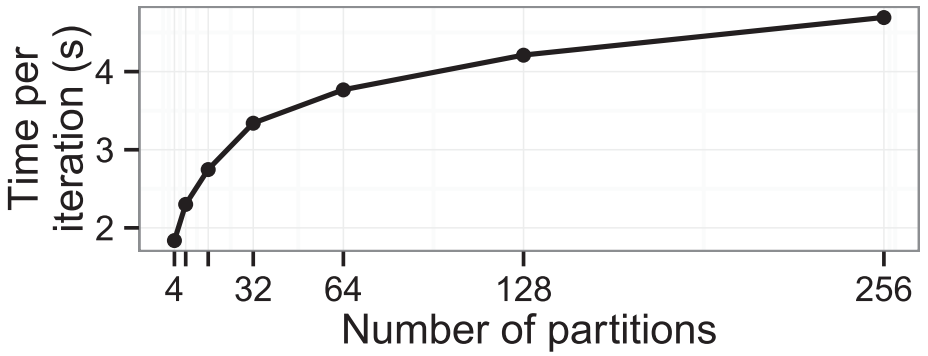


Fig. 17. Scale-out iteration time with partitions of 1GB.

represent a smaller sample of the dataset. Models obtained on small samples of the data can provide good starting points [11] 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, which uses a first-order Taylor approximation of the missing partitions' input until the partitions come back online. Figure 16 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 [37].

Scale-out: Figure 17 shows the iteration time for varying scale-up factors. It grows logarithmically as 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 [53] and Mesos [19]. REEF replaces software components common across many distributed data processing system architectures [7, 10, 20, 38, 46, 60, 61] 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). Moreover, REEF is designed to be extensible through its `Service` modules, offering applications with library solutions to common mechanisms such as group communication, data shuffle, or a more general RDD [60]-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 [28], Twister [16], Giraph [45], and VW [1]. REEF can also be leveraged to support stream processing systems such as Storm [30] and S4 [34] on managed resources, as demonstrated with Azure Streaming Analytics (Section 4.5). Finally, REEF has been designed to facilitate sharing data across frameworks, short-circuiting many of the HDFS-based communications and parsing overheads incurred by state-of-the-art systems.

The Twill project [49] and REEF both aim to simplify application development on 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 [24]) 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 [48] 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 [44] to run on YARN with little modification. Therefore, the goals of Slider and REEF are different.

Tez [38] 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 toward decoupling resource management from higher-level application stacks. In this article, 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, seeding 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 toward 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 [59] 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 top-level project. Code and documentation can be found at http://reef.apache.org. The level of engagement both within Microsoft and from the research community reinforces our hunch that REEF addresse s fundamental pain points in distributed system development.

ACKNOWLEDGMENTS

We would like to thank our many partners in the Microsoft Big Data product groups for their feedback and guidance in the development of REEF.

REFERENCES

[1] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. 2011. A reliable effective terascale linear learning system. *CoRR* abs/1110.4198 (2011).

[2] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. 2012. Scalable inference in latent variable models. In *ACM International Conference on Web Search and Data Mining (WSDM'12)*.

[3] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. 2011. Consistency analysis in bloom: A CALM and collected approach. In *Conference on Innovative Data Systems Research (CIDR'11)*.

[4] Colin McCabe and Andrew Wang. 2013. Centralized cache management in HDFS. https://issues.apache.org/jira/browse/HDFS-4949.

[5] The Kubernetes Authors. 2015. Kubernetes. Retrieved from https://kubernetes.io/.

[6] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A distributed shared log. *ACM Transactions on Computer Systems (TOCS)* 31, 4 (2013), 10.

[7] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. 2010. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *ACM Symposium on Cloud Computing (SoCC'10)*.

[8] Alex Beutel, Markus Weimer, Vijay Narayanan, Yordan Zaykov, and Tom Minka. 2014. Elastic distributed bayesian collaborative filtering. In *NIPS Workshop on Distributed Machine Learning and Matrix Computations*.

[9] Vinayak Borkar, Yingyi Bu, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. 2012. Declarative systems for large-scale machine learning. *IEEE Technical Committee on Data Engineering (TCDE)* 35, 2 (2012), 24–32.

[10] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *International Conference on Data Engineering (ICDE'11)*.

[11] Olivier Bousquet and Léon Bottou. 2007. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems (NIPS'07)*.

[12] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 571–582.

[13] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. 2006. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems (NIPS'06)*.

[14] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*. 1223–1231.

[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1 (2008), 107–113.

[16] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. 2010. Twister: A runtime for iterative mapreduce. In *ACM International Symposium on High Performance Distributed Computing (HPDC'10)*.

[17] Google. 2015. Guice. Retrieved from https://github.com/google/guice.

[18] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. 1998. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press.

[19] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*.

[20] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *ACM European Conference on Computer Systems (EuroSys'07)*.

[21] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. 2010. An analysis of traces from a production mapreduce cluster. In *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'10)*.
[22] Michael Kearns. 1998. Efficient noise-tolerant learning from statistical queries. *Journal of the ACM* 45, 6 (1998), 983–1006.
[23] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.
[24] J. Kreps, N. Narkhede, and J. Rao. 2011. Kafka: A distributed messaging system for log processing. In *International Workshop on Networking Meets Databases (NetDB'11)*.
[25] Arun Kumar, Nikos Karampatziakis, Paul Mineiro, Markus Weimer, and Vijay Narayanan. 2013. Distributed and scalable PCA in the cloud. In *BigLearn NIPS Workshop*.
[26] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *ACM Symposium on Cloud Computing (SoCC'14)*.
[27] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.
[28] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI'10)*.
[29] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*.
[30] Nathan Marz. 2015. Storm: Distributed and Fault-Tolerant Realtime Computation. http://storm.apache.org.
[31] Erik Meijer. 2012. Your mouse is a database. *Communications of the ACM* 55, 5 (2012), 66–73.
[32] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A timely dataflow system. In *ACM Symposium on Operating Systems Principles (SOSP'13)*.
[33] Shravan Narayanamurthy, Markus Weimer, Dhruv Mahajan, Tyson Condie, Sundararajan Sellamanickam, and S. Sathiya Keerthi. 2013. Towards resource-elastic machine learning. In *BigLearn NIPS Workshop*.
[34] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. 2010. S4: Distributed stream computing platform. In *IEEE International Conference on Data Mining Workshops (ICDMW'10)*.
[35] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: A not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*.
[36] Ariel Rabkin. 2012. Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software. Ph.D. Dissertation, UC Berkeley (2012).
[37] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1985. *Learning Internal Representations by Error Propagation*. Technical Report. DTIC Document.
[38] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. 2015. Apache tez: A unifying framework for modeling and building data processing applications. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*.
[39] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, scalable schedulers for large compute clusters. In *ACM European Conference on Computer Systems (EuroSys'13)*.
[40] Marc Shapiro and Nuno M. Preguiça. 2007. Designing a commutative replicated data type. *CoRR* abs/0710.1784.
[41] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*. ACM, New York,1135–1149. DOI: http://dx.doi.org/10.1145/2882903.2915229
[42] Alexander Smola and Shravan Narayanamurthy. 2010. An architecture for parallel topic models. *Proceedings of the VLDB Endowment* 3, 1–2 (2010), 703–710.
[43] Michael Stonebraker and Ugur Cetintemel. 2005. One size fits all: An idea whose time has come and gone. In *International Conference on Data Engineering (ICDE'05)*.
[44] The Apache Software Foundation. 2017. Apache Accumulo. Retrieved from http://accumulo.apache.org/.
[45] The Apache Software Foundation. 2017. Apache Giraph. Retrieved from http://giraph.apache.org/.
[46] The Apache Software Foundation. 2017. Apache Hadoop. Retrieved from http://hadoop.apache.org.
[47] The Apache Software Foundation. 2017. Apache Mahout. Retrieved from http://mahout.apache.org.
[48] The Apache Software Foundation. 2017. Apache Slider. Retrieved from http://slider.incubator.apache.org/.
[49] The Apache Software Foundation. 2017. Apache Twill. Retrieved from http://twill.apache.org/.
[50] The Netty Project. 2015. Netty. Retrieved from http://netty.io.
[51] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive – A warehousing solution over a map-reduce framework. In *Proceedings of the VLDB Endowment (PVLDB'09)*.

[52] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Communications of the ACM* 33, 8 (1990), 103–111.
[53] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache hadoop YARN: Yet another resource negotiator. In *ACM Symposium on Cloud Computing (SoCC'13)*.
[54] Markus Weimer, Yingda Chen, Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Yunseong Lee, Tony Majestro, Dahlia Malkhi, Sergiy Matusevych, Brandon Myers, Shravan Narayanamuthy, Raghu Ramakrishnan, Sriram Rao, Russell Sear, Beysim Sezgin, and Julia Wang. 2015. REEF: Retainable evaluator execution framework. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*.
[55] Markus Weimer, Sriram Rao, and Martin Zinkevich. 2010. A convenient framework for efficient parallel multipass algorithms. In *NIPS Workshop on Learning on Cores, Clusters and Clouds*.
[56] Matt Welsh. 2013. What I wish systems researchers would work on. Retrieved from http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html.
[57] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An architecture for well-conditioned, scalable internet services. *SIGOPS Operating Systems Review* 35 (2001), 230–243.
[58] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. 2009. Stochastic gradient boosted distributed decision trees. In *ACM Conference on Information and Knowledge Management (CIKM'09)*.
[59] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*.
[60] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'10)*.
[61] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: Parallel databases meet mapreduce. *VLDB Journal* 21, 5 (2012), 611–636.

Received December 2015; revised February 2017; accepted July 2017