Experiences Running Mixed Workloads on Cray Analytics Platforms

Haripriya Ayyalasomayajula
Cray Inc.
901 Fifth Avenue, Suite 1000
Seattle, WA 98164
hayyalasom@cray.com

Kristyn J. Maschhoff
Cray Inc.
901 Fifth Avenue, Suite 1000
Seattle, WA 98164
kristyn@cray.com

***Abstract*—The ability to run both HPC and big data frameworks together on the same machine is a principal design goal for future Cray analytics platforms. Hadoop™provides a reasonable solution for parallel processing of batch workloads using the YARN resource manager. Spark™is a general-purpose cluster-computing framework, which also provides parallel processing of batch workloads as well as in-memory data analytics capabilities; iterative, incremental algorithms; ad hoc queries; and stream processing. Spark can be run using YARN, Mesos™or its own standalone resource manager. The Cray Graph Engine (CGE) supports real-time analytics on the largest and most complex graph problems. CGE is a more traditional HPC application that runs under either Slurm or PBS. Traditionally, running workloads that require different resource managers requires static partitioning of the cluster. This can lead to underutilization of resources.**

In this paper, we describe our experiences running mixed workloads on our next generation Cray analytics platform (internally referred to as "Athena") with dynamic resource partitioning. We discuss how we can run both HPC and big data workloads by leveraging different resource managers to interoperate with Mesos, a distributed cluster and resource manager, without having to statically partition the cluster. We also provide a sample workload to illustrate how Mesos is used to manage the multiple frameworks.

***Keywords*-HPC, Big Data, Mesos, Marathon, Yarn, Slurm, Spark, Hadoop, CGE**

I. Introduction

The ability to run both HPC and big data frameworks together on the same platform is highly desirable. Such a capability provides better overall system resource utilization, as dedicated clusters targeting one specific capability or framework may become idle at times based on workload characteristics and requirements. One single system capable of supporting multiple frameworks eliminates the need to maintain multiple systems. A second and equally important benefit this capability provides is significant gain in user productivity. By enabling mixed workflows to run on a single platform, users can run more complex workflows. Users no longer have to copy large amounts of data between systems. Moving multiple terabytes of data between systems becomes a significant bottleneck for many mixed-workload applications. This may also enable applications running in different frameworks to exchange data more seamlessly using light-weight interfaces, either leveraging a shared file system such as Lustre™, or by storing data in memory without needing to write data to the file system. Analytics frameworks provide REST endpoints that are helpful for users and operators. We can use the REST API provided by the frameworks to connect to the HTTP endpoints of their components.

Traditionally, various implementations of Parallel Batch System (PBS), and more recently Slurm, have been the resource managers used for HPC platforms. Big data frameworks work with a variety of different resource managers for their job scheduling and resource management. The Hadoop [1] ecosystem uses YARN (Yet Another Resource Negotiator) [2] as its default resource manager. Apache Spark [3] comes with its own standalone resource manager but can also work with YARN or Mesos [4]. On most systems today, running workloads that require different resource managers requires static partitioning of the cluster or HPC system.

In this paper, we describe our experiences running mixed workloads on an Athena development system with dynamic resource partitioning. The Athena analytics test platform used for this study was an Aries™-based system with node-local SSDs. Each node has two sockets populated with 12-core Haswell processors, 512GB of DDR-4 memory, a 1.6 TB SSD and dual 1.0 TB hard drives. We first describe the three frameworks supported on this platform and provide some background about the unique characteristics and core components for each of the frameworks. We then discuss Mesos, a distributed cluster and resource manager, and describe how we are able to support both HPC and big data workloads by leveraging different resource managers to interoperate with Mesos without having to statically partition the cluster. We then introduce a mixed workload that uses both an analytics-style application (Spark) and an HPC-style application (Cray Graph Engine) and illustrate how Mesos is used to manage the multiple frameworks. Finally we discuss our experiences running this mixed workload and identify possible improvements and areas of future work.

II. Description of Frameworks

In this section we describe the three frameworks that we support on the Athena system: the Hadoop/YARN ecosystem, the Spark analytics framework, and the Cray Graph

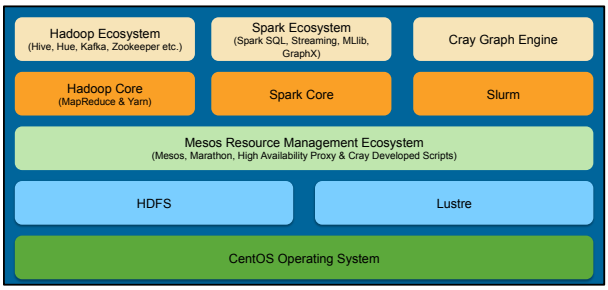

Figure 1. Software architecture diagram for the Athena platform

Engine (CGE) framework. For each framework, we discuss its core components and the deployment modes supported.

Figure 1 shows the overall structure of the Athena software stack.

A. Hadoop/YARN Ecosystem

Hadoop is an open-source framework used for parallel distributed processing of batch workloads on large data sets. The Hadoop/YARN ecosystem is built around two core components: Hadoop Distributed File System (HDFS™) [5] (storage) and an analysis system, MapReduce [6] (processing). Together, they provide a computing solution that is scalable, cost-effective, flexible and fault-tolerant. It is designed to scale up from small servers to clusters with thousands of nodes.

HDFS serves as primary data storage for Hadoop workloads and also supports a wide range of data formats. HDFS works best for storing very large files and retrieving them in a write once, read many times fashion. It is designed to offer fast local storage and high-throughput access to data. Fault tolerance and resiliency are ensured by replicating the data across the nodes. Data locality is offered by scheduling task execution on the nodes where the data is present. HDFS components also ensure persistency of the data. HDFS provides a limited interface for managing the distributed file system; other applications in the Hadoop ecosystem can access data through this interface.

On the Urika-XA™platform and also on the Athena platform, HDFS storage is supplied by the SSD and HDDs on each of the compute nodes to form the HDFS filesystem. Data stored in HDFS is accessible from everywhere and at all times, both by the applications running within the Hadoop ecosystem framework, but also from other frameworks including Spark and CGE. Storing data on the SSDs located on the compute nodes that needs to be persistent across jobs and globally accessible is fundamentally different from the data storage model used by HPC-style applications, where persistent data is stored off-node in an external filesystem.

YARN was first introduced in Hadoop version 2. YARN supports both MapReduce and a collection of non-MapReduce applications in the same Hadoop environment. While YARN provides resource management functionality to workloads launched in the Hadoop ecosystem, it cannot be extended easily to non-Hadoop workloads. YARN provides a scheduler integrated with its resource manager. This becomes a limitation when writing a new framework on top of YARN, as the new framework will also have to use the scheduler provided by YARN. This may not be the most optimal solution for all frameworks. So in order to manage resources across all three frameworks we need a scalable yet more "global" resource manager solution to schedule different types of workloads, both big data applications contained within the Hadoop ecosystem and other traditional HPC applications, such as MPI-based applications. Mesos allows each framework to implement its own scheduler which overcomes this limitation.

At CUG 2014, Sparks et al. [7] discussed the development of a Cray framework for Hadoop on the XC30 system based on the myHadoop project by Sriram Krishnan. In their paper they discussed many of the obstacles that complicate the adoption of Hadoop as a part of typical HPC workflow, the most significant involving workload management. Several components of the Hadoop ecosystem, including database applications such as Accumulo™or HBase™, are intended to be used as long-running, shared services, and are not well suited to the batch oriented model typically used for managing large HPC systems.

B. Spark Analytics Framework

Apache Spark is a general-purpose cluster-computing framework that also provides parallel processing of batch workloads. Spark supports a wide range of distributed computations and facilitates reuse of working data sets across multiple parallel operations by its potential to keep data in memory. Spark facilitates interactive ad hoc querying on large datasets. It offers Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

Spark provides high-level APIs in Java, Scala, Python and R. Unified programming abstractions called *Resilient Distributed Datasets* (RDDs) are the fundamental building blocks of Spark. They are the primary storage primitives that facilitate storage of data in memory and across multiple compute nodes.

Spark can be run using its stand-alone resource manager or over several existing cluster managers such as Hadoop YARN or Apache Mesos. Spark jobs can be run in batch mode using the `spark-submit` command or they can be run interactively through the supported Spark shells (Scala, Python, SparkR, SparkSQL).

When Spark is run in batch mode, the Spark programmer writes a driver program, which is the main program. Inside Main, the programmer sets up the SparkContext. As part of setting up the SparkContext, the programmer can specify arguments on how the Spark job is to be run. In addition,

when `spark-submit` is called, users can alternately pass arguments, such as the number of executors, which can specify how the job is to be run.

The Spark program then connects with a cluster manager to gain resources required to launch the jobs. The Spark driver establishes a connection with the resource manager. Spark acquires executors on nodes in the cluster. Executors are processes that run computations. These long-lived processes can store RDD partitions in memory across operations. One or more RDDs are defined in the driver program and actions are invoked on them. When Spark is run interactively, the `spark-shell` will start up the Spark context.

Although there is overlap between the Spark and Hadoop workloads, the Spark and Hadoop frameworks were developed using different design considerations, and each these offer optimal solutions to different classes of algorithms and constraints. A discussion of early experiences developing applications for both Spark and Hadoop aimed at evaluating the performance offered by Spark can be found in [8].

C. CGE Framework

In contrast to Hadoop and Spark, which were originally developed to run on commodity clusters, the Cray Graph Engine (CGE) was developed to run in a Cray HPC environment. CGE is a high performance semantic graph database based on the industry-standard RDF graph data format and the SPARQL graph query language. At CUG 2015, we presented our early work to port the back-end query engine of the Urika-GD appliance to the Cray XC40 system [9].

The CGE query engine performs several functions. First, the query engine is responsible for reading RDF, translating it into its memory-resident representation of the database, and writing the database to disk. Second, the query engine is able to read in a previously compiled database from disk in order to build its memory-resident database image. Third, it accepts SPARQL queries from the front end, evaluates them against the memory-resident database, and returns the results to the front end. Fourth, it accepts SPARUL (SPARQL Update) commands from the front end and updates the memory-resident database according to those commands. Finally, it also accepts checkpoint commands from the front end, writing the database to disk.

The back-end query engine of CGE is written as a distributed application using Coarray C++ as the underlying distribution mechanism. Coarray C++ is a C++ template library that runs on top of Cray's Partitioned Global Address Space (PGAS) library (libPGAS), which is an internal Cray library supporting the Cray compiler, built on top of DMAPP [10]. As a graph analytics application, maintaining optimal network performance for small word remote references is essential. In order to fully utilize the RDMA and synchronization features provided by the Aries interconnect, we leveraged existing software running on our XC40 systems, bringing over the necessary components of the Cray XC software stack required to launch and run the CGE application. Currently Slurm is used for launch and job initialization, and is the mechanism used to configure Aries communication. The use of Slurm is not exposed to the user directly. The user only interacts with the CGE launcher. The CGE launcher is a Python-based script, which on Athena uses Slurm to allocate nodes from Mesos, start Slurm components on the allocated nodes, set up the HPC software stack, launch CGE, and return nodes to Mesos when the job is finished. We provide more details on how this done in the next section.

III. Cluster Resource Management

We chose Apache Mesos as the primary resource manager for the Athena platform. Mesos supports diverse frameworks on the same platform without static partitioning. When one performs static partitioning, specific portions of the cluster are allocated to run each framework. Resources allocated to a specific framework are idle when there are no jobs running under that framework, potentially resulting in poor resource utilization. Mesos allocates resources dynamically to launch jobs of different frameworks, thereby improving system utilization compared to static partitioning.

A. Mesos

Apache Mesos is the main resource manager for the Athena platform. It provides the necessary functionality to share system resources between the three diverse frameworks. Over the last decade, big data frameworks such as Hadoop and Spark have gained popularity and new cluster computing frameworks continue to evolve. Having multiple frameworks supported in the same system provides user with the ability to select the best framework for each application. Each of these frameworks evolved with different design considerations targeted to solve problems of different classes of applications. Similarly, each of these frameworks interacts with different resource managers to schedule its jobs on the cluster. Apache Mesos addresses the need for a common interface that allows cluster resource management for multiple existing frameworks. It also has the ability to support future frameworks.

Resource management and scheduling are two related problems. Each framework has different scheduling needs based on its programming model, communication pattern, task dependencies and data placement. The way the frameworks address fault tolerance and high availability can differ too. To have one common platform with multiple frameworks we can either choose to have a centralized scheduler or a decentralized scheduler. To satisfy these requirements with a single scheduler is very complex and not feasible, particularly while maintaining scalability and resiliency. In addition, it is difficult to predict the scheduling requirements for any upcoming frameworks. Mesos does not implement a centralized scheduler. Instead, it delegates the scheduling

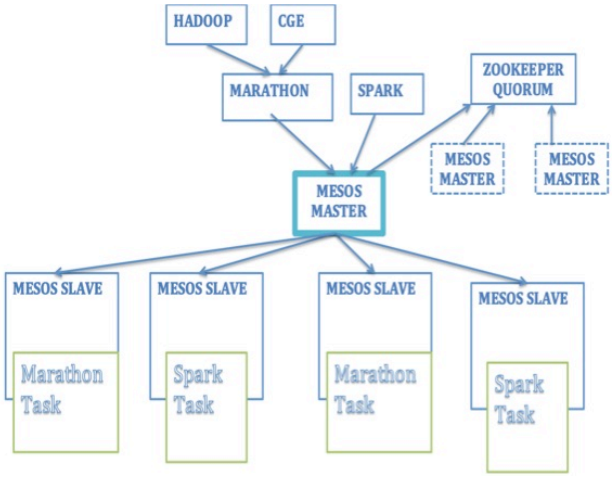

Figure 2. Overview of Mesos architecture

decision to frameworks. Mesos has a distributed two-level scheduling mechanism called resource offers. It encapsulates a bundle of resources that a framework can allocate on a cluster node to run tasks into resource offers.

Mesos decides how many resources to offer to each framework based on an organizational policy. The frameworks decide which resources to accept and which computations to run on them. The task scheduling and execution decisions are left to the framework. This enables the framework to implement diverse approaches to various problems in the cluster example: data locality and fault tolerance, which can evolve independently. While the decentralized scheduling model is not always the globally optimal solution, it is efficient to support diverse workloads.

Like many other distributed systems, Mesos has been implemented in a master-worker pattern. Mesos consists of Mesos master, Mesos slave, and frameworks. There is one Mesos agent/slave daemon that runs on each node of the cluster. There is a Mesos master that manages agent/slave daemons running on each cluster node. Frameworks that register with Mesos run tasks on these slaves. A resource offer comprises a list of free resources on multiple slaves. Frameworks running on Mesos have two components: scheduler, a process that registers with master to be offered resources, and executor, a process that is launched on a slave node to run the framework tasks.

Figure 2 provides an overview of the Mesos architecture. The Mesos master decides how many resources to offer to each framework. The framework scheduler then selects which of the offered resources to use. The framework chooses either to accept or decline the resources. When a framework accepts offered resources to use, it passes to Mesos a description of the tasks it wants to launch. The master allocates the proper resources and then sends tasks to the slave. A framework has the ability to reject the resources that do not satisfy its requirements or constraints. It is this ability that allows frameworks to ensure constraints, such as data locality. However, it poses some challenges too. Frameworks have to wait for resources that satisfy the constraints. Mesos may have to send out multiple resource offers before one is accepted.

To avoid this, Mesos lets the framework set filters to specify that a framework will always reject certain resources. Mesos shares resources in a fine-grained manner. It allows frameworks to achieve near-optimal data locality by taking turns reading data stored on each machine. Mesos also supports having multiple versions of frameworks in the same cluster. On the Athena platform, for example, support for multiple versions of frameworks could be used to support multiple versions of Spark.

B. Framework Resource Allocation

On the Athena system, we have a hierarchy of resource managers operating with Mesos. The frameworks are configured to interact with Mesos for acquiring resources to launch their jobs. For frameworks not native to Mesos, we provide interfaces to allow these resource managers, such as Slurm and YARN, to dynamically acquire and release resources from Mesos.

Mesos is set up to run in high-availability mode with three masters and operates with a quorum of two. It is configured with Zookeeper™, which is a distributed coordinating service. This ensures fault tolerance and resiliency. If one of the master processes dies, one of the other two masters is elected as a leader. Mesos slaves are started on the nodes of the cluster.

Spark running as a native Mesos framework on the Athena system: Spark is run as a native Mesos framework and interacts directly with Mesos for resource allocation. This is accomplished by simply pointing the Spark master to a Mesos cluster set up in high-availability mode with three masters configured with ZooKeeper. By default we set the Spark master to the Mesos masters in the SparkConf file. For our experiments we run Spark on Mesos using the default coarse-grained mode. The coarse-grained mode launches only one long-running Spark executor on each Mesos slave. The Spark executor manages the parallel tasks across all the cores on that node. By default, it acquires all cores in the cluster (that are offered by Mesos), but the user can limit the number of cores requested by setting the spark.max.cores configuration parameter. Figure 3 shows how Spark interacts with Mesos for resource allocation.

Marathon: Hadoop/YARN and CGE are not run as native Mesos frameworks. We build our solutions for both Hadoop/YARN-based applications and traditional Slurm-based HPC applications on top of Marathon. Marathon is a cluster-wide initialization and control system for running services under the Mesos ecosystem. Marathon registers itself as a Mesos framework. Marathon is REST-based and

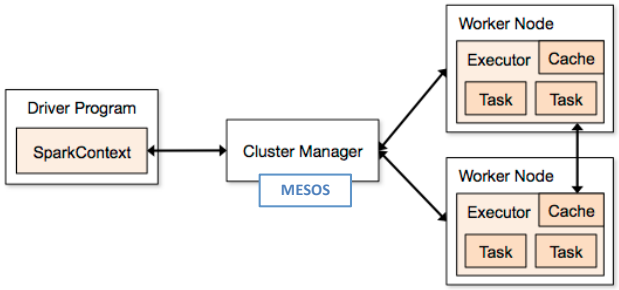

Figure 3. Running Spark on Mesos

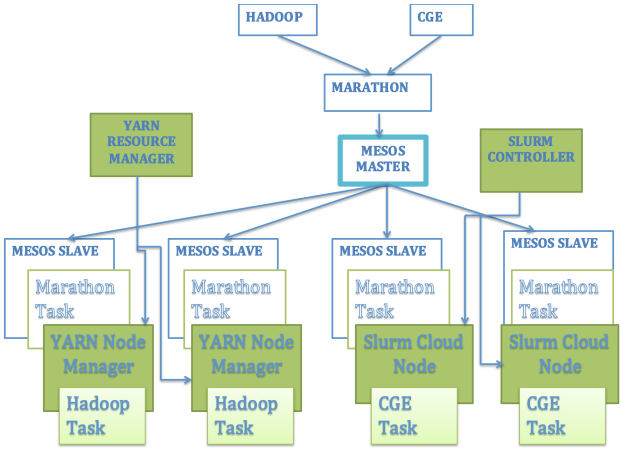

Figure 4. Launching Hadoop and CGE using Marathon on Athena

provides an API for starting, stopping and scaling long-running services. On the Athena platform, Marathon is used on top of Mesos to launch Slurm and YARN. Figure 4 provides an illustration of the launch process.

Hadoop on the Athena platform: Hadoop version 2 workloads require YARN, and Mesos enables running multiple frameworks together. Hence, YARN and Mesos should coexist. Our solution uses Marathon to launch a dynamic YARN sub-cluster under Mesos. Marathon is a part of the Mesos ecosystem that provides a mechanism for launching long-running applications. The scripts we developed use Marathon to expand or shrink a YARN cluster. The cluster remains under the control of Mesos even when we run other cluster managers. When we call the `flex-up` script, YARN node managers are started as Mesos slave tasks using Marathon. Marathon negotiates with Mesos for resources and sets up a YARN cluster for launching jobs. Once the YARN cluster is flexed up with the required number of nodes, users can submit Hadoop jobs to YARN. When the Hadoop jobs have completed and YARN no longer requires the resources a `flex-down` script terminates the application running under Marathon. This stops the YARN node managers and returns resources from YARN back to Mesos.

CGE on the Athena platform: We launch CGE on the Athena platform using Slurm configured to run in Slurm's Elastic Computing Cloud mode. Slurm is a free open-source workload manager that has been adapted by Cray to support HPC applications on Cray platforms. Slurm normally expects to own and manage all the resources on a platform. In Slurm's Elastic Computing Cloud mode, Slurm allocates from a fixed pool of physical nodes and launches the sub-tasks of a job on each node. From an HPC application perspective, Slurm is important primarily because it knows how to configure communication among the sub-tasks, whereas Mesos and Marathon have no native awareness of these kinds of issues.

With the advent of cloud computing services like Amazon's EC2 product, Slurm was extended, using elements of its power management infrastructure, to permit leasing of Slurm resources on an as-needed basis from a cloud computing provider. The mechanism provides for two scripts run by the Slurm controller daemon. One of the scripts "suspends" a node, meaning that it terminates Slurm components running on that node and releases its "lease" on the node. The other script "resumes" a node, meaning that it leases the node and initiates Slurm components on that node. When the Slurm controller determines that it needs to extend its cloud computing resources to accommodate a job, it 'resumes' a cloud node. When the Slurm controller no longer needs those resources it 'suspends' the cloud computing resource.

We observed that Mesos could be treated as a cloud computing service on the Athena system by adding the necessary supporting scripts to suspend and resume cloud nodes managed primarily by Mesos. To do this, Slurm uses Marathon to allocate nodes from Mesos and launch Slurm components on those nodes. A set of Cray-developed scripts provides the interaction with Mesos and Marathon necessary to make this happen. With this infrastructure in place, we can then launch CGE using Slurm with the required HPC software stack and hardware initialization, almost in the same way Slurm performs this function on the XC40 system. The main difference is that on the Athena platform, Slurm is deferring to Mesos for the underlying resource management.

A summary of the steps involved when the CGE launcher gets invoked follows. As part of the launch command, the user specifies how many nodes to use and how many images per node. If the number of nodes and images is not supplied by the user, the default is to request one node and a heuristic is used to determine the number of images per node based on number of cpu cores. Memory considerations for holding the memory-resident database and the complexity of the queries to be run are two guiding factors when selecting an appropriate node count to use. The cge-launch script composes an appropriate Slurm srun command, which queues the job to the Slurm controller. From there the Slurm controller determines what resources it needs from its cloud service (Mesos) and runs Slurm with Mesos "resume"

script to obtain the needed nodes. This script launches a complement of Slurm daemons under Marathon (which takes care of the Mesos offer solicitation and acceptance process and starts the Slurm daemons). Once the Slurm daemons are running and registered with the Slurm controller, the Slurm controller places the job on the newly-started nodes and lets it run. When the job completes, the nodes become idle. The Slurm controller waits for a brief system-configured interval in case a subsequent job might use the idle nodes immediately; then, if the nodes remain idle, "suspends" them using the Slurm with Mesos suspend script, returning the no-longer-needed nodes to Mesos.

IV. Social Network Analysis Workflow

To illustrate how Mesos is used to manage the multiple frameworks, we use a mixed workload that includes both an analytics-style application (Spark) and an HPC-style application (CGE). The example is derived from a social network analysis workflow originally developed in Spark to run on the Urika-XA system [11]. The original Spark workflow has two modes or pipelines, a real-time analytics pipeline and a batch analytics pipeline. The real-time pipeline performs operations which can be completed in a small window timeframe, such as simple aggregations. Example aggregations include counting the total number of tweets, unique users or unique hashtags. The batch analytics pipeline operates with a larger window timeframe to both collect more data and perform more complex analysis on that data.

We have modified the batch analytics pipeline to illustrate how efficiently one can use both Spark and CGE to execute the components of this workflow. The modified pipeline demonstrates the exchange of data between these applications using either HDFS or Lustre. This modified use case also shows how we can apply Spark to the ETL (Extract, Transform, Load) components of the workflow for which it is most suited, and likewise can apply CGE to graph analysis components of the workflow where it is most suited. This does not require the copying of large amounts of data that has to be accessed by different frameworks between systems or from one file system to another.

The full workflow includes ETL, numerous aggregations, and joins. The batch analytics pipeline includes running a community detection graph algorithm. We use Spark and Spark Streaming to process the data as a series of micro-batches. Spark is used to perform the ETL and generate RDF data which can then be loaded into CGE. These are tasks at which Spark excels. The key programming abstraction used in Spark Streaming is called a DStream, or distributed stream. Conceptually, a DStream is just a series of RDDs, and any operation applied on a DStream translates to operations on the underlying RDDs. Although Spark can be used to perform the various aggregations and joins required by the workflow, the relatively poor performance

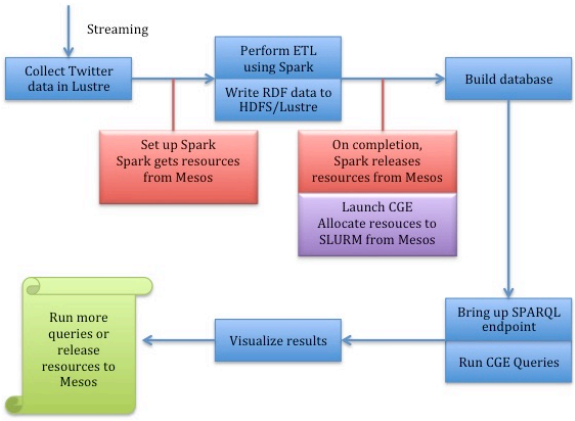

Figure 5. Sequential workflow for social network analytics example

of the GraphX community detection algorithm limits the amount of data that can be used for identifying communities. This task is much better suited for CGE.

Figure 5 shows a simple flow diagram for the social-analytics workflow where we run Spark followed by CGE in a serial fashion. This workflow will demonstrate the process of starting up one framework across most of the nodes on the system and running, shutting down, and bringing up another framework on those same nodes, all on the same hardware. Although the workflow executes the two components (Spark, CGE) serially, the overall runtime is still greatly reduced. The magnitude of the efficiency of CGE over Spark for the graph analytics portion of the workflow easily justifies the added the overhead of generating and writing RDF data to HDFS, dynamically switching from running a Spark framework across all nodes to running the CGE framework across all nodes, loading the RDF data into CGE and running queries.

Figure 6 shows a modified flow diagram for the social-analytics workflow where we have the Spark Framework and the CGE Framework both running at the same time on different nodes on the system. In this workflow, the RDF data generated from each streaming batch from Spark can be loaded into CGE as SPARQL update operations to an already-running CGE database. The exchange of data

between the two system components is managed through the HDFS file system using the time-stamped RDF data directories generated from Spark Streaming as the mechanism for initiating the SPARQL load operation for those data files.

The source data is from Twitter. Each tweet record contains the user making the tweet and any users mentioned in the text of the tweet. Spark Streaming is used to process the data as a series of micro-batches. The ETL phase consists of parsing the JSON record and reorganizing it into structures analogous to relational tables (tweets, users, relationships, hashtags). For each batch, we generate and write out RDF data to HDFS by using the Spark Streaming saveAsTextFiles operator. The output directory on HDFS where the generated RDF data is written for each batch includes a time prefix. This gives each batch interval a uniquely-named directory, allowing several batches to be processed before switching to run CGE. We write the RDF data to an HDFS directory with the All_SSD storage policy set (see Hadoop 2.7 Archival Storage, SSD & Memory online documentation [12]). This causes HDFS to attempt to place all blocks (and their replicas) into the SSD storage tier, which reduces I/O overheads for both writing from Spark and loading the data into CGE.

After some pre-determined number of batches are processed, we load the RDF data from several batches into CGE, where we first construct a graph from the network of user pairs that have mentioned each other. We then use the Cray-developed built-in graph function (BGF) extensions to SPARQL to run community detection. By accumulating RDF data containing an edge list of the users who mention other users over a larger window of time, the input contains more relationships on which to build the communities.

Figure 7 shows the Spark code to generate the RDF data used later in CGE to construct the network or graph of users who mention each other. We first extract from the distributed streaming tweet records the users who mention other users in their tweet. The userMention function takes as input a DStream of Tweets and outputs a DStream of UserMention. Next, we further restrict to users who have mentioned each other. Figure 8 shows how this can be done in Spark using the intersection of two sets. This intersection corresponds to the pairs of users who have bidirectional edges between them. User A has mentioned User B, and User B has mentioned User A, and thus we infer that User A knows User B.

If we run the workflow as initially designed only using Spark, we are limited to looking at communities only using relationship data within a single batch interval. Although the join operation used to compute the intersection in Spark performs reasonably well for small joins, as the batch window is increased, even this join becomes problematic. The biggest motivation for moving the join operation for generating the network of users who know each other from the Spark pipeline to a SPARQL query run using CGE is that we can now look at communities over much larger time intervals.

Figure 9 shows the SPARQL code used to construct the network of users who mention each other and also to run community detection. The results from this query are the user community assignments, and for current implementation of CGE these are written to a user-specified directory on Lustre. We are currently extending CGE to provide the option of writing results to the HDFS as well.

Overall workflow performance is also a significant motivator to use both Spark and CGE for performing this workflow, rather than just performing the full workflow in Spark. CGE performance for running the community detection portion of the workflow (Label Propagation) is more than 10 times faster than the Spark GraphX Label Propagation algorithm when run on the same number of nodes for even moderately sized graphs (one million edges). This performance difference grows substantially with larger graphs. We provide some detailed timings of Spark and CGE performance and the time required to switch between frameworks in the following section.

A. Workflow Launching Details

In this section we walk through the components of the workflow in detail and show from a user perspective how each of the application components is launched as well as how the resource manager manages these requests and allocates resources as the workflow progresses.

The workflow is designed to connect to Twitter to download live tweets and process them in real time from Twitter4j [13] or to reprocess historical data that has been archived. To capture historical data, a Java process which downloads data from Twitter runs continuously on a login node. This Java process receives the tweets as JSON records and appends each text to a file. Every hour, the current file is closed and gzipped, and a new file is started. The archived data resides on Lustre and is stored as timestamped events. The workflow allows the reprocessing of data by starting at a given timestamp and streaming data forward.

To start the Spark component of the workflow, a bash script (`run`) is used to launch both the real-time analytics pipeline and the batch analytics pipeline. The following are inputs to the bash script.

```
Input Arguments
 1. lane: fast or slow
 2. start datetime, eg "2015-01-01"
 3. files per batch
 4. max batches
 5. total number of executors cores
 6. executor mem, eg 128g
```

In the following example, we specify 24 files per batch, running 10 batches, requesting 1,024 executor-cores and 128 GB of memory per executor.

```
./run slow "2015-11-01" 24 10 1024 128g
```

This processes approximately 10 days' worth of our internally archived tweet data. The full Twitter firehose

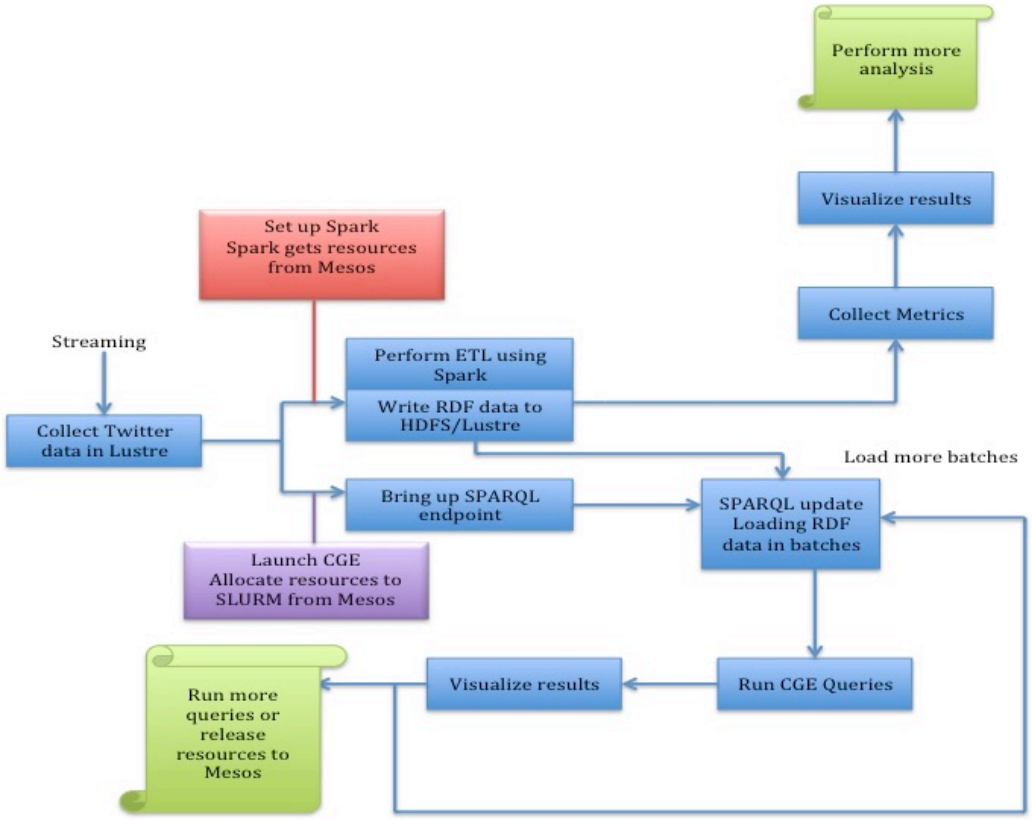

Figure 6. Parallel workflow for social network analytics example

is about 600 million tweets per day. For our experiments we are limiting ourselves to less than one percent of the firehose. The tweets downloaded are based on a list of search keywords. For our internally archived repository, we have recorded approximately two million tweets per day. Thus for the example above, we are processing approximately 20 million tweets combined for the 10 batches.

The fast lane corresponds to the real-time analytics pipeline and the slow lane corresponds to the batch analytics pipeline. Both lanes can be run simultaneously, although they will then be sharing system resources. For this demonstration, we modified the slow lane so that the Spark component of the batch analytics pipeline also writes out RDF data containing the (user, user.mentions) pairs per batch, thus allowing us to later load this data into CGE for further processing.

The `run` bash script then invokes a second script which is where `spark-submit` is initially called. `spark-submit` starts executing Main. Inside Main the Spark context is set up, and any parameter settings used to configure and optimize Spark are set. Two examples of user-provided Spark configuration parameters passed into Main are parameters for setting the total number of Spark executor cores to use (spark.cores.max) and the size of executor memory (spark.executor.memory).

When a Spark context is set up, Spark registers as a framework with Mesos. The Mesos master then gives resource offers to Spark. Spark decides if the resources offered by Mesos satisfy the resource requirements for launching its tasks. If the resource requirements match, Spark accepts the resources from Mesos and coordinates with the Mesos Master to launch its tasks on the Mesos slaves.

For this workflow the core of Main mostly consists of initializing the StreamingContext, specifying the pipeline to execute, and starting the streaming. More information on Spark Streaming can be found in the Spark Users Guide [14].

For the workflow described in Figure 5, we then shut down Spark, releasing the resources back to Mesos. Next we either can load data directly from HDFS into CGE or execute a simple script to pull data from HDFS to Lustre, concatenating all of the distributed RDF files into one large

```
1
2     type UserMention = (UserId, UserId)
3
4     def userMention( tweets: DStream[Tweet]) : DStream[UserMention] = {
5        val userMention = tweets
6        .flatMap { tweet => tweet.mentions
7       .map(mention => (tweet.user, mention))}
8       .filter(t => t._1 != t._2) // no self-mentions
9       .distinct
10
11      val predicate = "<http://mentions>"
12      val endTag = "."
13      val URIfront = "<http://"
14      val URIback = ">"
15      val add_space = "␣"
16
17      val userMentionRDF = userMention.map(edge => URIfront + edge._1 + URIback + add_space
18                + predicate + add_space + URIfront + edge._2 + URIback + add_space + endTag)
19
20      // This will write data to a directory on HDFS to be resident on the SSDs
21      userMentionRDF.saveAsTextFiles("hdfs:/All_SSD/userMentionRDF")
22
23      // This will write data to the users HDFS home directory
24      userMentionRDF.saveAsTextFiles("userMentionRDF")
25
26      // One can also write the generated RDF data directly the Lustre filesystem
27      //userMentionRDF.saveAsTextFiles("file:///mnt/lustre/kristyn/userMentionRDF")
28
29      userMention
30  }
```

Figure 7. Spark code using Scala API for generating and writing RDF data from Tweet DSream

```
1
2     type UserKnows = (UserId, UserId)
3
4     def userKnows( userMention: DStream[UserMention] ) : DStream[UserKnows] = {
5        val ab = userMention
6        val ba = userMention.map(_.swap)
7        val userKnows = ab.intersection(ba)
8        userKnows
9     }
10
11    def communityDetection(userKnows : RDD[UserKnows]) : RDD[UserComm] = {
12       //  Repartition userKnows before constructing graph
13       userKnows.repartition(512)
14       val userGraph: Graph[ UserId, Int ] = Graph.fromEdgeTuples(userKnows, 0L)
15       val groups: Graph[UserId, Int] = LabelPropagation2.runPregel(userGraph)
16       val userComm = groups.vertices
17       userComm
18    }
```

Figure 8. Spark code using Scala API for selecting edges from the userMention DStream

file called *dataset.nt*. This is the most efficient method for building very large databases for CGE, but CGE can also directly load files from HDFS, saving the extra copy.

Next we start up CGE using the `cge-launch` script.

```
cge-launch
  -d (path to dir where RDF data resides)
  -o (path to dir on Lustre to store results)
```

Note that to load the RDF data directly from HDFS, we can provide a graph.info file to CGE to indicate where the files are stored.

```
Example for contents of graph.info file
```

```
1
2  PREFIX cray:   <http://cray.com/>
3  PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
4
5  SELECT ?vertex ?comm
6  WHERE{
7  CONSTRUCT {
8       ?s ?weight ?o  .
9  } WHERE {
10   ?s <http://mentions> ?o .
11   ?o <http://mentions> ?s .
12   BIND("1"^^xsd:integer as ?weight)
13 }
14 # Invoke community detection algorithm (Label Propagation)
15 # and run for 20 iterations
16 INVOKE  cray:graphAlgorithm.community_detection_LP(20)
17 PRODUCING ?vertex ?comm
18 }
19 ORDER BY ?comm
```

Figure 9. SPARQL query for generating graph and running community detection

```
hdfs:/All_SSD/UserMentionRDF-1459988200000
hdfs:/All_SSD/UserMentionRDF-1459988400000
hdfs:/All_SSD/UserMentionRDF-1459988600000
hdfs:/All_SSD/UserMentionRDF-1459988800000
hdfs:/All_SSD/UserMentionRDF-1459989000000
```

Here `cge-launch` works with Marathon/Mesos to request resources, and Slurm starts up CGE. Marathon applications submitted by Slurm accept resources from Mesos based on the resource availability. Once the resources are available the Slurm daemons are started as Mesos slave tasks as illustrated in Figure reffig:UsingMarathon.

Once CGE starts, it builds the database and waits for queries. We then start up the CGE command line interface `cge-cli` to run queries.

```
cge-cli sparql SNA_community.rq
```

Here the file "SNA_community.rq" contains the SPARQL code listed in Figure 9. The results for the query are written to Lustre. We can then use the following command to shut down CGE and release all resources back to Mesos.

```
cge-cli shutdown
```

For the workflow described in Figure 6, we keep the Spark framework running and continue streaming tweet data through the Spark pipeline. CGE can be started at any point on a different set of nodes, either with an empty database or with an initial set of RDF data obtained from an initial set of batches processed. We can then pipeline the insertion of new RDF data via the CGE command-line front end using the SPARQL LOAD operator. The generation of the update queries provided to `cge-cli` can be automated to look for new files being written to HDFS and the Spark Streaming pipeline process runs. An example of what the SPARQL update using the LOAD operator would look like is shown below.

```
LOAD <hdfs:///All_SSD/(new RDF dir/filename)>
```

We are able to make some interesting observations by examining the output logs from the CGE runs. As we increased the window of data analyzed, the premise was that we would better capture relationships between users. To look at this we ran the community detection algorithm using CGE analyzing the first 24 hours of twitter data, then analyzing five days of data, then analyzing 10 days, and finally analyzing 30 days. All the comparisons were made using 16 nodes.

For the 24-hour period the number of (user.user.mention) pairs or RDF triples generated using Spark and written to HDFS was 2,399,916. On 16 nodes this took 3.8 seconds to build the database from the RDF data in CGE. Running the query, which includes computing the network graph of users who know each other (existence of bi-directional edges connecting users), running the community detection algorithm, and sorting the users by community, only took 2.9 seconds. The resulting network graph and communities from a 24-hour period was also interesting in that from the large number of initial input RDF triples, which each contain a record of where a user mentioned another user in a tweet, the number of users who "know" each other based on our criteria was significantly smaller. The size of the graph on which community detection was run was 60,853 vertices and 76,896 edges. The largest community found contained only 152 users (vertices in the graph), with many users not yet assigned to communities after 20 iterations.

If we then look at a window of five days, the number of (user.user.mention) pairs or RDF triples generated using Spark and written to HDFS was 12,033,038. The resulting size of the graph on which community detection was run was 281,536 vertices and 404,866 edges. The largest community

found now contains 442 users. There are still a lot of users not yet assigned to communities, but by just looking at the average number of edges per vertex in the input graph, we can see that more relationships are being captured by analyzing data over larger windows of time. Also note that although we are analyzing five times as much data, the time to build the database is still only 6.47 seconds, and the query took 3.2 seconds.

For the 10-day window, the number of RDF triples generated using Spark and written to HDFS was 19,469,132 and the resulting size of the graph was 541,804 vertices and 861,476 edges. The total startup time, which includes reading in the RDF files and building the database, took only 12.2 seconds, and the query took 3.74 seconds. As the size of the window increases, we continue to capture both more users and more connections between users.

The number of RDF triples generated in our 30-day window was 54,990,937, which is slightly less than 2 million triples per day. For our 30-day window, the number of RDF triples we are processing is still about 10 times smaller than the 600 million tweets per day for the full Twitter firehose, so the capability to analyze even larger graphs is still highly desirable. For the 30 day window we collected some additional performance data, looking at the overhead from loading data from HDFS or Lustre, and then looking at the performance relative to Spark for running the community detection portion of the workflow.

For the 30-day window, the total startup time was 17.63 seconds when loading the RDF files from HDFS, of which 5.44 seconds of this time was the RDF load time. We also looked at the total startup overhead if instead we first copied all the data files out of HDFS onto Lustre and collected these into a single large file. In this case the total startup time increased slightly to 18.72 seconds, of which 5.5 seconds was the RDF load time.

The size of the network graph (1,422,267 vertices, 2,775,008 edges) increased in both the total number of users and the number of connections. The time to run community detection again only increased slightly, up to 5.54 seconds. We then compared this time to the original Spark-only workflow where we ran the community detection in Spark using GraphX within a single batch. For the Spark implementation of community detection (Label Propagation), we are using a more optimized version of this routine than is provided in the Spark distribution. Our implementation includes optimizations made to the Spark PageRank algorithm but not yet folded into the distributed version of LabelPropagation provided with GraphX. We refer to the optimized version as LabelPropagation2.

The time for running the community detection using Spark is dominated by the time required to repartition and then convert the UserKnows RDD (distributed edgelist) to the GraphX Graph distribution; see Figure 8, lines 13 and 14. This repartitioning and graph construction took 520 seconds for two million edges. This builds the VertexRDD and EdgeRDD GraphX data structures used to represent and operate on the graph. Once the graph was constructed, each iteration of the Label Propagation algorithm only took 11 seconds, but the total runtime was 11.4 minutes, or 684 seconds. To run this component in CGE, the combined time of loading the RDF and building the database (17.63 seconds) and the query time (5.54 seconds) is 23 seconds, almost a 30x speedup.

Table I
RDF AND USER NETWORK GRAPH STATISTICS

Window	RDF Triples	Vertices	Edges	Average Edges per Vertex
24 hours	2,399,916	60,853	76,896	1.26
5 days	12,033,038	281,536	404,866	1.44
10 days	19,469,132	541,804	861,476	1.59
30 days	54,990,937	1,422,267	2,775,008	1.95

For the network graphs generated using the smaller time windows (24-hour period) it is reasonable to just have these computations remain in Spark, but as was discussed earlier, the smaller windows limit the possible connections found between users. Using both CGE and Spark enables more detailed analysis of the data over larger windows of time.

V. FUTURE PLANS

Apache Myriad is an open-source project that also enables the co-existence of Apache Hadoop and Apache Mesos on the same analytics platform. Myriad provides this ability by running Hadoop YARN as a Mesos framework, and thereby enables Hadoop workloads and other Mesos frameworks to run side by side. Our plan is to investigate using Myriad in place of the Cray-developed solution (which uses Marathon to flex up YARN sub-clusters under Mesos).

Using Slurm's Elastic Computing Cloud feature, while moderately successful, did not provide the level of system management integration that we initially hoped for. We now believe that creating a Marathon framework capable of performing the Aries setup required for CGE/DMAPP would allow more precise control of system resources and provide the ability to more effectively clean up error cases that may arise when running CGE tasks.

We also plan to continue to develop a more general HPC framework beyond the subset of components needed to run CGE. This will entail bringing over more of the HPC software stack running on our XC40 systems to the Athena platform, such as the full Cray Compiler Environment (CCE) and Cray MPI message passing library.

VI. CONCLUSION

In this paper, we described our early experiences running mixed workloads across the Spark, CGE and Hadoop frameworks on Cray's Athena analytics platform. To accomplish

this we created the necessary interfaces for the different resource managers to interoperate with Mesos. We then illustrated how our solution provided an an efficient utilization of resources using a sample workload. We also showed that by providing a single system capable of running both a traditional analytics (big data) and HPC workload we were able to improve overall performance for this workflow, not only improving time to solution, but enabling more detailed analysis of the data over a longer time period.

ACKNOWLEDGMENT

The authors would like to thank Mike Ringenburg, Richard Korry, Andy Kopser, Bill Sparks and Eric Lund for taking time to review the paper and provide valuable feedback. We would like to express our special thanks to Eric Lund for helping with CGE integration with HDFS, having this working in time for the paper, and also his work on the Mesos/Slurm integration for launching CGE on the Athena system. We would also like to thank Mike Hinchey for his assistance with the social networking analytics workflow.

REFERENCES

[1] T. White, *Hadoop: The Definitive Guide* , 2nd ed. O'ReillyMedia, Inc., October 2010.

[2] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm\?id\=1863103.1863113

[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[5] J. Shafer, "The Hadoop distributed filesystem: Balancing portability and performance," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2010.

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm\?id\=1251254.1251264

[7] J. Sparks, H. Pritchard, and M. Dumler, "Cray framework for hadoop for the cray XC30," in *Cray User Group Conference (CUG '14)*, Lugano, Switzerland, 2014.

[8] H. Ayyalasomayajula, "An Evaluation of the Spark Programming Model For Big Data Analytics," Master's thesis, University of Houston, 2015. [Online]. Available: https://uh-ir.tdl.org/uh-ir/handle/10657/1130?show=full

[9] K. Maschhoff, R. Vesse, and J. Maltby, "Porting the Urika-GD graph analytic database to the XC30/40 platform," in *Cray User Group Conference (CUG '15)*, Chicago, IL, 2015.

[10] T. Johnson, "Coarray C++," in *7th International Conference on PGAS Programming Models*, Edinburgh, Scotland, 2013.

[11] M. Hinchey, "Implementing a social-network analytics pipe using Spark on Urika-XA," in *Cray User Group Conference (CUG '15)*, Chicago, IL, 2015.

[12] *Hadoop 2.7 Archival Storage, SSD & Memory*. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html

[13] *Twitter4j*, 2016 (accessed April 8, 2016). [Online]. Available: http://twitter4j.org

[14] *Spark Lightning-fast cluster computing*, 2016 (accessed April 8, 2016). [Online]. Available: http://spark.apache.org/docs/latest/streaming-programming-guide.html