Building Pipelines for Heterogeneous Execution Environments for Big Data Processing

Dongyao Wu, Liming Zhu, Xiwei Xu, Sherif Sakr, Daniel Sun, and Qinghua Lu, NICTA

The Pipeline61 framework supports the building of data pipelines involving heterogeneous execution environments. It reuses the existing code of the deployed jobs in different environments and provides version control and dependency management that deals with typical software engineering issues.

Researchers have developed big data processing frameworks such as MapReduce and Spark to tackle the ever larger datasets distributed on large-scale clusters. These frameworks significantly reduce the complexity of developing big data programs and applications. In practice, many real-world scenarios require pipelining and integration of multiple data-processing and data-analytics jobs. For example, an image-analyzing application requires many preprocessing steps such as image parsing and feature extraction, and the core machine-learning algorithm is only one component in the whole analytic flow. However, the developed jobs aren’t easily integrated or pipelined to support more complex data analytics scenarios. To integrate data jobs executing in heterogeneous execution environments, developers must write much glue code to get data into and out of those jobs. According to Google researchers, a mature real-world system might contain only 5 percent machine-learning code and 95 percent glue code.

To support the integration and pipelining of big data jobs, researchers have proposed higher-level pipeline frameworks such as Crunch, Pig, and Cascading. (For more on pipeline frameworks, see the sidebar.) Most of these frameworks are built on top of a single data-processing execution environment and require the pipelines to be written in their specifically defined interfaces and programming paradigms. In addition, pipeline applications will keep evolving to address new changes and requirements. These applications could also contain various legacy components that require different execution environments. So, maintaining and managing such pipelines becomes complicated and time-consuming.

The Pipeline61 framework aims to reduce the effort for maintaining and managing data pipelines across heterogeneous execution contexts, without major rewriting of the original jobs. It integrates data-processing
EXISTING PIPELINE FRAMEWORKS

Most frameworks for building pipelined big data jobs are built on top of a data-processing engine (for example, Hadoop) and use an external persistent service (for example, Hadoop Distributed File System) to exchange data. Table A compares the most important pipeline frameworks for big data jobs.

Crunch (https://crunch.apache.org) defines its own data model and programming paradigm to support writing the pipeline and executing pipeline jobs on top of MapReduce and Spark. Pig uses a dataflow-based programming paradigm to write ETL (extract, transform, and load) scripts, which are translated into MapReduce jobs during execution. Cascading provides an operator-based programming interface for pipelines and supports executing Cascading applications on MapReduce. Flume (https://flume.apache.org) was originally designed for log-based pipelines. It lets users create a pipeline using configuration files and parameters. MRQL (MapReduce Query Language; https://mrql.incubator.apache.org) is a general system for query and optimization on top of various execution environments such as Hadoop, Spark, and Flink (https://flink.apache.org). Tez is an optimization framework based on directed acyclic graphs; it can optimize MapReduce pipelines written in Pig and Hive.

Unlike those frameworks, Pipeline61 (see the main article)

- supports pipelining and integration of heterogeneous data-processing jobs (using MapReduce, Spark, and scripts);
- reuses existing programming paradigms rather than requiring developers to learn new ones for writing analytic algorithms; and
- provides automated version control and dependency management to facilitate historical traceability and reproducibility, which are important for continuously developing pipelines.

Like Pipeline61, the Apache Object Oriented Data Technology (OODT; http://oodt.apache.org) data grid framework lets users capture, locate, and access data from heterogeneous environments. Compared to Pipeline61, OODT provides more general task-driven workflow execution, in which developers must write their own programs to invoke different execution tasks. In contrast, Pipeline61 focuses on deep integration with contemporary big data processing frameworks including Spark, MapReduce, and IPython. Also, OODT uses an XML-based specification for pipelines, whereas Pipeline61 provides programmable interfaces in different programming languages. Finally, OODT maintains general information and metadata for the shared datasets. Pipeline61 provides explicitly defined provenance information for both I/O data and transformations for each task in a pipeline. Subsequently, Pipeline61 natively supports reproducing or reexecuting historical pipelines and even part of the pipelines.

References

<table>
<thead>
<tr>
<th>Framework</th>
<th>Crunch</th>
<th>Pig</th>
<th>Cascading</th>
<th>Flume</th>
<th>MRQL*</th>
<th>Tez</th>
<th>OODT*</th>
<th>Pipeline61</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution context</strong></td>
<td>MapReduce, Spark</td>
<td>MapReduce</td>
<td>MapReduce (Spark and Flink in the future)</td>
<td>Fume</td>
<td>MapReduce, Spark, Flink, and so on</td>
<td>MapReduce</td>
<td>Grid</td>
<td>MapReduce, Spark, and scripts</td>
</tr>
<tr>
<td><strong>Pipe definition</strong></td>
<td>Operators</td>
<td>Queries</td>
<td>Operators</td>
<td>Configurations</td>
<td>SQL</td>
<td>Edges</td>
<td>Elements in workflow</td>
<td>Pipes (various execution contexts)</td>
</tr>
<tr>
<td><strong>Pipeline connection</strong></td>
<td>Source and target</td>
<td>Query composition</td>
<td>Branches, joins, and mapTo</td>
<td>Source and sink</td>
<td>SQL</td>
<td>No</td>
<td>Control logic</td>
<td>Connecting notations</td>
</tr>
<tr>
<td><strong>Data model</strong></td>
<td>PCollection</td>
<td>Schema</td>
<td>Pipe</td>
<td>No abstraction</td>
<td>Schema</td>
<td>Vertex</td>
<td>Various</td>
<td>Depends on the execution context</td>
</tr>
<tr>
<td><strong>Programming languages</strong></td>
<td>Java</td>
<td>Pig</td>
<td>Scala or Java</td>
<td>Configuration file</td>
<td>SQL</td>
<td>Java</td>
<td>XML and others</td>
<td>Scala or Java</td>
</tr>
</tbody>
</table>

* MRQL stands for MapReduce Query Language; OODT stands for Object Oriented Data Technology.
components that execute in various environments, including MapReduce, Spark, and scripts. It also reuses the existing programs of data-processing components as much as possible so that developers don’t need to learn a new programming paradigm. In addition, it provides automated version control and dependency management for both data and components in each pipeline instance during its life cycle.

**A Motivating Example**

A suspicion detection system motivated our research; Figure 1 shows the system’s data-processing pipeline. Input data comes from different departments and organizations; the collected data might have different formats with different schemas. CSV stands for comma-separated values, JSON is JavaScript Object Notation, MR is MapReduce, and HDFS is Hadoop Distributed File System.

![Figure 1](image1.png)

**FIGURE 1.** A suspicion detection system’s data-processing pipeline. Input data comes from different departments and organizations; the collected data might have different formats with different schemas. CSV stands for comma-separated values, JSON is JavaScript Object Notation, MR is MapReduce, and HDFS is Hadoop Distributed File System.

CSV (comma-separated values), text, or JSON (JavaScript Object Notation)—with different schemas.

Owing to different technical preferences at different pipeline stages, different data scientists or engineers might develop data-processing components using different techniques and frameworks, such as IPython, MapReduce, R, and Spark. Some legacy components might also be implemented with Bash scripts or third-party software. So, it’s complicated and tedious to manage and maintain those pipelines, which involve heterogeneous execution environments and are updated throughout the life cycle. Replacing the old pipeline framework with a new one is also expensive, perhaps even unaffordable. In the worst-case scenario, developers might need to reimplement all the data-processing components from scratch.

In addition, as we mentioned before, pipeline applications keep evolving and being updated to deal with system changes and new system requirements. For example, new data sources might be added as new inputs, existing data sources’ formats and schemas might change, and the analytic components might be upgraded to improve efficiency and accuracy. All these can cause continuous changing and updating of the pipeline components. One challenge is to provide traceability and reproducibility during pipeline evolution. Pipeline developers might want to check a pipeline’s history to compare the effects before and after updates. Also, each data-processing component should be able to roll back to a previous version when necessary.

**Pipeline61**

To meet these challenges, Pipeline61 employs three main components (see Figure 2). The **execution engine**
triggers, monitors, and manages pipeline execution. The data service provides a uniformly managed data I/O layer that performs the tedious work of data exchange and conversion between various data sources and execution environments. The dependency and version manager automates version control and dependency management for the data and components in the pipeline. A management API lets developers test, deploy, and monitor pipelines by sending and receiving messages.

The Pipe Model
Pipeline61 represents every pipeline component as a pipe, which has these associated entities:

- A pipe’s name must be unique and is associated with the pipe’s management information. The name can contain the namespace information.
- A pipe’s version number automatically increases to represent different versions of that pipe. Users can specify a version number to execute a specific version.
- The pipeline server manages and maintains the pipe. The pipe needs the pipeline server’s address information so that it can send notification messages back to the pipeline server during execution.
- The URLs of the input path and output path contain the protocol and address of the pipes’ I/O data. The protocol represents the persistent system types such as HDFS (Hadoop Distributed File System), JDBC (Java Database Connectivity), S3 (Amazon Simple Storage Service), file storage, and other data storage systems.
- The input format and output format specify the I/O data’s reading and writing formats.
- The execution context specifies the execution environment and any other information the execution framework requires.

Execution contexts are associated with different data-processing frameworks. Pipeline61 currently has three major execution contexts:

- The Spark execution context contains a SparkProc attribute that provides the transformation function from input RDDs (resilient distributed datasets) to output RDDs or from the input DataFrame to the output DataFrame for SparkSQL.
- The MapReduce execution context contains a few structured parameters to specify the Mapper, Reducer, Combiner, and Partitioner for a MapReduce job. Other parameters could be added as key-value parameters.
- The shell execution context contains a script file or inline commands for execution. Python and R scripts are subclasses of shell pipes with more inputs and outputs controlled by the data service. One limitation of a shell pipe is that developers must manually deal with data conversion for the input and output.

Figure 3 shows how to simply write a SparkPipe. Developers wrap the Spark RDD (resilient distributed datasets) functions with the SparkProc interface, then use SparkProc to initiate a SparkPipe object.

Pipline61 lets developers seamlessly integrate different types of pipes at the logical level. It provides method notations to connect pipes to form pipelines. After pipes are connected, the previous pipes’ output becomes the following pipes’ input. We show a more concrete example later, in our case study.

The Execution Engine
The execution engine has three components.

The pipeline server back end contains message handlers that receive and process messages from users and running tasks. Users can send messages to submit, deploy, and manage their pipeline jobs and dependencies. Running tasks can send messages reporting their runtime status. Runtime messages can also trigger the events for scheduling and recovering processes during execution.

The DAG (directed acyclic graph) scheduler traverses the task graph...
Task launchers launch execution processes for pipes. Currently, Pipeline61 employs three types of task launchers:

- The Spark launcher initiates a subprocess as the driver process to execute a Spark job. It captures the notifications of runtime status and sends the notifications back to the pipeline server for monitoring and debugging.
- The MapReduce launcher initiates a subprocess to submit the MapReduce job specified by the pipe. The subprocess waits until the job has succeeded or failed before sending the execution states back to the pipeline server.
- The shell launcher creates a sequence of channeled processes to handle the shell scripts or commands specified by the shell pipes. Once the sequence of processes succeeds or any of them fails, the related state messages will be sent to the pipeline server.

Developers can implement new task launchers to support new execution contexts.

- through an API provided by the execution framework (such as Hadoop and Spark) or
- by initiating a subprocess and executing the program logic in the launched process (such as with shell scripts, Python, and R).

Theoretically, a process launcher can extend any task that can be started by executing a shell script.

The Data Service

Every pipe executes independently at runtime. A pipe reads and processes the input data according to the input paths and formats, then writes the output into the selected storage system. Managing the data I/O for various protocols and formats is often tedious and error prone. So, the data service performs most of the data I/O work for developers.

The data service provides a collection of data parsers, each of which reads and writes data in a specific execution environment according to the given format and protocol. For example, for a Spark pipe, the data service uses either the native Spark API to load text files as RDD objects or the SparkSQL API to load data from JDBC or JSON files as a Spark DataFrame. For a Python pipe, the data service uses the Python Hadoop API to load CSV files in HDFS and transfers them as a Python DataFrame. Basically, the data service maps data protocols and formats to the concrete data parsers in the specific execution context.

To provide flexibility, we could extend the data service by implementing and registering new types of data parsers. Data-parsing toolkits such as Apache Tika (http://tika.apache.org) can serve as a complementary implementation in the data service.
The Dependency and Version Manager

It’s crucial but complicated for pipeline administrators to manage and maintain the pipelines through their life cycles. To relieve the pain of pipeline management, the dependency and version manager helps users maintain, track, and analyze the historical information of the pipeline data and components.

The dependency and version manager maintains three major types of information for every pipeline. The pipeline execution trace (see Figure 4) maintains the dataflow graph for every execution instance of a pipeline application. Every graph node also contains the metadata for the component in that instance, such as the start time, end time, and execution status.

The pipe dependency trace (see Figure 5a) maintains the historical metadata for different versions of each pipeline component. It stores the dependency information as a tree structure for each component. Metadata stored in the tree includes the name, version, author, time stamp for the latest update, and dependent libraries for execution.

The data snapshot (see Figure 5b) contains the input and output locations and sampled data for every execution instance of each pipeline component.

Using this historical information, Pipeline61 users can analyze the pipeline history and reproduce historical results by rerunning older versions of pipelines.

A Case Study

The following case study shows Pipeline61’s effectiveness and advantages. It involved three data sources from different organizations in three formats, including CSV, text, and JSON. Two groups of data scientists were analyzing the overall datasets with a few manually connected programs written in MapReduce and Python. We introduced our pipeline framework to automate execution of the pipeline job and facilitate its management. Figure 6 shows how we specified the pipeline in Pipeline61.

First, we specified three data mappers—csvMapper, jsonMapper, and textMapper—to process the input data in different formats. We specified three MapReduce pipes by passing the existing Mapper classes as data parsers.

Next, we used the RDD function DataJoinerProc to specify a Spark pipe called dataJoiner to combine the three mappers’ outputs.

Finally, we specified two branches of analysis pipes to consume the output from dataJoiner. Because each analysis branch was interested in different input features, we added a feature extractor before each of the two analysis components. Then, we implemented the last two analysis components as Python and Spark pipes. Eventually, we defined the overall dataflow by connecting the specified pipes, using the connecting notations.

In this scenario, using existing pipeline frameworks such as Crunch and Cascading would have required developers to reimplement everything from scratch following their specific programming paradigms. Such reimplementations are risky and time-consuming. It would have not only eliminated the reuse of existing programs written in MapReduce, Python, or shell scripts but also...
restricted the use of data analysis frameworks such as IPython and R. In contrast, because Pipeline61 focuses on pipelining and managing heterogeneous pipeline components, it can significantly reduce the effort to integrate new data-processing components with existing ones written in legacy code.

Future development and updating of the pipeline will also benefit from Pipeline61’s version and dependency management. For example, if developers want to update a component to a new version, they can sample the component’s latest input and output from the data snapshots history. Then, they can implement and test the new program on the basis of the sampled data to ensure that the new version doesn’t break the pipeline.

Before submitting an updated component to the production environment, developers can specify a new pipeline instance with the updated component and compare its output with the online version to double-check the correctness. Moreover, if a recently updated component shows any errors after deployment, the pipeline manager can easily roll back to a previous version. This is possible because the pipeline server automatically maintains every component’s historical data and dependencies.

These DevOps supports are very meaningful for maintaining and managing pipeline applications, but existing pipeline frameworks (see the sidebar) rarely offer them.

Pipeline61 still has limitations. It doesn’t check the compatibility of data schemas across multiple data-processing frameworks. So far, developers must manually test the input and output of every pipe during pipeline development to ensure that a pipe’s output can be fed into the next pipe. To solve this problem, we plan to use existing schema-matching techniques.

Also, most intermediate results during pipeline execution must be written to underlying physical data storage (such as HDFS) for connecting pipes with different execution contexts and ensuring pipeline component reliability. So, pipeline execution in Pipeline61 is generally slower than in other frameworks, which, as we mentioned before, run independently on a single execution environment without integrating with external systems. One possible way to solve this is to store only the data that’s considered important or required by the developers. However, this would involve a trade-off between reliability and the integrity of history management.

Acknowledgments
The Australian Government’s Department of Communications and the Australian Research Council’s ICT Centre of Excellence Program fund NICTA (National ICT Australia).

Reference
ABOUT THE AUTHORS

DONGYAO WU is a PhD candidate in computer science and engineering at the University of New South Wales and NICTA. His research interests include big data infrastructure, cloud computing, distributed systems, and parallel computing. Wu received an MSc in computer science and technology from the Institute of Software, Chinese Academy of Sciences. Contact him at dongyao.wu@nicta.com.au.

SHERIF SAKR is an associate professor in the Health Informatics department at King Saud bin Abdulaziz University for Health Sciences. He’s also affiliated with the University of New South Wales and NICTA. Sakr received a PhD in computer and information science from Konstanz University. He’s an IEEE Senior Member. Contact him at sherif.sakr@nicta.com.au.

LIMING ZHU is the research director of the Software and Computational Systems Research Program at Data61, which combines NICTA and CSIRO (Commonwealth Scientific and Industrial Research Organisation) researchers. He also holds conjoint professor positions at the University of New South Wales and University of Sydney. His research interests include software architecture, dependable systems, and data analytics infrastructure. Zhu received a PhD in software engineering from the University of New South Wales. He’s a committee member of the Standards Australia IT-015 (system and software engineering) group and IT-038 (cloud computing) group and contributes to ISO/SC7/WG42 on architecture-related standards. Contact him at liming.zhu@nicta.com.au.

DANIEL SUN is a researcher at NICTA and a conjoint lecturer in the School of Computer Science and Engineering at the University of New South Wales. His research interests include system modelling and evaluation, algorithms and analysis, reliability, energy efficiency, and networking in parallel and distributed systems. Sun received a PhD in information science from the Japan Advanced Institute of Science and Technology. Contact him at daniel.sun@nicta.com.au.

XIWEI XU is a researcher in Data61’s Analytics and Architecture Group and an adjunct lecturer at the University of New South Wales. She works on dependability, cloud computing, DevOps, and big data. Xu received a PhD in software engineering from the University of New South Wales. Contact her at xiwei.xu@nicta.com.au.

QINGHUA LU is a lecturer in the China University of Petroleum’s Department of Software Engineering. Her research interests include software architecture, cloud-computing dependability, big data architecture, and service computing. Lu received a PhD in computer science and engineering from the University of New South Wales. Contact her at qinghua.lu@nicta.com.au.