A Pipeline Framework for Heterogeneous Execution Environment of Big Data Processing

[1,2]Dongyao Wu, [1,2]Liming Zhu, [1,2]Xiwei Xu, [1,3]Sherif Sakr, [1,2]Daniel Sun, [1,4]Qinghua Lu

[1]NICTA, Sydney, Australia
[2]School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
[3]King Saud bin Abdulaziz University for Health Sciences, National Guard, Riyadh, Saudi Arabia
[4]College of Computer and Communication Engineering, China University of Petroleum, Qingdao, China

Abstract—**Many real-world data analysis scenarios require pipelining and integration of multiple (big) data processing and analytics jobs, which are often executed in heterogeneous environments, such as MapReduce, Spark, R/Python/Bash scripts. For such a pipeline, a large amount of glue code has to be written to get data across environments. Maintaining and evolving such pipelines are difficult. Existing pipeline frameworks trying to solve such problems are usually built on top of a single environment, and/or require the original job to be re-written against a new APIs or paradigm. In this article, we propose Pipeline61, a framework that supports the building of data pipelines involving heterogeneous execution environments. Pipeline61 reuses the existing code of the deployed jobs in different environments and also provides version control and dependency management that deals with typical software engineering issues. A real-world case study is used to show the effectiveness of Pipeline61.**

Keywords: big data, pipeline, Spark, MapReduce

I. Introduction

Over the past years, big data processing frameworks, such as MapReduce and Spark, have been presented to tackle the ever large datasets distributed on large scale clusters. These frameworks significantly reduce the complexity for the development of big data programs and applications. In practice, many real-world scenarios require pipelining and integration of multiple data processing and analytics jobs. For example, an image analyzing application requires many pre-processing steps such as image parsing and feature extraction while the core machine learning algorithm is only one component within the whole analytic flow. However, the developed jobs are not easy to be integrated or pipelined to support more complex data analytics scenarios. To integrate data jobs executed in heterogeneous execution environments, a large amount of glue code has to be written to get data into and out from the deployed jobs being integrated. According to Google's report, a mature system in the real world might contain only 5% machine learning code and (at least) 95% glue code [1].

To support the integration and pipelining of big data jobs, many higher-level pipeline frameworks have been proposed, such as Crunch [2], Pig [3], and Cascading [4]. Most of these existing data pipeline 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 would keep evolving to address the new changes and requirements. These pipeline applications could also contain various legacy components that need to be executed on different execution environments. Therefore, the maintenance and management of such pipelines become very complicated and time-consuming.

In this article, we present the framework - **`Pipeline61`** which aims to reduce the effort for maintaining and managing data pipelines across heterogeneous execution contexts without major rewriting of the original jobs. In particular, **`Pipeline61`** 1) integrates data processing components that are executed on various environments, including MapReduce, Spark and Scripts; 2) re-uses the existing programs of data processing components as much as possible so that developers do not need to learn a new programming paradigm; 3) provides automated version control and dependency management for both data and components in each pipeline instance during its lifecycle.

II. Motivating Example

Our work is motivated by a realistic suspicion detection system, the data processing pipeline of which is shown in Figure 1. In this scenario, multiple input data are collected from different departments and organizations, such as vehicle registration records provided by government's road services, personal income reports provided by government's tax office, or travel histories provided by airline companies. For different data sources, the collected data may have different formats (e.g. CSV, text and JSON) with different schemas.

On one hand, due to different technical preferences in different stages of the pipeline, data processing components might be developed by different data scientists or engineers 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. As a result, it is complicated and tedious to manage and maintain those pipelines which involve heterogeneous execution environments and keep being updated through the lifecycle. Using a new pipeline framework to replace the old one is also cost-expensive and even un-affordable. In the worst case scenario, developers might need to re-implement all the data processing components from scratch.

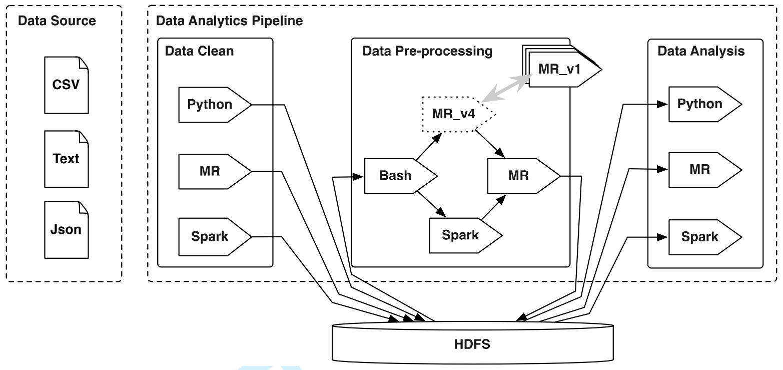


Figure 1. The data process pipeline of a real-world suspicion detection system

On the other hand, pipeline applications keep evolving and being updated to deal with continuous changes and requirements to the system. For example, new data sources could be added as new inputs; existing data sources might introduce changes to their formats and schemas; the analytic components can also be upgraded to improve the efficiency and accuracy. All of these can cause continuous changing and updating of the components in the pipeline. One challenge is to provide both traceability and reproducibility during the evolving process of the pipelines. Pipeline developers may want to check the history of a pipeline to compare the effects before and after updates. In addition, each of the data processing components should be able to roll back to a previous version when it is required.

III. PIPELINE61

To bridge the gap, we present our framework `Pipeline61`.The architecture of the framework is illustrated in Figure 2. There are three main components within the framework: Execution Engine, Dependency &Version Manager and Data Service. Current implementation of `Pipeline61` supports the execution of data processing components based on MapReduce, Spark and Bash scripts. The Execution Engine is responsible for triggering, monitoring and managing the execution of pipelines. Data Service provides a uniformly managed data I/O layer that manages the tedious work of data exchange and conversion between various data sources and execution environments. The Dependency and Version Manager provides a few mechanisms to automate the version control and dependency management for both data and components within the pipelines. The pipeline framework is interactive through management APIs, which allow developers to test, deploy and monitor the pipelines via sending and receiving messages.

Pipe Model

In `Pipeline61`, every component in a pipeline is represented a ***Pipe,*** which has the following attributes:

- ***Name***: The name of the pipe, which must be unique and is associated with all the management information of the pipe. The pipe's name may contain the namespace information.
- ***Version***: The version number of the pipe, which automatically increases to represent different versions of the same pipe. Users can also specify the version number to execute a specific version of the pipe.
- ***Pipeline Server***: The address of the associated pipeline server, which is responsible of managing and maintaining the pipe. The pipe needs the address information for sending back the notification message to the pipeline server during execution.
- ***Input Path / Output Path***: URL of the path contains both the protocol and address of the input/output data of the pipes. The protocol represents the persistent system types such as HDFS, JDBC, S3, File and other data storage systems.

- ***Input Format / Output Format***: specifies the exact reading/writing format of the input/output data.
- ***Execution Context***: specifies the execution environment and any other information required by the underlying execution framework.

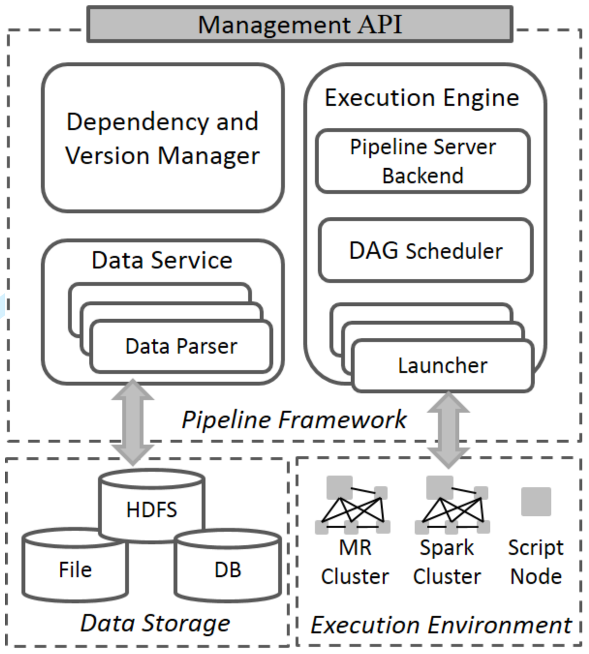


Figure 2. Architecture overview of Pipeline61

The ***ExecutionContext*** attribute is associated to different data processing frameworks. There are three major ***ExecutionContexts*** in current framework:

- ***Spark ExecutionContext*** contains a *SparkProc* attribute which provides the transformation function from input RDDs to output RDDs or from input *DataFrame* to output *DataFrame* for *SparkSQL*;
- ***MapReduce ExecutionContext*** 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.
- ***Shell ExecutionContext*** contains a script file or in-line commands for execution. Python and R scripts are considered as sub-classes of shell pipes with more inputs and outputs controlled by the data service. One limitation of the shell pipe is that it relies on the developers to manually deal with the data conversion for the input and output.

The code snippet below shows how to simply write a *SparkPipe*. Basically, developers just need to wrap the Spark RDD functions with the *SparkProc* interface, then use the *SparkProc* to initiate a *SparkPipe* object.

```
class Wordcount extends SparkProc[String, (String, Int))] {
        def process(in:RDD[String], sc:SparkContext):RDD[(String, Int)] = {
           in.flatmap(_.split(“ ”)).map(w=> (w, 1)).reduceBykey(_ + _)
        }
}
wordcountPipe = new SparkPipe(name = "wordcount", version = "0.0.1", exec = new Wordcount)
```

In `Pipline61`, different types of pipes can be seamlessly integrated at the logical level. Method notations are provided to connect pipes as pipelines. After being connected with others, the output of previous pipes are considered as the input of the following ones. A more concrete example is shown in section IV.

Execution Engine

The execution engine contains three components: Pipeline Server Backend, DAG Scheduler and a bunch of Task Launchers.

Pipeline Server Backend contains a number of message handlers which receive and process the messages sent by both users and running tasks. Users can send messages to submit, deploy and manage their pipeline jobs and dependencies. Tasks of the running pipelines can also send messages to notify their runtime status. Runtime messages can also trigger the events for scheduling and recovering processes during execution.

DAG Scheduler traverses the task graph of a pipeline in a backward manner and submits the tasks to the corresponding environments for execution. A task is scheduled for execution when all of its parent tasks have been successfully computed.

Task Launchers are used to launch execution processes for pipes. Different types of pipes are launched by the corresponding launchers for different execution contexts:

- *Spark Launcher* initiates a sub-process as the driver process to execute the Spark job and captures the notifications of runtime status then send back to the pipeline server for monitoring and debugging purposes.
- *MR Launcher* initiates a sub-process to submit the MapReduce job specified by the pipe. The sub-process waits until the job has succeeded or failed before sending the execution states back to the pipeline server.
- *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.

New task launchers can be implemented to support new execution contexts in two ways: a) through APIs provided by the execution frameworks (such as Hadoop and Spark); b) by initiating of a sub-process and execute the program logic in the launched process (such as Shell scripts, Python and R). Theoretically, any task that can be started by execute a shell script can be extended through a process launcher.

Data Service

Every pipe is executed independently during runtime. A pipe reads and processes the input data according to the input paths and formats then writes the output into the expected storage system. Managing the data IO for various protocols and formats are often tedious and error-prone. Thus, we extract out most of the data IO work from developers by providing a data service in `Pipeline61`.

Data service provides a collection of data parsers, each of which is responsible for reading and writing data in a specific execution environment according to the given format and protocol. For example, for a Spark pipe, the data service uses the native Spark API to load text files as RDD objects or uses SparkSQL API to load data from JDBC or JSON files as Spark DataFrame. For a Python Pipe, CSV files in HDFS are loaded through PythonHadoop API and transferred as Python DataFrame. Basically, Data Service provides the mapping from data protocols and formats to the concrete data parsers in specific execution context.

For the flexibility consideration, data service could be extended by implementing and registering new types of data parsers. Data parsing toolkits such as Apache Tika [12] can be used as complementary implementation in the data service.

Dependency and Version Manager

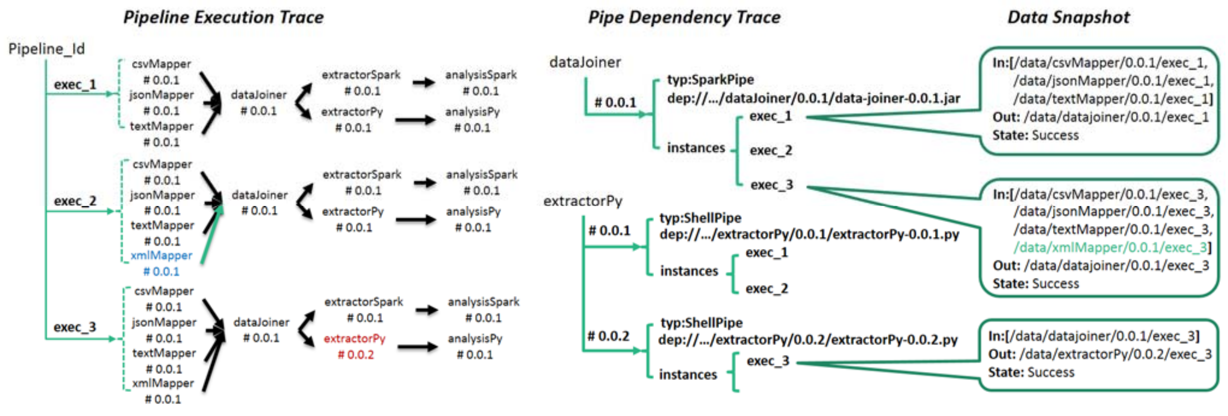

Pipeline Execution Trace
Pipe Dependency Trace
Data Snapshot
Pipeline_Id
exec_1
exec_2
exec_3
csvMapper # 0.0.1
jsonMapper # 0.0.1
textMapper # 0.0.1
xmlMapper # 0.0.1
dataJoiner # 0.0.1
extractorSpark # 0.0.1
extractorPy # 0.0.1
extractorPy # 0.0.2
analysisSpark # 0.0.1
analysisPy # 0.0.1
dataJoiner
# 0.0.1
typ:SparkPipe
dep://.../dataJoiner/0.0.1/data-joiner-0.0.1.jar
instances
exec_1
exec_2
exec_3
extractorPy
# 0.0.1
typ:ShellPipe
dep://.../extractorPy/0.0.1/extractorPy-0.0.1.py
# 0.0.2
typ:ShellPipe
dep://.../extractorPy/0.0.2/extractorPy-0.0.2.py
In:[/data/csvMapper/0.0.1/exec_1, /data/jsonMapper/0.0.1/exec_1, /data/textMapper/0.0.1/exec_1]
Out: /data/datajoiner/0.0.1/exec_1
State: Success
In:[/data/csvMapper/0.0.1/exec_3, /data/jsonMapper/0.0.1/exec_3, /data/textMapper/0.0.1/exec_3, /data/xmlMapper/0.0.1/exec_3]
Out: /data/datajoiner/0.0.1/exec_3
State: Success
In:[/data/datajoiner/0.0.1/exec_3]
Out: /data/extractorPy/0.0.2/exec_3
State: Success

Figure 3. Historical and dependency information maintained in Pipeline61

For realistic pipeline applications, they are continuously evolving and being updated to deal with new changes and requirements. It is always a crucial but complicated work for pipeline administrators to manage and maintain the pipelines through their lifecycles. To relieve the pain for managing evolving pipelines, `Pipeline61` provides a dependency and version manager to help users maintain, track and analyze the historical information of both data and components involved in the pipelines. Dependency and version manager maintains three major information for every pipeline: *Data Snapshot*, *Pipeline Execution Trace*, *Pipe Dependency Trace*, as shown in Figure 3.

- *Data Snapshot*, contains the input, output locations and sampled data for every execution instance of each component in a pipeline.
- *Pipeline Execution Trace,* maintains the data flow graph for every execution instances of a pipeline application. Every node in the graph also contains the meta-data for the component in that instance, such as start, end time and execution status.
- *Pipe Dependency Trace*, maintains the historical meta-data for different versions of each component in a pipeline. The dependency information is stored as a tree structure for each component. Meta-data stored in the tree includes, the name, version, author, timestamp for the latest update, also the dependent libraries for execution.

With those historical information maintained by Dependency and Version Manager, users of **`Pipeline61`** are able to apply analysis on the pipeline history and re-produce the historical results by re-run older versions of the pipelines.

IV. CASE STUDY

In this section, we present a case study that shows the effectiveness and advantages of **`Pipeline61`**. The scenario is that we have three data sources collected from different organizations with three different formats, including CSV, Text and JSON. There are two groups of data scientists who are doing data analysis on the overall datasets with a few manually connected programs written in MapReduce and Python. We would now introduce a pipeline framework to automate the execution and facilitate the management of the pipeline job. The code snippet below shows how this pipeline is specified in **`Pipeline61`**.

```
csvMapper = new MRPipe(name = "csvMapper", version = "0.0.1", mapper = new CSVMapper, inputPath =
"hdfs://127.0.0.1:9001/user/org1/data/csv/")
jsonMapper = new MRPipe(name = "jsonMapper", version = "0.0.1", mapper = new JSONMapper, inputPath =
"hdfs://127.0.0.1:9001/user/org2/data/json/")
textMapper = new MRPipe(name = "textMapper", version = "0.0.1", mapper = new TextMapper, inputPath =
"hdfs://127.0.0.1:9001/user/org3/data/text/")
dataJoiner = new SparkPipe(name = "dataJoiner", version = "0.0.1", exec = new DataJoinerProc)
featureExtractorPy = new SparkPipe(name = "extractorPy", version = "0.0.1", exec = new ExtractorPy)
featureExtractorSpark = new SparkPipe(name = "extractorSpark", version = "0.0.1", exec = new ExtractorSpark)
analysisPy = new ShellPipe(name = "analysisPy", version = "0.0.1", script = "file:///user/dev/analysisPy.py")
analysisSpark = new SparkPipe(name = "analysisSpark", version = "0.0.1", exec = new SparkAnalysisProc)
pipeline = ((csvMapper, jsonMapper, textMapper) ->: dataJoiner) :-> (featureExtractorPy :-> analysisPy,
featureExtractorSpark :-> analysisSpark)
PipelineContext.exec(pipeline)
```

We firstly specify three data mappers, *csvMapper*, *jsonMapper* and *textMapper*, to process the input data in different formats. These three MapReduce pipes are specified by passing the existing Mapper classes as data parsers. For the second phase of the pipeline, a Spark Pipe named *dataJoiner*, is specified with the RDD function *DataJoinerProc* to join the three outputs from previous mappers as a whole. In the last step, two branches of analysis pipes are specified to consume the output from the *dataJoiner*. Because each of the analysis branches is interested in different features of the input, two feature extractors are added before each of the actual analysis components, respectively. Then, the last two analysis components are implemented as Python and Spark Pipes. Eventually, the overall data flow is defined by connecting all the specified pipes together using the connecting notations.

In this scenario, using existing pipeline frameworks such as Crunch and Cascading requires developers to re-implement everything from scratch following their specific programming paradigms. It not only wastes the re-usage of existing programs written in MapReduce, Python or Shell Scripts, but also restricts utilizing data analysis frameworks like IPython and R.

In contrast, **`Pipeline61`** does not re-invent new programming paradigm. Instead, it focuses on pipelining and management of heterogeneous components in the pipelines. Thus, it can significantly reduce the effort to integrate with existing data processing components with legacy code, for which the task of re-implementing everything is risky and time-consuming.

The future developing and updating processes of the pipeline also benefit from the version and dependency management of **`Pipeline61`**. For example, if developers want to update one of the components to a new version, they can sample the latest input and output of the component from the data snapshots history. Then, developers can implement and test the new program based on the sampled data to make sure that the new version does not break the pipeline. Before submitting the updated component to the production environment, the developer 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 as all the historical data and dependencies of every component are automatically maintained by the pipeline server. These DevOps supports are very meaningful for the actual maintenance and management of pipeline applications but rarely offered by existing pipeline frameworks.

V. Existing Pipeline Frameworks

Most of the current frameworks for building pipelined big data jobs are built on top of a data processing engine (e.g. Hadoop), and use an external persistent service (e.g. HDFS) for exchanging data. Table 1 provides an overview and compares the most important pipelining frameworks of big data jobs. Among them, Crunch is a pipeline framework that defines its own data model and programming paradigm to support writing the pipeline and executing pipeline jobs on top of both MapReduce and Spark. Pig uses a data-flow based programming paradigm to write ETL scripts, which is translated into MapReduce jobs during execution. Cascading provides an operator-based programming interfaces for pipelines and supports executing Cascading applications on MapReduce. Flume [5] is originally designed for log-based pipelines. It allows creating a pipeline using configuration files and parameters. MRQL [6] is a general system for query and optimization on top of various execution environments such as Hadoop, Spark and Flink [7]. Tez [8] is a DAG-based optimization framework, which can optimizes MapReduce pipelines written in Pig and Hive.

Compare to the frameworks above, `Pipeline61` provides three main functionality: 1) support pipelining and integration of heterogeneous data processing jobs (MapReduce, Spark and scripts); 2) re-use the existing programing paradigms rather than requiring developers to learn new ones for writing analytic algorithms; 3) provide automated version control and dependency management to facilitate historical traceability and reproducibility, which are very important for a continuously developing pipelines.

Apache OODT [11] is a data grid framework which provides capturing, locating and access data among heterogeneous environments. It shares both similarity and difference with `Pipeline61`. Firstly, Apache OODT provides more general task-driven workflow execution, in which developers need to write their own programs to invoke different execution tasks. However, `Pipeline61` focuses on deep integration with contemporary big data processing frameworks including Spark, MapReduce and IPython. Secondly, OODT uses a XML based specification for their pipelines while `Pipeline61` provides programmable interfaces in different programming languages. At last, OODT maintains general information and meta-data for the shared datasets, but `Pipleine61` provides explicitly defined provenance information for both input/output data and transformations for each task within the pipelines. Subsequently, `Pipeline61` natively supports re-produce/re-execution of historical pipelines and even part of the pipelines.

Table 1. Comparison of Existing Data Pipeline Frameworks

Pipeline framework	Crunch	Pig	Flume	Cascading	Tez	MRQL	Pipeline61
Execution context	MR & Spark	MR	Flume	MR (Spark, Flink future)	MR	MR, SPARK, Flink, etc.	Spark, MR, script
Pipe definition	Operators	Queries	Configurations	Operators	Edges	SQL	Pipes (various execution context)
Pipeline connection	Source, Target	Query composition	Source, Sink	Branches, Joins, mapTo	No	SQL	Connecting notations
Data model	PCollection	Schema,	No abstraction	Pipe	Vertex	schema	Depends on execution context
Programming Languages	Java	Pig Latin	Configuration file	Scala/ Java	Java	SQL	Scala/Java

VI. Discussion

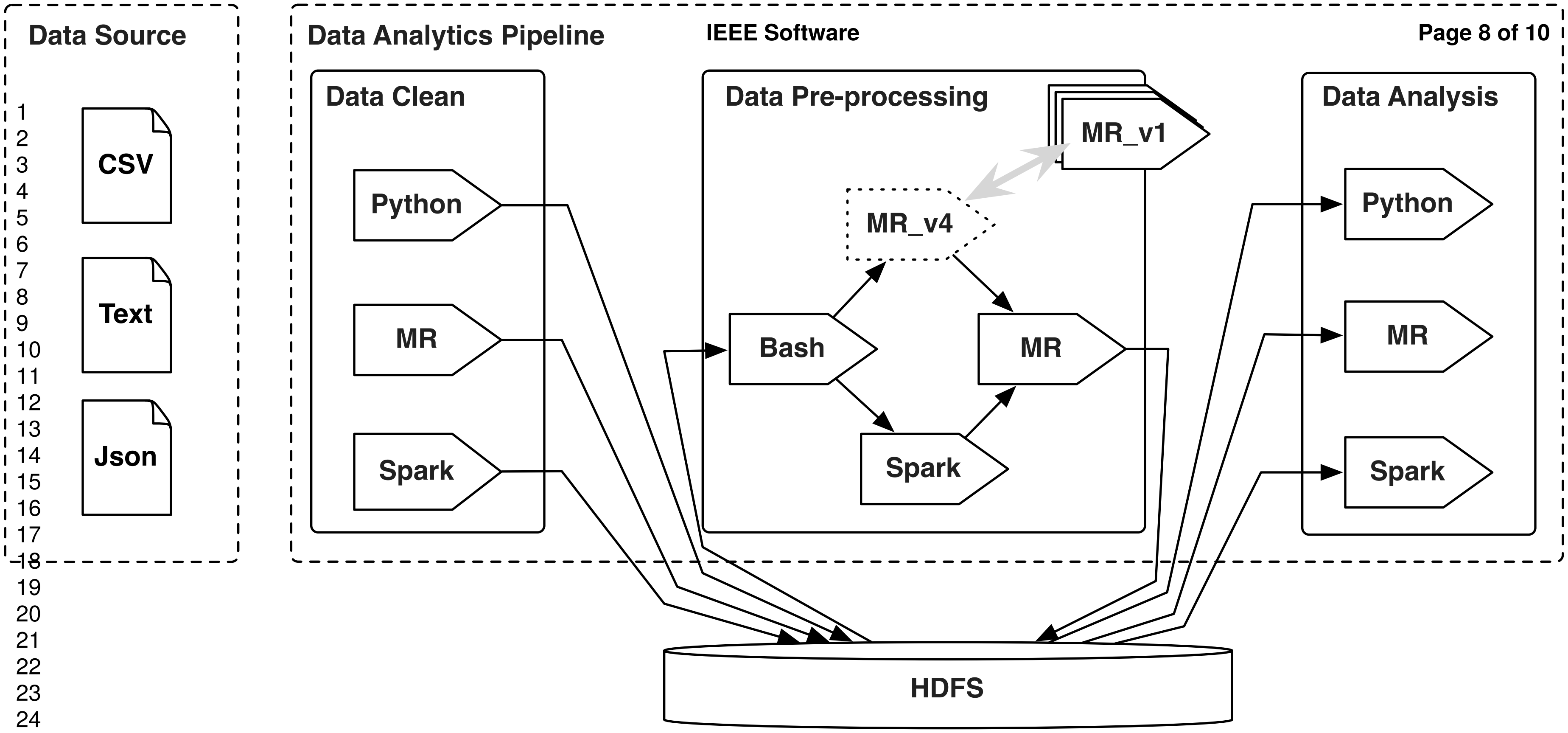
However, there are still some limitations of `Pipeline61`. Firstly, `Pipeline61` does not check the compatibility of data schemas across multiple data processing frameworks. So far, it relies on the developers to manually test the input and output of every pipe during the developing phase to make sure the output of a pipe can be feed into the pipe behind. In the future, we plan to utilize existing schema matching techniques to solve this problem. Secondly, most of the intermediate results produced during execution are require to be written to an underlying physical data storage (such as HDFS) for connecting various pipes with different execution contexts and ensuring the reliability for any components in the pipeline. Thus, the execution of pipelines in **`Pipeline61`** is generally slower than other frameworks, which are running independently over a single execution environment without integrating with external systems.

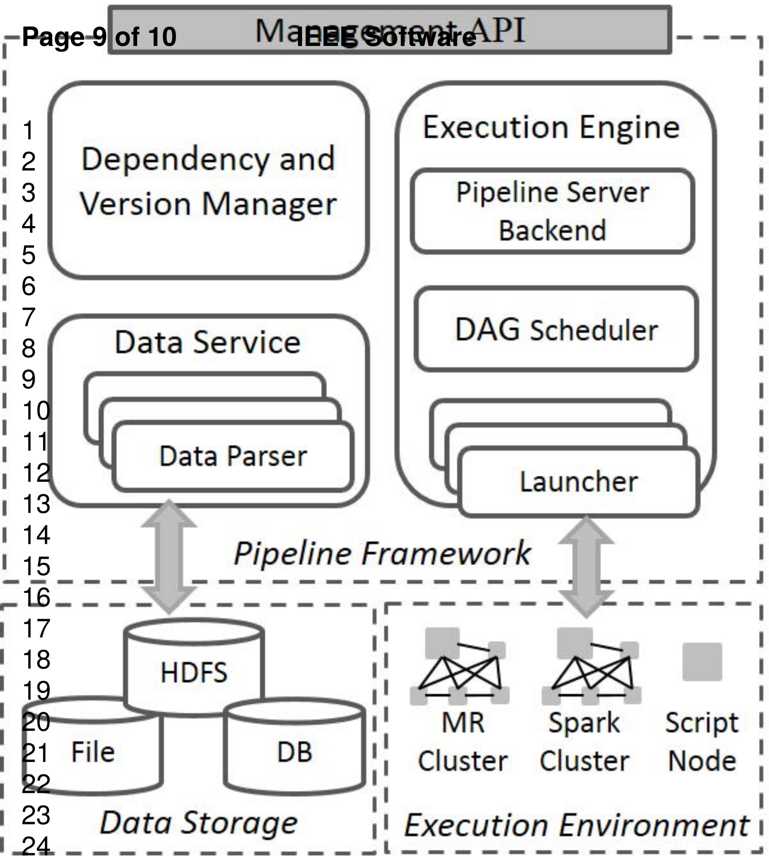
Acknowledgment

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] D. Sculley, G. Holt, D. Golovin, et al, “Machine learning: The High-Interest Credit Card of Technical Debt”, SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop), 2014.
[2] Apache Crunch – https://crunch.apache.org
[3] Alan Gates. Programming Pig - Dataflow Scripting with Hadoop. O'Reilly 2011
[4] Paco Nathan. Enterprise Data Workflows with Cascading. O'Reilly 2013
[5] Apache Flume – https://flume.apache.org/
[6] Apache MRQL – https://mrql.incubator.apache.org/
[7] Apache Flink – https://flink.apache.org/
[8] Bikas Saha, Hitesh Shah, Siddharth Seth, et al. “Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications”, In SIGMOD, pages 1357-1369, 2015.
[9] Ashish Thusoo, Zheng Shao, Suresh Anthony, et al. “Data warehousing and analytics infrastructure at facebook”, In SIGMOD, pages 1013-1020, 2010.
[10] Michael Armbrust, Reynold S. Xin, Cheng Lian,et al, “Spark SQL: Relational Data Processing in Spark”, In SIGMOD, pages 1383-1394, 2015
[11] Apache OODT - http://oodt.apache.org/
[12] Apache Tika - http://tika.apache.org/


Data Source
CSV
Text
Json
Data Analytics Pipeline
Data Clean
Python
MR
Spark
Data Pre-processing
MR_v1
MR_v4
Bash
MR
Spark
Data Analysis
Python
MR
Spark
HDFS


Management API
Dependency and Version Manager
Execution Engine
Pipeline Server Backend
DAG Scheduler
Data Service
Data Parser
Launcher
Pipeline Framework
HDFS
File
DB
Data Storage
MR Cluster
Spark Cluster
Script Node
Execution Environment