Making Real Time Data Analytics Available as a Service

Donna Xu[1,3], Dongyao Wu[1,2], Xiwei Xu[1], Liming Zhu[1,2,3], Len Bass[1]
[1]Software Systems Research Group, NICTA, Sydney, Australia
[2]School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
[3]School of IT, Sydney University, Sydney, Australia
doxu2620@uni.sydney.edu.au , {Dongyao.Wu, Xiwei.Xu, Liming.Zhu, Len.Bass}@nicta.com.au

ABSTRACT
Conducting (big) data analytics in an organization is not just about using a processing framework (e.g. Hadoop/Spark) to learn a model from data currently in a single file system (e.g. HDFS). We frequently need to pipeline real time data from other systems into the processing framework, and continually update the learned model. The processing frameworks need to be easily invokable for different purposes to produce different models. The model and the subsequent model updates need to be integrated with a product that may require a real time prediction using the latest trained model. All these need to be shared among different teams in the organization for different data analytics purposes. In this paper, we propose a real time data-analytics-as-service architecture that uses RESTful web services to wrap and integrate data services, dynamic model training services (supported by big data processing framework), prediction services and the product that uses the models. We discuss the challenges in wrapping big data processing frameworks as services and other architecturally significant factors that affect system reliability, real time performance and prediction accuracy. We evaluate our architecture using a log-driven system operation anomaly detection system where staleness of data used in model training, speed of model update and prediction are critical requirements.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures

General Terms
Design, Service-Oriented Architecture

Keywords
Software Architecture; Big Data Processing; Data Analytics Service

1. INTRODUCTION
With the rapid growth of computer capacity and speed, big data [2] have been generated from everywhere. These data can be processed and analysed so that the knowledge obtained can be used in various products. In order to process this large amount of data to enable enterprises to integrate the learned information with their products, many big data processing frameworks (such as Hadoop and Spark) have been developed. In addition, many machine learning libraries (e.g. MLlib and Mahout) have been developed on top of these big data frameworks to facilitate machine learning from large amounts of data. These machine learning algorithms are usually designed for batch processing where models are learned without real time requirements from the products using the models. The data are also batch pre-processed and already in a shared file system. However, if more real-time data is generated from the various data sources and needs to be reflected in the learned model, then the big data processing framework needs to be re-invoked to quickly update the existing model using the new data. There are important real time requirements in such periodic retraining since data staleness may result in poor prediction accuracy. For example, the new model needs to be immediately used by the product. In addition, the data sources, the big data processing framework itself and the use of the predictive models often need to be shared for different purposes. For example, key system events and system metrics generated from various sources in the past few seconds need to be accessible by the big data processing framework as they are important for a learned anomaly-detection model to be up-to-date. And this up-to-date model, not an early version, needs to be used in different systems such as alert management system and automated recovery system.

One approach is to make all key elements in data analytics as reusable services in a service oriented architecture. However, making real-time data analysis as a service is difficult. The challenges in turning real-time data analysis into services are:

1) Creating common service interfaces that are suitable for distinct data sources, model training (supported by big data processing frameworks), predictors (supported by the learned models) and different applications that can utilize all these services.
2) Wrapping of existing big data processing frameworks as a reusable model training service
3) Real-time challenges in processing of updated data to allow the learned model to be up-to-date while maintaining a near-real-time response time of the predictor service

To address the challenges above, we propose a general architecture to provide real time data analytics service, by wrapping dynamic model training services on backend training system and integrating it with data services, prediction services and the product that requires real time prediction. The same dataset along with its real time changing data is able to serve for different analytics purposes by different users. The trained model can be continually updated by real-time data streams, which is achieved by integrating batch and stream processing.

The contribution of this paper includes the following:

QoSA'15, May 04-08, 2015, Montreal, QC, Canada.

http://dx.doi.org/10.1145/2737182.2737186.

1) An architecture for real time Data Analytics as a Service and its integration with products using the data analytics;

2) Wrapping of big data processing framework as model training services;

3) Solutions to architecturally significant requirements around real-time and reusability;

4) Implementation and evaluation of the architecture by using a run-time operation anomaly detection system driven by real time logs and monitoring metrics.

The rest of the paper is organized as follows. Section 2 provides some related work to the key areas of this research. Section 3 presents the design of the architecture. Section 4 provides an explanation on the technical details of how the architecture can be built, using Spark framework as an example. Section 5 describes the case study and presents the experimental design, results and analysis. Section 6 concludes the paper.

2. RELATED WORK

2.1 Big Data Frameworks

The MapReduce paradigm [4] has been popular in many applications and is utilized by big data frameworks. Both Apache Hadoop [10] and Apache Spark [14] are Big Data distributed computing frameworks with an architecture that aims at fast and scalable batch data processing, following the MapReduce paradigm. In addition, there are significant extensions to traditional parallel processing for processing large data sets [1, 4, 9]. Although Extract-Transform-Load (ETL) has been a key use scenario of these big processing frameworks, increasingly the frameworks are used for machine learning to support data analytics applications. Parallel processing has also inspired parallelized and distributed machine learning algorithms for massively distributed data sets. The distributed machine learning frameworks (such as MLlib [12] and Mahout [11]) provide a solution for high level users or programmers. Spark's scalable machine learning library, MLlib, allows machine learning developers to easily construct distributed machine learning programs, while the machine learning layer Mahout on Hadoop is being rewritten to utilize Spark because of the performance of Spark. However, these big data batch processing frameworks and their machine-learning layer are usually not designed to be easily invokable as well-designed model training services by different users. It is also out of the scope of these frameworks to enable the resulting learned models to be used by different external products, not to mention dealing with real time requirements of these products.

On the other hand, many big data stream processing frameworks exist to deal with data sources continuously producing data. For example, Spark has a good streaming support integrated, which supports the building of real time predictive analytics services. Spark Streaming [15] is an extension to the Spark core, which provides a fast and scalable streaming data processing, where it integrates batch processing in streaming processing. Apache Storm [16] is another distributed computing framework for real time data processing. However, there are limited streaming machine learning libraries available. Most of the distributed machine learning libraries are only supported by batch processing frameworks. Therefore, in order to adapt distributed machine learning algorithms for real time machine learning, batch and stream processing need to be integrated.

2.2 Machine Learning as a Service

There are many commercialized Hadoop-like Software-as-a-Service offerings such as Amazon EMR [20], which is able to effectively process large amount of data, or Azure HDInsight [6], which provides big data processing and analysis in cloud. Each of these services supports only whole processing software-as-a-service which allows many big data solutions to be available on the platform, while we focus on fine-grained RESTful service design. Furthermore, there are many machine learning tools that aim at providing predictive analytics service. Google Prediction API [8] helps developers in creating many machine learning applications using RESTful API. However, these products all have their own disparate design styles of data services and prediction services. Some of them do not explicitly expose model training as a reusable service. For model training and predicting services, there are no real-time requirements considered.

Generic architecture on big data processing, such as the Lambda architecture [18], has a portion that deals with real time processing. However, these generic architectures do not use a service-oriented design and clearly differentiate data services, training services, predictor services and its integration with the final products.

On the other hand, RESTful services are increasingly used as a flexible way in designing data-related services. REST [5] stands for Representational State Transfer which is an architecture style originally used for the HTTP protocol retrieving web documents. The Representation concept in REST aligns well with the data abstraction behind a data service. Protocols and Structures for Inference (PSI) [13] is an initial attempt for presenting inference-related concepts as resource-oriented RESTful web services. There are different types of services and data access designs in PSI specification such as learner, predictor, data transformer and relations. However, PSI does not provide the architecture design behind these services and its integration with big data processing frameworks and external products. There are also no real time considerations.

In our approach, we use RESTful services for our architecture and follow some parts of the PSI specification for service interface design. We also tailor and improve PSI by focusing on the wrapping of big data processing frameworks and real time considerations.

Each of these systems and frameworks provides specific elements in building a real time data analytics architecture, however, none of them is able to supply all the requirements. Each of them is designed for solving some specific purposes only. For example, big data batch processing frameworks aim at processing large amount of data on distributed systems only once. The challenge remains to integrate all of them in a single solution, so that big data frameworks can be easily invoked as a service which is able to serve different users.

3. ARCHITECTURE DESIGN

The design of the data analytics service architecture, consists of three layers:

1) **Backend training system**, which can continuously process updated data streams and provide up-to-date results;

2) **Service wrapping layer**, which provide a service abstraction for data-analysis services;

3) **Service user interfaces**, which allow different jobs and products to be easily managed in accessing the real time data analytics services.

We also focus on how these components can be integrated together in order to provide real-time data analytic services. Figure 1 shows the overall design of the data analytics service architecture consisting of these different components.

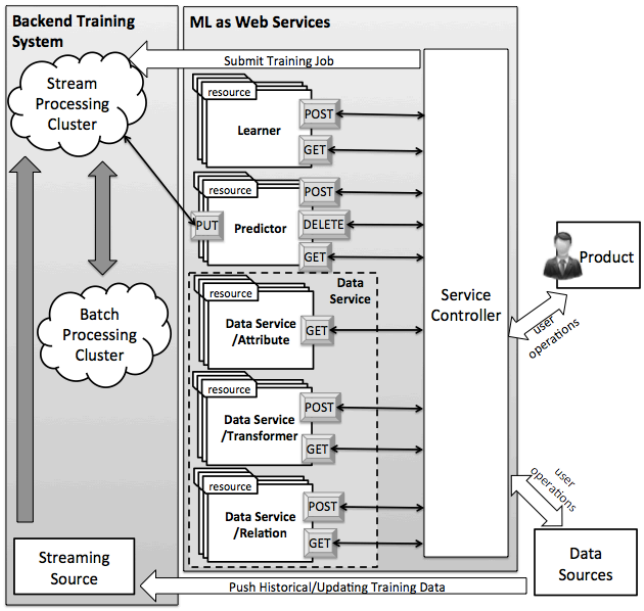

Figure 1. Overall Design of Data Analytics Service Architecture

3.1 Service Components

As shown in Figure 1, the architecture is made up of different service components in order to provide real time data analytics service. We discuss these services components in three parts.

3.1.1 Backend Training System

The backend training system consists of two logical clusters, batch and stream processing clusters. If a big data framework supports both types of clusters (such as Spark [14]), only one physical cluster is needed. The central functions of the backend system are training predictive models using some initial historical data and updating these models using additional real time data if necessary.

The initial training using some historical data is usually done by running an existing machine learning algorithm (implemented in some machine learning library) in the batch processing cluster because most of the machine learning libraries (such as MLlib and Mahout) are not designed for dynamic model learning for real time update.

A streaming source needs to be connected with the stream processing cluster as shown in Figure 1, which keeps delivering streaming data to the backend system for training. This is achieved by having RESTful data resource wrapped (explained in the next section), which contains the URI location reference to the streaming data. Each training job in the stream processing cluster can use the URI to locate the streaming data and monitors this specific location for real time training data. The stream processing cluster will process any new data added to the streaming source to have the model updated applying an efficient algorithm to update the model without relearning the whole data sets. The RESTful data resource can be reused by different model learning jobs.

The backend training system allows data to enter into the system in streams, and then integrates batch and stream processing on these data by using micro-batch processing. Each data stream is treated as a sequence of small batches so that they can be delivered from stream processing cluster to batch processing cluster along with the existing model, and the result after being applied scalable machine learning algorithms on the batch processing cluster will be delivered back to the stream processing cluster. The interaction between these two clusters requires either cluster resource sharing or a database connection in between.

3.1.2 Services Wrapping Layer

In order to provide common service interfaces that are suitable for distinct data sources, model training (supported by big data processing frameworks), predictors (supported by the learned models), RESTful interfaces are designed to allow clients to request on some main machine learning activities such as training, prediction and model updating. Here we follow guidance in the Protocols and Structures for Inference (PSI) [13] specification. Our design is intended to allow people who are not machine learning experts to access machine learning technologies easily, by sending HTTP requests to one or some of the five services which are Learner, Predictor, Transformer, Attribute and Relation following the CRUD (Create/Retrieve/Update/Delete) operation semantics.

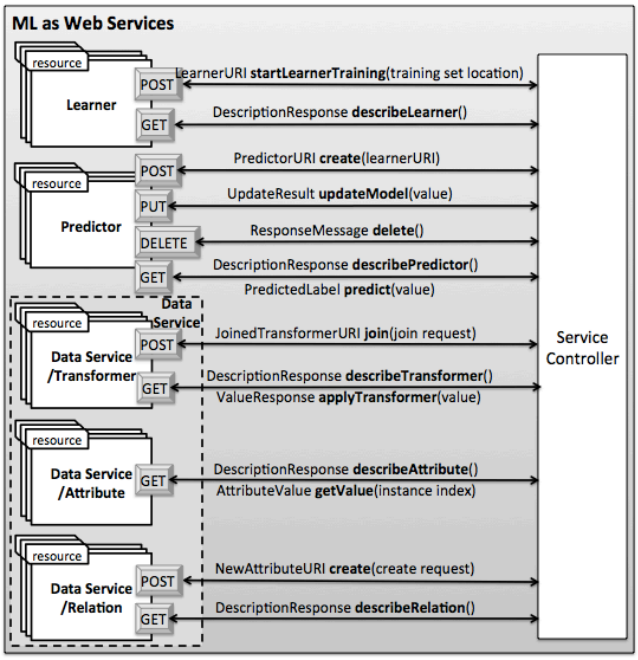

Figure 2. RESTful Web Services Interface Design

Figure 2 shows the details of our service interface design which involves the following main parts:

1) Service Resources

A typical data analysis jobs is modeled as five main RESTful service resources (which supports CRUD operations) in our system:

Learner provides different learner resources on different machine learning algorithms to the users, and constructs a corresponding *Predictor* (discussed later) with a learned model based on the training data set provided in the task schema. The task schema should provide the parameters required to learn the requested model, such as lambda value for learning Gaussian SVM, and the references to the training data set and true labels.

Predictor is the central service of the system, providing a prediction given a new instance of data. Each predictor is associated with a learner that constructs the predictor, and has a learned model as a result of processing one or several learning tasks. After a predictor has been constructed with a learned model, it can be updated with more training data. Predictor always holds the latest model trained by the historical and streaming data. Therefore, it is able to provide the user real time predictive analytics and reporting. It can be deleted if the user requests termination of the current job.

The following three service resources are all "Data Services" serving different purposes.

Transformer allows the data in one form to be transformed into another form in a batch fashion. The resulting data after applying a *Transformer* can be used for machine learning using a Learner or it can represent the extracted knowledge discovered from the original data.

Attribute is conceptually Transformer but applied to a specific data instance to return formats required by a particular *Learner*. They can be used if we only want to train a portion of features in the dataset. This is also important for performance reasons in the stream processing sense to have new data transformed quickly.

Relation represents a data set, which consists of multiple data instances. In our architecture, a relation resource on historical data set will be generated before the initial training. The real time updating data does not need to be constructed as a relation resource. All the information from the streaming data will be directly used for updating the learned model. Storage of streaming data and potential conversion to new *Relation* are handled separately.

Here is a list of examples of URL templates and HTTP response on these service resources.

1. Learner
POST http://data_analytics.com/learners/{model}
Learning Request:
{"task_schema": {
"parameter": $int,
"source": "$http://data_analytics.com/relations/{dataset}" }
}

201 Created
Location: http://data_analytics.com/learners/{model}
2. Predictor
GET
http://data_analytics.com/predictors/{model}?value={features}

200 OK
{"prediction_result": $prediction_result}
3. Transformer
GET
http://data_analytics.com/transformers/{method}?value={value}

200 OK
{"result": $number}
4. Attribute
GET
http://data_analytics.com/relations/{dataset}/{attribute}?instance={index}

200 OK
{"attribute": {{attribute1}: …, {attribute2}: …, …}}

5. Relation
GET http://data_analytics.com/relations/{dataset}

200 OK
{"source": $data_location}

2) Service Controller
The service controller manages and controls all incoming requests and dispatches them to the five corresponding resources of the respective model training job while hiding the complicated internal interaction between them. It also helps achieve load balancing and resource allocations to satisfy real time requirements. .

3) Request Mapping
The interactions between all the services and service controller are described as ***requests*** which is formatted as:

[*ReponserMessage*] [*Operation*]([*Arguments*])

• *Operation* is one of the HTTP methods (POST, GET, PUT and DELETE) sending from the service controller to one of the services. CRUD operations are mapped to HTTP methods. For example, **describeLearner** is mapped to GET for retrieving learner description.

• *Arguments* are the information/parameters a client provides when he/she is sending a request. For example, **applyTransformer**(value) needs the client to provide the value message he/she would like to apply a transformer on.

• *ResponseMessage* is the response the service returns to the service controller.

3.1.3 Service User Interfaces
It is important to hide the complexity of the web services in order to manage multiple training jobs and requirements from different products. To achieve this, we use the design pattern Facade [7] to wrap the service controller, and then provide simplified and unified interfaces to web services. As Figure 1 shows, service controller acts like a web service client in the web interface design to interact with the web services. It unifies all interfaces to define a high-level interface which allows the end users to use it more easily by providing only five interfaces: data preprocessing, describing a particular resource, training, prediction and stopping current job. For each of the user operation, it is mapped to one or more requests to the resources, which can be reused by different users on different purposes. Details of the mapping will be discussed later.

The product is a running system which requires real time data analytics using the predictor service. Data sources will keep generating more real time data and get data labeled and preprocessed by data services. All the resulted data will be contributed to updating the model. Then the data produced by the product which requires real time data analytics will request prediction service.

3.2 Interactions
3.2.1 Interaction Between Backend Training System and Service Layer
1. Challenges of Integration

Big data processing frameworks are usually designed for batch or streaming processing without the consideration of being invoked as a service by different service consumers with different quality requirements such as real time requirements. A major

architectural challenge is how to wrap such existing processing frameworks as reusable service with some quality of service control. More specifically, the challenges include:

1) Existing big data processing frameworks such as Hadoop and Spark require jobs to be submitted as a binary jars/libs rather than runtime functional calls. Thus, in order to retrieve the results of the submitted jobs, a shared external storage system (such as HDFS) is required for data exchanging which impacts real time aspects of getting the results. Alternatively, programmers need to manually write callback operations within their job execution logic, which forces programmers to consider more things other the analysis business logic and makes their program more complicated and error prone.

2) Big data processing jobs are relatively time-consuming so it is very inefficient to provide simple and synchronized response for clients and other components for real time data analytics purposes.

The issues above become the obstacles for interaction between big data processing clusters and the service layer. To deal with these problems in our framework, we proposed two architectural solutions:

1. None- intrusive solution

As shown in Figure 3, we add a function callback in every data analysis program and let it send the processing results back to the service layer. Then, the results of the job are updated at the service layer. A service callback containing updated URI is sent back to the service client – the product. After receiving the updated URI, the product can be triggered to apply latest tasks according to the results.

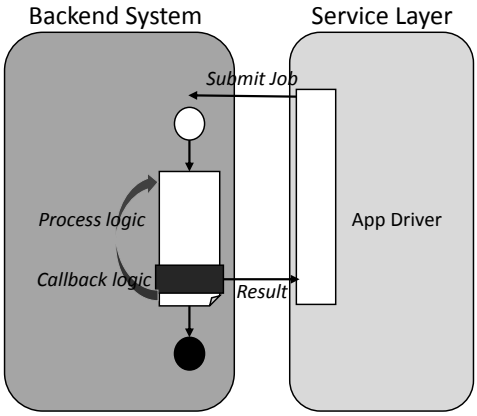

Figure 3. Synchronized Interaction between Backend Training System and Service Layer

This solution is simple to implement and friendly to the backend because no modification is required for backend system. However, it is not friendly to the service layer because the client doesn't know when the job would be finished so it has to periodically interact with the backend to check whether a notification is received or becomes blocked by waiting for the response results.

2. Intrusive Solution

For a more thorough solution, we believe the functionality for sending notification back to clients should be provided by the backend data processing framework rather than being done in the procedure of every job. So we modified the existing Spark framework to enable asynchronously sending callbacks from the job scheduler.

As shown in Figure 4, during the submission a job, both the application driver at the service layer and the job manager in the backend cluster will register a *Promise* object associated with the job ID. A *Promise* object is a referenced container which will contain the future results of a task/job. The *Promise* object will be returned to the caller while the application driver is running as a background process. After the job has been executed successfully, a notification with the promised result (can be actual data or usually a URL) will be sent back from backend and forwarded to the result dispatcher in the service layer. The service side can perform operations on the *Promise* object without blocking to wait for the results. The future operations applied on the *Promise* object will be automatically triggered and executed once the *Promise* object is notified to be succeeded.

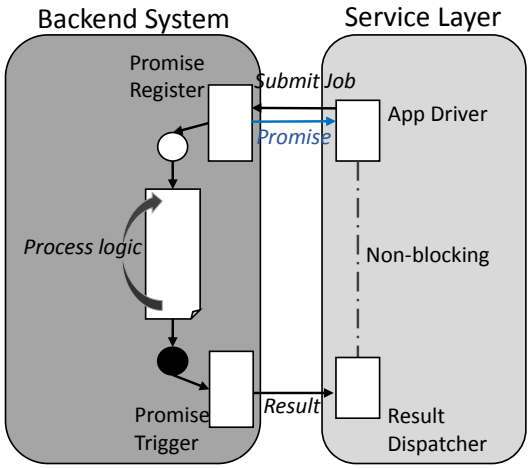

Figure 4. Asynchronous Interaction between Backend Training System and Service Layer

The advantages of this solution include:

a) Simplicity of the data processing program, the program submitted is clean without concerning about callback logic;

b) The *Promise* object returned from the backend can be used as a normal local object at client side so it reduces the complexity of interaction and communication between the client and backend server by avoiding waiting for the response of a submitted job. The trade-off is that this solution needs more support from the backend system so extra effort is required to modify the existing big data framework.

We have implemented both solutions in our prototype and we use the synchronized version in our evaluation part as it doesn't involve any code change in backend system.

3.2.2 Interaction Between Data Analytics Service and Product

The life-cycle management between the product and the entire data analytics service is shown in Figure 5. The user interface integrates the product and data sources with the RESTful web services by mapping each of the operations to the service resource requests. The operator of the product could get the information about any service by sending describing request to service controller which is mapping to GET request to the service. The operator is able to request training of a model where the argument of the training request contains the desired model type (such as

Gaussian SVM). The request maps to POST request to both learner and predictor services. A predictor URI will be returned to the operator, where the service associated with this URI containing a dynamic model can be requested to predict data by the product. After a learner and its corresponding predictor are constructed, a training job will be submitted from the service controller and run on the stream processing cluster in backend system.

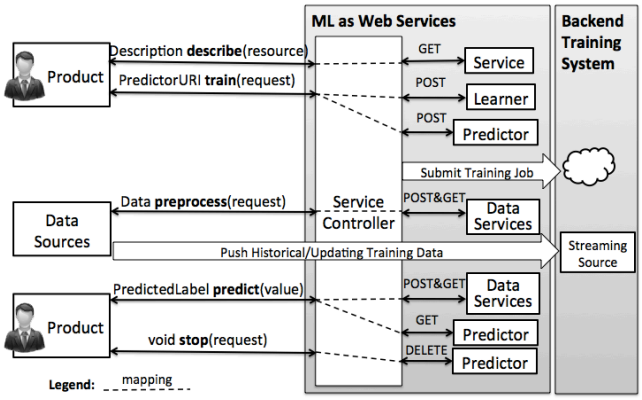

Figure 5. Life-Cycle Management Between the Product and the Entire Data Analytics Service

The real time training data generated by the data source usually requires some feature-engineering and format related data transformation, this is achieved by sending preprocess requests to the Transformer or Attribute resource through the service controller. The data delivery can be done whenever we would like the model in the predictor to be learned or continually updated. Each time the system finishes training one stream of incoming data from the streaming source, the stream processing cluster sends a PUT request to the predictor service, so that the model in the predictor will be updated to the latest. Thus real time predictive analytics and reporting will be provided to the operator for the product.

Once the product requires predicting data, a GET request will be sent to the predictor service along with the data, and it may also require data services to transform the predicting data.

Finally, the operator is able to stop the current training job when the product no longer requires the prediction service, by sending a DELETE request to the predictor service.

4. IMPLEMENTATION

This section provides detailed explanation on the entire architecture implementation and deployment. It discusses the technical details in implementing backend training system and data analytics service. The implementation on service user interfaces is simple and straightforward and the details were discussed in Section 3.

4.1 Backend Training System

We have selected Spark as the big data framework and MLlib as the scalable machine learning library to provide machine learning algorithms to train the data. Spark supports both batch processing and stream processing (Spark Streaming), so only one set of business logic needs to be implemented, which means that there will be only one big data processing cluster. Our approach can be generalized to other big data frameworks and distributed machine learning libraries. These processes will be described and discussed in this section.

4.1.1 Program Design and Implementation

The machine learning program in the driver program on Spark master is designed to collect live data streams once every *t* seconds and use these data streams to update the existing model in predictor service, where *t* seconds is the batch interval set during the creation of StreamingContext. Each stream processing framework has StreamingContext-like class for configurations such as setting stream source and batch interval. The batch interval determines how frequently the live data streams will be read in.

The pseudocode of the machine learning program in driver program on Spark master is shown in the following.

```
modelType ← args[0]
predictorURI ← args[1]
streamingSource ← args[2]
batchInterval ← args[3]
param ← args[4]
featureNum ← args[5]
model ← Vector.zeros(featureNum)
ssc ← StreamingContext(batchInterval)
inputData ← ssc.textFileStream(streamingSource)
intermediateData ← inputData.map(timestamp,
LabeledPoint(label, features))
for all r ← intermediateData.foreachRDD do
   r.cache()
   if r.count > 0 then
      model ← modelType.train(r.LabeledPoint, param,
model.weights)
      minTime ← r.timestamp.top(1)(OrderingASC)
      maxTime ← r.timestamp.top(1)(OrderingDESC)
      RESTClient ← ClientResource(predictorURI)
      RESTClient.put(model, minTime, maxTime)
   end if
end for
```

When a training job is submitted to the backend system, there are a number of details provided by the user, such as the requested model type, predictor URI, training set location, desired batch interval, parameters requested by the target model, and pre-defined feature number. The first six lines initialise the variables that passed from the user request. We then initialize the dynamic model to be 0. After the streaming configurations are set up, we collect real time data every *batchInterval* seconds. A map transformation operation will be applied on the data stream to create an intermediate data collection which aims at having it formatted in a desired way for easier processing in the later operations. We define the each line of the input training data to follow the format as the following:

TIMESTAMP, TRUE LABEL, FEATURES

The first field TIMESTAMP represents the time at when the line of this data instance has been generated. TRUE LABEL is the ground truth class of the current data instance so that we could use the supervised machine learning algorithm to train the model. FEATURES is an array of numerical values represents the context of the data instance. For example, in transaction fraud detection system, each of the training data instance contains three fields, the time at when the transaction happened, whether it is marked as fraud, and some context information extracted from this transaction followed by a predefined format such as the value of how long the transaction takes, how many times used for entering the correct password, or if the amount of money being transferred exceeds a certain limit.

The resulting intermediate data is in DStreams, which are batches divided from input data streams. It is necessary to convert DStreams to a sequence of batches where each of them is a

sequence of RDDs, which are parallel data structure that stores a collection of data partitioned over the nodes in the Spark cluster. Thus the machine learning models provided by MLlib library can be applied on RDDs for batch processing.

As we then see in the *for* loop, it is useful to cache the intermediate data collection as it will be used repeatedly, including extracting the maximum and minimum timestamp in the current data collection, counting of total number of data instances and training. Finally, the model will be updated and will be sent to the predictor service along with the max and min timestamp to indicate the version of the trained model.

4.1.2 Distributed Environment Deployment

To deploy a machine learning program from local to distributed environment, we need to explore the possible deployment modes the selected framework supports, and set up the required environment.

Spark provides several deployment modes. We have deployed applications on Amazon EC2, where the spark-ec2 script allows the user to create and terminate a Spark cluster automatically. It is designed to be convenient because as long as the user specifies the size and the name of the cluster, the cluster can be automatically created, and each worker node is able to execute tasks that assigned to them. Furthermore, starting the cluster from a customized AMI (Amazon Machine Image) [17] allows the user to include additional required software for running the machine-learning program.

4.2 Wrapping Data Analytics As a Service

In order to make real time data analytics a reusable service, the backend training system and a predictor need to be wrapped as a service. The service is able to serve different users from the same dataset as the training dataset is designed to be located in an external system and the location can be set as streaming source by different stream processing clusters. This section presents the implementation of the RESTful web services.

4.2.1 RESTful Web Services

We implement the RESTful web services using the Restlet framework in Java. Restlet is one of the mainstream frameworks for RESTful web service and it is also the first REST framework.

Each Restful application built on Restlet usually utilises two classes; Application class and Resource class. Each Resource object handles HTTP methods such as GET, POST, PUT and DELETE and can be mapped to its corresponding URI in Application object.

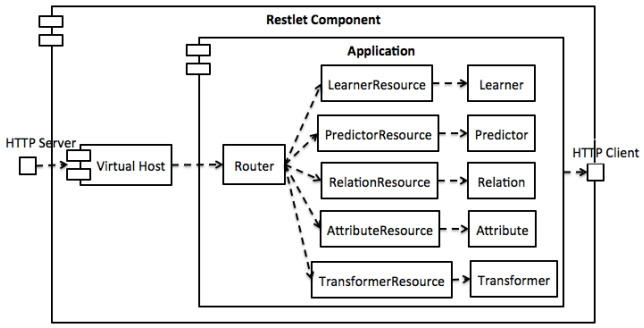

Figure 6. Restlet Component and Main Classes

Figure 6 shows all the framework classes implemented in our service. The Restlet component is used to manage Virtual servers and Applications. There is only one Virtual server in our implementation, which is used to host multiple names on the same server. The application manages a group of Resources and Representations. There are five Resources in our implementation which can be shown in the figure, and five corresponding Representations as well. Applications are portable and can be reconfigured so that they can run on difference virtual servers over different Restlet implementations.

5. EVALUATION

In order to evaluate real time data analytics service architecture, we used the above implementation in a case study. This section presents the case study, the evaluation design, experiment results and analysis.

5.1 Case Study

We use an anomaly detection system as the product which requires real time data analytics over logs and monitoring metrics. Operation logs usually contain information about the progress of an operation such as upgrade, reconfiguration, and deployment. Monitoring metrics usually indicate the status of a system in terms of CPU utilization, memory usage and network throughput. In our early work [19], we showed that using progress indication in operation logs is critical for interpreting monitoring metrics in anomaly detection and enables the operators to be able to make right decisions at the right time. For example, logs indicating an ongoing upgrade may suppress unnecessary alarms caused by legitimate termination of instances and associated CPU utilization increase. Evidently, this requires the real-time logs and metrics to be used for anomaly detection model training.

5.2 Data Collection

We use the anomaly detection product over application upgrade operations where upgrade logs and typical monitoring metrics of the upgrade application are the data sources. A data transformer will use timestamps to correlate operation status from operations logs and metrics from monitoring systems and produce a single line for each time period (in seconds) as Relation data source. Specifically, the line of each training data instance consists of a timestamp, which indicates the time when the data is produced, a true label referring to normal/abnormal. More features can be added by manipulating existing attributes in the data instance. The line of each predicting data instance has a timestamp and the associated attributes.

5.3 Experiment Setting

We run our experiments on Amazon Web Services. There are up to 9 EC2 instances being used in the experiment. Each of these instances uses t2.medium Ubuntu operating system with 2 cores and 4GB of Memory. One instance acts as Spark Master and the others act as Spark Workers. There is 512MB memory reserved by the system for usage such as crash recovery. Therefore, we are able to allocate up to 3.5GB to the executor memory for each worker in order to perform in-memory processing for Spark jobs. Spark 1.0 and OpenJDK 7 are installed.

5.4 Experimental Design

In order to evaluate the system regarding to the system reliability, real time performance and prediction accuracy, we focus on the data the system needs to process and how well it is able to achieve "real time". The better it achieves "real time", the more accurate the prediction of the new data would be. Therefore, the central measurement is the latency it takes from the training data generation to having the learned model updated. The smaller the latency we achieve, the higher the reliability and performance the

system will be, and thus more streaming data will be trained to improve the prediction accuracy.

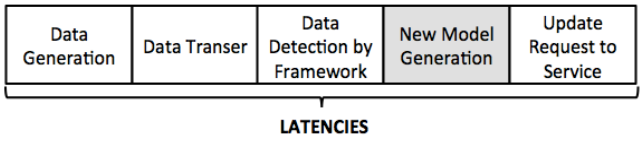

Figure 7. Latencies From Data Generation to Having Learned Model Updated

Figure 7 shows the elements of latency during the process of updating learned model. Data generation latency consists of system producing the operation logs and metrics, applying feature engineering and adding true labels by the data services. The data transfer latency depends on the network speed for transferring data from the system logs and metrics generation location to the operator and then from the operator to the Hadoop-compatible filesystem. The latency of data detection by the Framework depends on the internal data detecting mechanism in Spark in this example. The new model generation latency mainly comes from data processing in backend cluster, including counting the number of streaming data instances in each batch interval, extracting the maximum and minimum timestamp of the input data, and training these data to update the existing model. Finally, the latency for sending an update request to web service to update learned model depends on the network speed. Overall, only the model generation latency can be adjusted and tuned by the setting of our system. All other latencies are influenced by the factors that are not controlled by our system. Therefore, we are focusing on the latency of new model generation, and investigate the factors that affect it. Our hypotheses for these factors are historical data volume, batch interval, streaming data velocity, number of cores in the cluster and the total cluster memory. Five sets of experiments are designed and conducted. In each set, one factor is set to be fixed and all others are varied, to investigate the relationship between each of these potential factors and the model update latency.

5.5 Streaming Data Simulation

We programmatically simulate streaming data with a specific steady speed that enters into the system for a continuous time. We use AWS S3 as the filesystem that stores the training data. We will push the data files into the specified location on S3 at a specific speed for a continuous 200 seconds. For example, to simulate data streaming into the system at velocity 2MB/s, we will push a 500KB file every 250 milliseconds for 200 seconds. The latency will be measured by taking the difference between the end of the time of data simulation and the time when the learned model finishes updating on these 400MB updating data.

5.6 RESULTS AND ANALYSIS

This section presents the results of the experiment we have designed and carried out, evaluates data analytics architecture and assesses the efficacy of the proposed factors on the learned model update latency.

5.6.1 Streaming Data Velocity

To evaluate the performance of the system regarding to the data velocity, we perform six experiments to measure the model update latency on different data velocities ranging from 0.5MB per second to 20MB per second, using 1, 2 and 4 workers respectively each time. We set the batch interval to be 10 seconds, and each worker has 512MB executor memory. The latencies on different data velocities using different number of workers in the cluster has been summarised in Table 1 and shown in Figure 8.

Table 1. Table Of Latency On Different Number Of Workers For Each Velocity

VELOCITY	1 WORKER	2 WORKERS	4 WORKERS
0.5MB/s	8.9s	3.1s	8.3s
1MB/s	9.2s	7.1s	4.8s
2MB/s	8.2s	7.4s	6.3s
5MB/s	88.6s	9.6s	4.4s
10MB/s	318.4s	119.8s	10.0s
20MB/s	793.8s	384.2s	178.1s

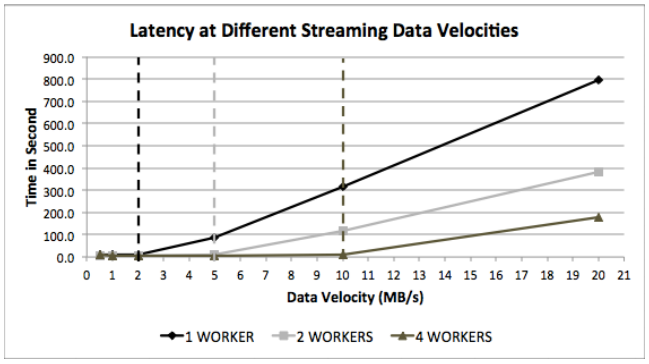

Figure 8. Latency On Different Data Velocities Using Different Number Of Workers

For each fixed number of workers, the latency rises sharply after a certain data velocity. Interestingly, the trend line of the latency against the data velocity is almost linear after this point of data velocity, and the steepness of the line is different for different numbers of workers. Here we define the near real time to be 10 seconds, which is a reasonable definition as the batch interval is set to be 10 seconds. The table clearly shows that the system is able to handle up to 2MB/s streaming data without any delay using 1 worker, 5MB/s streaming data using 2 workers and 10MB/s data using 4 workers. Therefore, the latency is relatively low (less than 10 seconds) before a particular velocity point, and goes up dramatically after the point, as the streaming data is too fast to be processed. Furthermore, the more workers we have in the cluster, the higher velocity the system will be able to handle without delays in each batch.

The figure then highlights that the latency goes up as the data velocity increasing when the data velocity goes beyond the system handling point using each fixed number of workers. In addition, the steepness of the trend line gets smaller if we are using more workers. For example, if we change the number of workers from 1 to 4, there is a decrease by 65 % in the steepness of the line.

5.6.2 Level of Parallelism

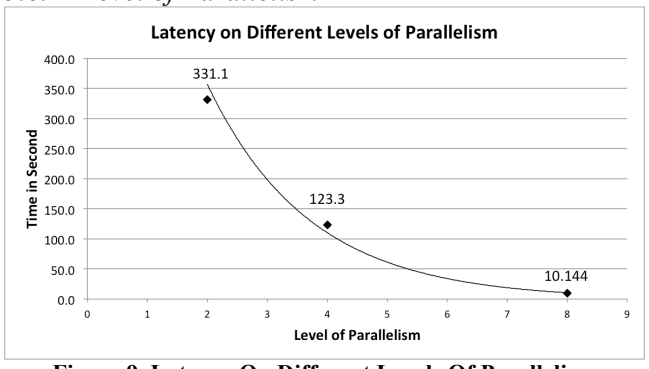

Figure 9. Latency On Different Levels Of Parallelism

The level of parallelism represents the number of cores in the cluster that can be used to achieve parallel data processing. As we mentioned in the experiment setting, each instance has two cores. In the experiments we conduct, we set the batch interval to be 10 seconds, the total cluster memory to be 2GB, and data streams entering into the system at 10MB/s for a continuous of 200 seconds.

Figure 9 highlights that as we increase the level of parallelism, the latency drops down exponentially. It clearly shows how the latency falls down from 331 seconds to 10 seconds by adding the number of cores from 2 to 8. Therefore, we can conclude that the factor of the level of parallelism contributes a significant reduction in the model update latency.

5.6.3 Total Size of Cluster Memory

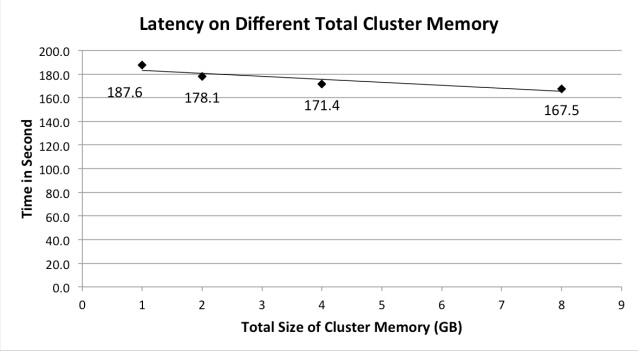

Figure 10. Latency On Different Size Of Cluster Memory

The total size of the memory in the cluster represents the total executor memory of the workers that are used to process data. We conduct the experiment with the setting of 10 seconds as batch interval using 4 workers with the level of parallelism 8. Data streams are simulated to enter into the system at 20MB/s for a continuous of 200 seconds.

Figure 10 presents the total latency on a different size of the cluster memory. It clearly shows that as we increase the cluster memory, the latency decreases. However, the improvement in decreasing the latency is minimal. For example, doubling the total cluster memory from 2GB to 4GB only results in a latency improvement of around 7 seconds. The reason that increasing the total size of the cluster memory does not contribute in latency improvement is that the system is focusing on processing the streaming data and can automatically cache the data for iterative machine learning algorithms. Therefore, as long as the total amount of data in each batch and the intermediate results produced during data processing fit into the memory, no matter how large the memory is, it would not help too much. Furthermore, the small improvement in latency using larger memory size is contributed by the fewer times the buffer pages are evicted when the memory is full. For example, suppose each worker processes a total of 50MB streaming data for each batch, and there are around 50MB intermediate results produced during data processing, then there will be 100MB that needs to be processed on each worker for each batch. If the total memory is 2GB with 512MB on each worker, the first time the buffer page eviction will be requested is at batch 6. However, the first time the buffer replacement is triggered using the total memory 4GB in the cluster with 1024MB on each worker will be at batch 11. Thus, the difference of 7 seconds in the latency between using 2GB and 4GB memory comes from difference of the number of times to clear the cache.

5.6.4 Batch Interval

Batch interval represents the period of time the system is collecting the streaming data each time. The experiments are conducted on different batch interval settings over 1, 5, 10 and 15 seconds, with fixed setting of a total cluster of 2GB memory using 4 worker, simulating 10MB/s and 5MB/s streaming data for a continuous of 200 seconds respectively.

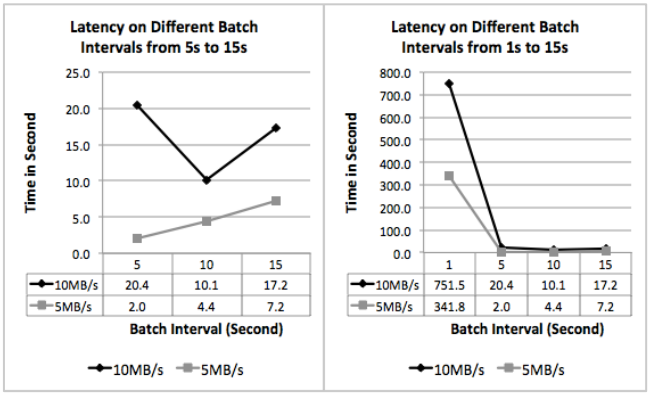

Figure 11. Latency On Different Batch Intervals

The figure on the left-hand side of Figure 11 shows a comparison of the latency with 10MB/s and 5MB/s streaming data speeds over different batch intervals. It indicates that setting batch interval to be 1 second causes a significant amount of latency on both 10MB/s and 5MB/s streaming data. The figure on the right-hand side of Figure 11 has a closer look at the performance on batch interval 5, 10 and 15 seconds. Interestingly, the shapes of the line for different data velocities are different. When the data velocity is 10MB/s, the latency line is similar to a quadratic curve, with the minimum value at batch interval 10 seconds, which means the smallest latency is achieved at the setting of batch interval 10 seconds when there are a total of 2GB memory in the cluster with 4 workers and data is streaming into the system at 10MB/s. For data velocity of 5MB/s, the relationship of the latency against the batch interval is linearly increasing. The smallest latency is 2 seconds when the batch interval is 5 seconds. Therefore, the smallest latency for different streaming data speeds happens at different batch interval settings. We can conclude that for a particular streaming data velocity and fixed number of workers and size of memory, the latency will go down relatively steeply as the batch interval increasing from 1 second to the most optimised batch interval setting, and then the latency will go up with moderate steepness from the most optimised batch interval setting to a limit that the system is no longer able to handle.

5.6.5 Historical Data Volume

The evaluation on the performance and reliability of the system training on high volume data is also crucial, as the system will need to train a certain amount of historical data initially when the end user requests to learn a model. The experiment is conducted on different size of the historical data volume with the batch interval 10 seconds using 4 workers with 512MB memory on each worker. Figure 12 deals with the system performance on different historical data volume. The experiment is set up with 4 workers being used and a total memory of 2GB in the cluster. It clearly shows that as data volume increases from 0.5GB up to 16GB, its training time increases almost linearly. However, it takes more than four minutes to finish training the historical data with the data size more than 4GB, which results in a low performance because even though the end user tries to update the model, he/she needs to wait for several minutes to have model learned on

historical data after model learning request. It is recommended that the system trains on a small amount of historical data and updates the model on high speed streaming data.

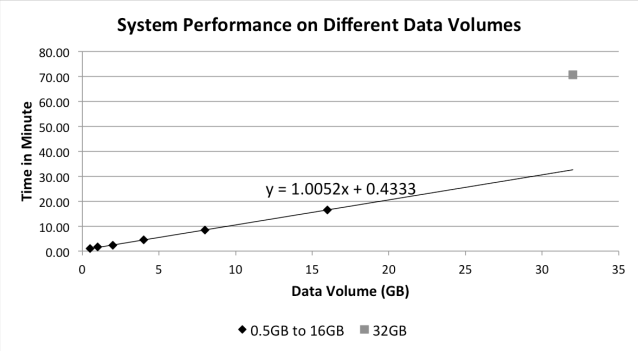

Figure 12. System Performance On Different Data Volumes From 0.5GB To 32GB

In addition, the figure shows the system performance on training 32GB historical data in grey rectangle dot. The linear trend line from 0.5GB to 16GB forecasts that the expected performance for training 32GB historical data should be around 33 minutes. Compared the expected performance with the actual one which is 70 minutes, it indicates that the system is highly unreliable in training 32GB historical data with batch interval 10 seconds using 4 workers in a cluster of 2GB memory. Our guess is that the total size of the cluster memory is no longer able to process this large amount of data at once, and the garbage collection limit has almost been exceeded. Therefore, if we add more memory into the cluster, the relationship between the performance of training on these sizes of historical data from 0.5GB to 32GB and data volumes will be almost linear. Based on the other factors we investigated that affects the performance of the system, we found that in order to improve the performance of training on large amount of historical data, the number of cores in the cluster needs to be increased to achieve higher level of parallelism. Adjusting batch interval will not help in training historical data as it is used to tune the performance on processing streaming data.

6. ACKNOWLEDGMENTS

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

7. CONCLUSIONS

The main contribution made in this paper is a real time data analytics service architecture design where it allows a machine learning model to be continually updated by real time data and it wraps big data processing framework as reusable services. We conduct the evaluation of the architecture regarding to the system reliability, real time performance and prediction accuracy. We also investigated the factors that affect the model updating latency. We have concluded that a reliable, high-performance system with high prediction accuracy can be achieved by low historical data volume and streaming data velocity, using a high level of parallelism with sufficient memory allocated to each node, and using the right batch interval which can be found by testing from a conservative batch interval. The machine learning algorithms supported are limited to the applied machine learning library and big data frameworks.

8. REFERENCES

[1] Chaiken, Ronnie, et al. "SCOPE: easy and efficient parallel processing of massive data sets." Proceedings of the VLDB Endowment 1.2 (2008): 1265-1276.

[2] Paul Zikopoulos, Chris Eaton, et al. Understanding big data: Analytics for enterprise class hadoop and streaming data. McGraw-Hill Osborne Media, 2011.

[3] Curbera, Francisco, et al. "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI." IEEE Internet computing 6.2 (2002): 86-93.

[4] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.

[5] Fielding, Roy Thomas. Architectural styles and the design of network-based software architectures. Diss. University of California, Irvine, 2000.

[6] Microsoft Azure HDInsight. http://azure.microsoft.com/en-us/services/hdinsight/.

[7] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design patterns: elements of reusable object-oriented software, volume 206. 1995.

[8] Google Prediction API. https://cloud.google.com/prediction/docs.

[9] Isard, Michael, and Yuan Yu. "Distributed data-parallel computing using a high-level programming language." Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. ACM, 2009.

[10] White, Tom. Hadoop: the definitive guide: the definitive guide. " O'Reilly Media, Inc.", 2009.

[11] Mahout: Scalable machine-learning and data-mining library. http://mahout.apache. org/.

[12] Mllib. https://spark.apache.org/mllib/.

[13] Protocols and structures for inference. http://psi.cecs.anu.edu.au/.

[14] Zaharia, Matei, et al. "Spark: cluster computing with working sets." Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 2010.

[15] Zaharia, Matei, et al. "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters." Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing. USENIX Association, 2012.

[16] Apache storm: Distributed and fault-tolerant realtime computation. https://storm. apache.org/.

[17] Amazon Machine Image (AMI). http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

[18] Lambda Architecture. http://lambda-architecture.net/.

[19] Xu, Xiwei, et al. "POD-diagnosis: Error diagnosis of sporadic operations on cloud applications." Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on. IEEE, 2014.

[20] Amazon EMR. http://aws.amazon.com/elasticmapreduce/.