Increasing the Accessibility to Big Data Systems via a Common Services API

Rohan Malcolm[*], Cherrelle Morrison[*], Tyrone Grandison[#], Sean Thorpe[*], Kimron Christie[*], Akim Wallace[*], Damian Green[*], Julian Jarrett[*], Arnett Campbell[*]

[*]Computational Science Research Group
School of Computing and Information Technology
University of Technology
St. Andrew, Jamaica

[#]Proficiency Labs International
Ashland, Oregon

{1103314, 1007672}@students.utech.edu.jm, tgrandison@proficiencylabs.com, thorpe.sean@gmail.com, {0802085, 1007549, 1002336}@students.utech.edu.jm, jjarrett@utech.edu.jm, arcampbell@utech.edu.jm

***Abstract* — Despite the plethora of polls, surveys, and reports stating that *most* companies are embracing Big Data, there is slow adoption of Big Data technologies, like Hadoop, in enterprises. One of the primary reasons for this is that companies have significant investments in legacy languages and systems and the process of migrating to newer (Big Data) technologies would represent a substantial commitment of time and money, while threatening their short-term service quality and revenue goals. In this paper, we propose a possible solution that enables existing infrastructure to access Big Data systems via a services application programming interface (API); minimizing the migration drag and (possibly negative) business repercussions.**

Keywords — Big Data, Hadoop, API, Big Data Systems

I. INTRODUCTION

Amazon, Apple, Facebook, Twitter, Netflix, and Google have set the standard for the current and emerging era in computing. Their core business is built on collecting, analyzing and monetizing large quantities of data. The magnitude of data and the processing required (within user expectations of response time) precludes the use of traditional data management technologies and has ushered in the age of *Big Data* [1].

Midsized and large organizations recognize the benefits of investing in Big Data systems; but have been slow in their adoption due to varying reasons [2, 3]. They range from lack of necessary skills as data scientist to the financial uncertainty on how one qualifies the tradeoffs on the return on investment (ROI) in the short run as opposed to the log run outlook of a business. Other concerns surround effort, value decay, service degradation and disruption from porting current systems to newer infrastructures and technologies. We purport that the latter rationale can be partially mitigated by technology.

In this paper, we propose a mechanism for methodically converting the current legacy enterprise application stack to one that leverages the latest and greatest Big Data technologies; while lessening the effort required and the possible disruption to the firm's value proposition and quality of service agreements.

We begin by presenting the fundamentals: What is Big Data? (Section II); and what is a typical Big Data stack and how it works (section III). In section IV, we present our proposal and discuss possible use cases (section V). Section VI presents related work, followed by future work (section VII) and conclusion (section VIII).

II. WHAT IS BIG DATA?

In the most basic of terms, Big Data refers to the collection, processing and analysis of extremely large data sets usually at scale of petabyte limits and beyond, especially for large scale University laboratory environments like ours. The magnitude and complexity of these data sets are very significant by way of volume that it becomes extremely difficult to process using contemporary database management tools or traditional data processing applications. For Big Data systems, there are normally challenges relating to capture, curation, storage, search transfer analysis and visualization of these data sets [4].

The McKinsey Global Institute estimates that data volume is growing 40% per year, and will grow 44 times larger between 2009 and 2020 [4]. The rise in prominence of Big Data stems from the value that can be extracted – correlations that can spot consumer and or business trends, insight that can be used to help with disease prevention, crime abatement, traffic routing, security breach detection, product enhancement, supply chain optimization, among others [4].

Formally, there is no official consensus on the scientific definition of Big Data. Gartner, and the rest of the computing industry, formally use the "3Vs" model as the basis for describing Big Data [5]. In 2012, Gartner presented its updated definition, which the industry's de facto standard and states that "Big Data are high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization" [6]. In recent times, the dimensions of

value and veracity have been added to the list of important characteristics of Big Data. Thus, the "5Vs" model is currently the acceptable standard for Big Data.

1. Volume - this demonstrates large amounts of data collected by organizations measured petabytes, exabytes, zettabytes and even yottabytes worth of data from multiple sources, e.g. sensors, social media, smart digital devices, etc.
2. Velocity - this considers real-time processing and analysis of large columns of data streaming in and out of an organization.
3. Variety - this is concerned with varying forms of data organization, i.e. structured, semi-structured or unstructured, and in multiple modalities, i.e. free text, video, audio, sensor data, logs, etc.
4. Value - this seeks to monetize through insight and influence spending, cost-saving strategies and optimizations.
5. Veracity - this addresses the integrity of data and their respective sources, as they impact critical choices.

The "5V's" model will be assumed for this paper. Now that we understand what Big Data is, let's examine the underlying architecture.

III. Typical Big Data Stack: What & How

A typical Big Data stack has a data management platform that distributes data across multiple machines, is fault-tolerant and enables querying and analysis across multiple, disparate machines, which may hold portions of the data being examined.

To better illustrate the actual use and operation of a Big Data stack, we will use the example of Hadoop, the de facto Big Data platform standard.

A. Hadoop

Hadoop is an open-source implementation of Google MapReduce, developed by the Apache Software Foundation that allows large amounts of structured and unstructured data sets to be handled quickly.

The Hadoop framework consists of two layers: Hadoop Distributed File System (HDFS) and MapReduce. It is written in Java and supported on any operating system platform. In Fig. 1, we see the Hadoop architectural overview.

Hadoop accepts some form of data and splits it into different portions across the cluster (this facilitates the MapReduce algorithm so as not to burden a single machine). After the completion of the maps, i.e. the tasks that run simultaneously on all the relevant machines, they are all gathered, reduced and written to an output file for storage [7].

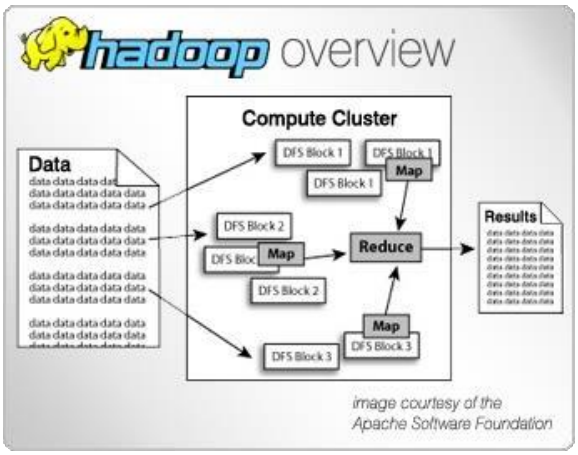

Fig. 1. Hadoop Architecture Overview

B. Hadoop Distributed File System (HDFS)

HDFS enables the file systems on the local host machines to be linked together to create one very large file system. HDFS is built with the tenet that faults are a normal occurrence (not an exception). Thus, HDFS always seeks to detect and recover from faults quickly. It copies data across multiple nodes to enable quick recovery in the event of node failure [7].

C. MapReduce

MapReduce is a programming model (an associated implementation) for processing large data sets with a parallel, distributed algorithm on a cluster [8].

MapReduce users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Fig. 2 illustrates the steps taken in a MapReduce job.

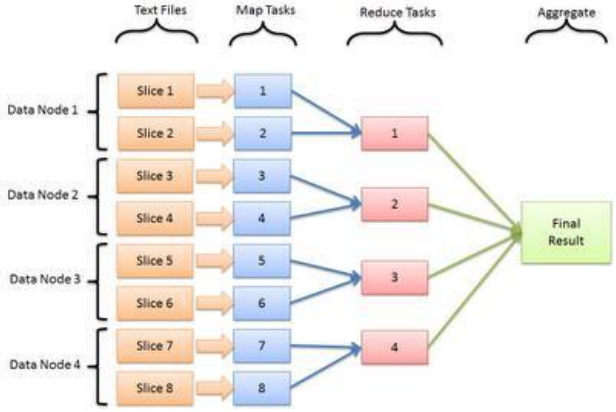

Fig. 2. Basic Structure of the MapReduce algorithm

Given this background on a standard Big Data stack, we can elucidate on the interface that we propose to leverage this stack.

IV. BIG DATA SERVICES (BDS) API

An API (Application Programming Interface) is a set of rules and instructions that an application can follow to access services and resources that are provided by another application. More commonly, an API represents a set of method or function calls that dictate the use of an underlying object [9, 10].

This paper purports the creation of a Big Data Services (BDS) API, which would facilitate the easy manipulation of Big Data that are stored on separate platforms; with the platforms potentially employing different languages, run-time environments and frameworks.

The BDS API is platform and language independent; allowing it to connect to and from any language or platform. This also applies to legacy languages and systems implemented in them.

The BDS API is focused on the goals of seamless processing, communication, storage and sharing.

Big Data Processing: The BDS API allows for cross platform processing of large data through the use of streams and stream managers. This allows the user to split data processing across several machines clusters as opposed to a single cluster.

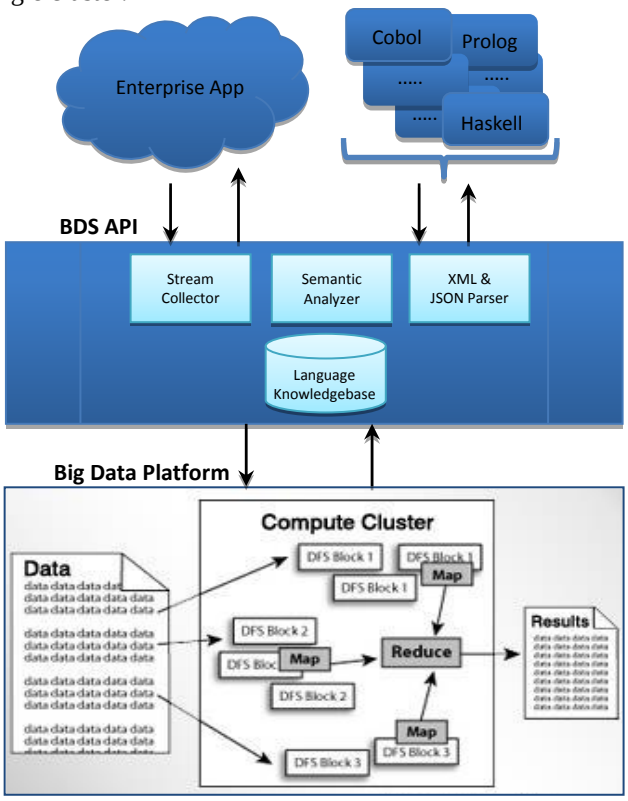

Fig. 3. BDS API Function

Language Communication: Through the use of Extensible Markup Language (XML) and JavaScript Object Notation (JSON) parsing, as well as a Language Knowledgebase (Fig. 3), the Semantic Analyzer (Fig. 3) of the BDS API is able to provide cross language support.

Storage (on Legacy Systems): As previously mentioned, traditional data warehousing systems cannot be updated without a large investment of time and funds. The BDS API, with its language communication support, allows data to be passed between systems regardless of architecture or programming language this allows legacy systems to communicate with more modern systems.

Data Sharing: Stream collectors, built into the BDS API, manage streams connections, and provide fault tolerance mechanisms to allow large data sets to be shared reliably and quickly.

In Fig. 3 below, we see that the BDS API may be utilized by any number of applications working alone or collaboratively. Here, we use Hadoop as an instance of a Big Data platform (without loss of generality). When an application (or a set of collaborative applications) wishes to communicate with the Big Data platform, it connects to the BDS API through streams[1]. A BDS API stream is similar to other conventional streams [11].

This streaming interface allows easy communication between the BDS API and the Big Data platform (Hadoop in the current instance), without the need for a specific API for every programming language.

Fig. 4 shows a simple example of how applications, possibly coded in different languages, can communicate with each other to work collaboratively towards an established goal. In this specific scenario, there are three systems or applications. As indicated earlier, each system has an associated process stream.

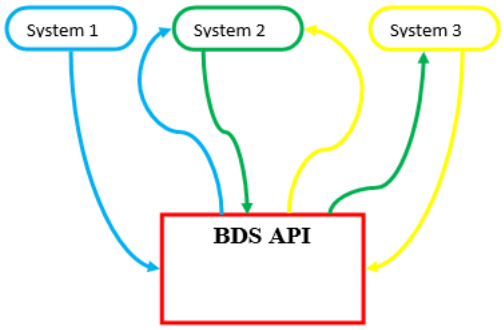

Fig. 4. System/Application Communication using the BDS API

We color-code the process stream for each system: blue represents *System 1*, green represents *System 2*, and yellow represents *System 3* (Fig. 4). In this situation, the current task involves *System 1* sending data (or a command), to the API, that is processed and becomes readable for *System 2*. *System 2* retrieves the information and sends data to the API, which processes the command and sends the results to *System 3*. *System 3* processes the data and sends it back to the API for processing then it sends it back to *System 2* for the initial process to be completed.

Given this illustration of how the BDS API may be used, we will expound on the components of the BDS API.

[1] A stream is a full-duplex bidirectional data transfer path between a process in the user's environment and one in the driver environment [11].

A. BDS API Components

As shown in Fig. 3, the BDS API has four (4) distinct components: the Stream Collector, the Semantic Analyzer, the Language Knowledgebase and the XML and JSON parser.

1) Stream Collector

This component stores and manages all aspects of the API related to stream connection. The Stream Collector also contains information about all known languages' stream methods and the criteria required to connect successfully and efficiently to all language streams. The Stream Collector is normally the sole part of the API that an (external) application communicates with. The Stream Collector acts as the coordination point that communicates with the other BDS components to accomplish a given task. This single entry and exit approach enables uniform enforcement of security constraints. On the contrary, this also results in the Stream Collector potentially becoming a performance bottleneck. Hence, we have designed it with bottleneck detection in data processing in an attempt to avoid this problem. The Stream Collector also contains functionality for fault tolerance, which enables this component to keep data flowing at all times.

2) Semantic Analyzer

Semantic analysis is the science of figuring out the meaning of linguistic input [12]. Semantic analysis deals with processing of an entered language in an effort to gain some knowledge of what the data entered should represent.

The BDS API uses semantic analysis to gain knowledge about what the application wants to pass, such as structures, datasets and other forms of data. Through this process the raw data can be identified and then converted to a new format fitting the language to which it will be passed.

Since the Semantic Analyzer requires quite a bit of knowledge about the programming languages only a few languages (i.e. C, Java, Cobol, FoxPro and Erlang) were selected for version 1 of the BDS API and on future iterations more languages will be added. In a nutshell, this component allows the BDS API to understand varying structures of the different languages and handle them effectively.

3) Language Knowledgebase

To do semantic analysis, the Semantic Analyzer has to have knowledge about the structure of the language. The Language Knowledgebase stores all information related to a language in respect to the needs of the BDS API.

At the core, the knowledgebase is a warehouse of structures, standards and functions that are most commonly used in each programming language. The major drawback of using this design is that it requires the storage of a potentially large amount of data. It will also take time for specific data to be found and updated (as languages evolve). To address this issue, the knowledgebase uses a clustering technique where languages that use similar structures for certain instances, are grouped together to save on space and time overhead to access it.

4) XML and JSON Parser

Both XML and JSON are easily parsed [13, 14], which makes them good candidates for cross language support and communication. This component can create and read data in these formats. The benefit is that XML and JSON complies with current standards; also, we are able to work with applications that are already built to expect data in these formats.

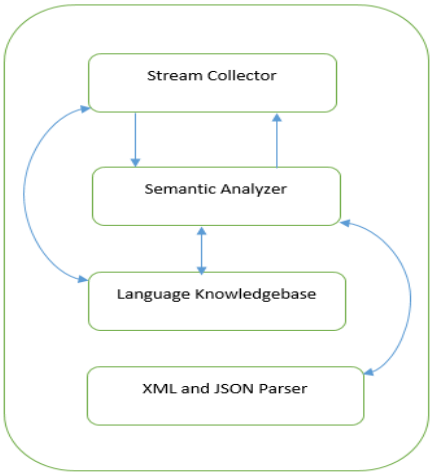

Fig. 5. BDS API Component Communication

Fig. 5 shows the communication between each component of the BDS API. The Stream Collector only communicates with the Language Knowledgebase when it is either establishing a new stream or terminating an existing stream. For all other issues, the Stream Collector communicates with the Semantic Analyzer. Upon receiving a request from the Stream Collector, the Semantic Analyzer deciphers the message with the help of either the Language Knowledgebase or the XML/JSON parser and then re-constructs a new message based on the encoding criteria for the language to which the message will be sent. The result is then returned to the Stream Collector, which sends the message through the stream to the destination application. The destination application may or may not be the application that made the initial call.

B. Initial set of BDS API Methods/Functions

The BDS API contains several functions/methods aimed at, but not limited to, the following: connecting streams, interfacing with Hadoop (we use this as the Big Data platform for version 1 of the BDS API), and transferring of files. Due to the number of functions, only a few where selected for presentation in this paper; based on their importance.

1) General Purpose

connect/0 – The connect/0 function connects an application stream to the BDS API. This function returns a reference number to be used for later connections.

connect/1 – The connect/1 function connects one application stream to another application stream or to Hadoop. It returns a connection reference to be used later by other functions. The function accepts 1 parameter of type string, which can either be the reference number for the stream that it

should connect to or the word 'hadoop' to identify connection to the Hadoop instance that it is running inside the API.

connect/2 – The connect/2 function connects one application stream to another application stream and also specifies the type of the data that should be passed through the stream. It returns a connection string to be used in later functions. The type of data can either be: XML (representing that data should be passed in XML format), JSON (representing that data should be passed in JSON format) or native (representing that data should be passed in a byte array format).

terminate/1 – The terminate/1 function accepts a connection string and terminates the connection. It should be noted that after this function is used the connection string will become unusable and errors will be thrown if an attempt is made to use it.

2) Language Communication

send_data/2 – The send_data/2 function allows data to be sent over a stream. It accepts two (2) parameters: a connection string (which must be an un-terminated reference string) and the data that should be passed. It returns a value indicating whether the data was successfully sent or not.

send_data/3 – The send_data/3 function allows data to be sent over a stream. It accepts three (3) parameters: a connection string (i.e. an un-terminated stream that has been returned from either connect/1 or connect/2 function), the data that should be passed, and the amount of bytes it will take up. This function is used when the application or language accepting the data needs to prepare some storage mechanism before persisting the data. It returns a value that states whether the data was successfully sent (or not).

send_data/4 – The send_data/4 function, like its predecessors, sends data across a connection, but it allows for the user to override the type of data to be transferred for one transaction. This is used in cases where the user needs to send data in a specific format once and wishes not to create a new connection. It accepts four (4) parameters: a connection string (similar to the other cases), the data that should be passed, and the amount of bytes it will take up (this value can be declared as null, this will let the API know that no byte size should be transferred) and the type of data to be passed. It returns a value stating if the data was successfully sent.

3) Hadoop

mapper/2 – The mapper/2 function allows the user to set the location of the mapper file that will be used in the Hadoop MapReduce operation. It accepts two (2) parameters: a valid, un-terminated connection string, and the location of the mapper file on the system. It returns a value that states if the value was set.

reducer/2 – The reducer /2 function allows the user to set the location of the reducer file that will be used in the Hadoop MapReduce operation. It accepts two (2) parameters: a valid, un-terminated connection string, and the location of the reducer file on the system. It returns a value that states if the value was set.

set_input_location/2 – The set_input_location/2 function allows the user to set the input file or folder location that will be used for the MapReduce operation. This function accepts two (2) parameters: a valid, un-terminated connection string and the location of the input file or folder on the system. It returns a value whether true or false stating if the value was set.

set_output_location/2 – The set_output_location/2 function allows the user to set the output file or folder location that will be used for the map reduce. This function accepts two (2) parameters: a valid, un-terminated connection string and the location of the output file or folder on the system. It returns a value that indicates if the value was set.

run_map_reduce/0 – the run_map_reduce/0 function starts the Hadoop MapReduce job. If any of the above functions are not properly executed, errors are thrown.

4) Data transfer

transfer_data/3 – The transfer_data/3 function allows for data to be sent across applications through the use of the BDS API streams. It accepts three (3) parameters: a valid, un-terminated connection string, the data that should be passed, and a Boolean value to indicate whether to send in parallel or not. This function may be viewed as being the same as the send_data functions, but it differs in the fact that it is used for transferring large data sets across platforms while the send_data functions are used for simple message communications.

Now that we have introduced the BDS API, let's explore how it can be used in real world situations.

C. Implementation

The BDS API as a proof of concept was implemented as a standalone server written in the JAVA programming language, using server socket and streams to provide cross communication language support. Data is transferred as byte arrays using the UTF-8 Standard over these streams. Byte arrays were chosen because the languages chosen for version one all adhered to the UTF requirement.

To track all application currently connected to and through the BDS API the HashMap class in Java was used for quick and easy validation of elements. Upon successful connection sessions to the API from a client, each session is assigned a unique reference number. This reference number is required for every other command used on the API excluding the Hadoop commands. Also this reference number is stored in the HashMap with other information about the connected node so that it can be easily acquired if so desired by another connecting node. To communicate with a subsequent client session which is registered with the API a further connection needs to be initiated (see the connect/1 or connect/2 documentation in Section IV sub section B) this connection is also given a reference number to be used through the application.

As suggested above the byte array is used to transfer data between the end users of the API. Once the byte array is received it is converted into the format which the API requires (this is based on which block of the execution the API call at

that instance has dereference). For example if the user is currently attempting to perform an API command byte array would be converted into a string and then checked against known commands. After a successful match then the command would execute as is required. An example of a command can be seen below in listing 2.

```
byte[] word = new byte[1024];
istream.read(word);
String command = new String(word).trim().replaceAll("\\s+","");
```

Listing 1: Showing array to string conversion

```
if (command.equals("send_data/2")) {
        byte[] reference_byte = new byte[1024];
        istream.read(reference_byte);
        String reference = new
            String(reference_byte).trim();
        byte[] message_byte = new byte[1024];
        istream.read(message_byte);
        String message = new String(message_byte).trim();
        date = new Date();
        logged.info("Attempting to retrieve connection for
            reference: " + reference + ": " +
            dateFormat.format(date));
        if (connectionList.get(reference) != null) {
            logged.info("Successfully retrieved connections for
                reference: " + reference + ": " +
                dateFormat.format(date));
            ConnectedSocket connection =
                connectionList.get(reference);
            SocketClass local =
                connection.getSocketToSendToFromRefere
                nce(this.communicateID);
            logged.info("Attempting to send message: " +
                message + " across connection refere: " +
                reference + ": " + dateFormat.format(date));
            String to_return = "Message " + message + "
                sent through conencted stream with
                reference " + reference;
            byte[] result_sender = to_return.getBytes();
            byte[] result_receiver = message.getBytes();
            ostream.write(result_sender);
            local.os.write(result_receiver);
            logged.info("Sent message: " + message + "
                across connection reference: " + reference +
                ": " + dateFormat.format(date));
        }  else {
            date = new Date();
            logged.info("Retrieval of connection failed, no
                such connection reference ("+reference+"):
                "+dateFormat.format(date));
            String to_return = "No socket connection exist
                with that reference.";
            byte[] result = to_return.getBytes();
            ostream.write(result);
        }
}
```

Listing 2: Showing sample code for the send_data/2 command

In handling various data structures such as lists, arrays and so on; an approach was taken where the data structure requirement were sent blocks at a time to the API which in turn would then be sent to the receiving node as plain text and a special identifier used to mark the ending of the stated structure.

For syntax and semantic analysis of the data structures referenced by the BDS; each language instance that is supported by the BDS launches its own instance or function call to the syntax and semantic parser invoked as stored procedures from the BDS API. In other words where the BDS API as service runs a non native language call, the various data structure translations of the non native language API is decoded by the syntax and semantic parser translation routines running within the BDS. To use the translation process features of our BDS API the convert/4 command is used (this command is not explained in the list above), upon executing this command and passing the required input parameters (i.e. language conversion input source, language conversion output source, the language name, language paradigm type, the name of the file to be returned at the language output source and the data content of the file involved in the language translation). The BDS API will read data parameters given data line by line each time looking for mentioned specific keyword input parameters so that it can identify the feature of the language to be converted. We make the tacit and complicit assumption that the correct language keyword input parameters have been specified to the BDS API. Upon finding this keyword it then parses the statement and finds the corresponding feature in the language it is converting to, then the parsed statement is then reconstructed in the required format. An example of a C to Java conversion can be seen below

```
#include <stdio.h>
int main()
{
        int age;
        printf("Enter age: ");
        scanf("%d",&age);
        printf("Your age is: %d \n",age);
        if(age>18){
                printf("you are old!!!");
        }else{
                printf("you are young!!!");
        }
        return 0;
}
```

Listing 3: Showing a simple C code

Statements such as int age would be noticed through syntax analysis as declarative statements while a printf statement would be recognized as an output statement, the code snippet would be converted to its Java equivalent as seen below.

```
import java.util.Scanner;
public class hello{
        public static void main(String[] args)
        {
                Integer age;
                System.out.print("Enter age: ");
                Scanner scan = new Scanner(System.in);
                age = scan.nextInt();
                System.out.print("Your age is: "+ age+" \n");
                if(age>18){
                        System.out.print("you are old!!!");
                }else{
                        System.out.print("you are young!!!");
                }
        }
}
```

Listing 4: Showing the C code of listing 1 transformed into Java code

Concurrency management is handled through threads in the JAVA programming language, upon first contact with the BDS API the connection is given a threaded class to be used throughout the life of the communication, this class contains the command references that are used to execute the commands on the API and also contains a static hash map which is used to track other connected clients and a second hash map to store connections between clients. The static list contains socket references for each client, this allows for each threaded class to communicate with any connected node without being required to create that specific socket connection. A thread is terminated by the client sending the terminate command. Fault tolerance is achieved via a combination of validation steps by the use of try/ catch blocks. The designers of the API believed checking for failures is a key debugging requirement of the BDS. The default assumptions Java's exception handling are well understood as suitable fault tolerance and security feature and hence explains our preference in how we designed this API. Before any command can be executed the static hash map class is checked to ensure that the current running thread exceptions for example is valid in its existence (otherwise the transaction thread is immediately terminated at the function call), if the command to be executed requires communication between two (2) of the BDS API's clients then the other client's reference number is also checked for validation along with the overall client connection. If any of these cannot be validated then the process will not be executed and the relevant parties notified. Each failure or exception thrown in the API will be logged for review later on, this review will be used to aid in making the API more resilient. Also the threaded class from which the exception was raised will force cancellation on the process which triggered the exception and alert the relevant parties of this exception. No other process will be affected by the failure.

V. Use Case Scenarios

We have previously hinted at ideal usage scenarios for the BDS API. In this section, we will explicitly highlight the most compelling cases.

A. Migrating Existing Data Systems

The BDS API provides easy migration of system information; with regards to the data involved. This API allows for large unstructured data sets to be passed from one system to the next through the use of file streams and fault tolerant atomic states embedded within our code design.

System migration is a hassle because companies have to spend excessive amounts of time and resources preparing the data for transfer. They sometimes require the building of facilities so that nothing obstructs the transaction; the BDS API would provide an alternative solution to this problem.

Through the use of file streams that connect to the different systems, data can be passed easily from one system to the next without the hassle that exists in a normal data migration. Also the built-in fault tolerant model of the API would add a failsafe to protect the loss of data.

Assume there exists *System A*, and an existing requirement to migrate the resources found on *System A* to another existing *System B*. The BDS API user would utilize the function connect/0 and connect/1 to connect both Systems together. The transfer_data/3 function would then be called by *System A*; passing in the third value as true to start the streaming of data to *System B* and the API would handle all necessary fault tolerance information and data transfer. After the data has been migrated *System B* would then check to ensure all packets were received and then terminate its connection.

B. Multiple Platforms Performing a Single Task

As the BDS API is platform independent, it enables cross-platform task execution and management. The BDS API could turn a normal single node process into a multiple node process through the utilization of its streaming interface and the management of these streams that it directly provides. For this type of support the API would need to be able to manage concurrent states and fault tolerant like our BDS API.

Consider some *Application A* that exists on several different systems, with all *Application A's* performing the same task. The BDS API, could connect all *Application A's* to a single master application and then utilizing the connect/0, connect/1 and send_data functions could distribute individual commands to the said applications. The API would further gather responses from the various slave nodes. The API would handle all platform specific concerns and distribute the correct commands accordingly. This enables the management of several applications across platforms.

C. Language Interoperability

As mentioned earlier, the BDS API supports communication among different programming languages; using the appropriate interface descriptor language (IDL) as required.

For example, consider an application that is built in Cobol that has a need to communicate with an application that is built in C#. Since the language utilizes the API as a medium it would allow for established session communication between the languages. Our API supports IDLs with the Cobol Object plug-ins. This converter allows Cobol, which is highly procedural, to now run as an object-oriented feature that is

popularly referred today as OCobol. This scalability feature of the BDS API enables proper acknowledgement of any external language interfaces using these IDL plug-ins. Built into our API design is a compiler syntax and semantic analyzer that allows for atomic state reduction of any language input to safeguard against side effects. Hence, when a message is passed from one language to the next it can be easily converted unambiguously. We have deliberately not indicated any preference in programming language paradigm as we believe that with time, suitable interface descriptors across language platforms will be seamless; especially as attempts are made to infuse extensible API support or IDLs between legacy and emergent languages.

VI. Related Work

The idea of making Big Data platforms, like Hadoop, more accessible is not novel. There are several ongoing efforts in the well established literature that seek to extract the benefits from storing, organizing, analyzing and searching Big Data. In this section, we present the related literature and how our own work fit into this emergent space. To date the BDS API is the only proof of concept tool as far as we have seen in the peer reviewed literature that provide multi-tier interface descriptive language functionality for the big data environments.

A. HIVE

Hadoop requires developers to write custom programs that are hard to maintain and reuse. Hive was created to solve this problem. It is an open source data warehousing solution built on top of Hadoop and it support queries expressed in a SQL-like declarative language called HiveQL. Statements in HiveQL are compiled into MapReduce jobs and executed [15]. Our BDS API dereferences these constructs to support its full program execution.

Where Hive was built to aid developers in writing better MapReduce jobs that are efficient and scalable it is clear that real benefit is to assist language developers who need to work on an enterprise scale with this Big Data. However, Hive is limited in its usage because it largely deals with bulk data sets even though it uses similar constructs to that of SQL which exists in Relational Databases. HiveQL or Hive should not be viewed as a relational database management system (RDMS) but as a batch processing integration tool for the Hadoop framework. A prime example is that Hive does batch deletes as opposed to individual deletes which are allowed in the RDMS. It should also be noted that it doesn't offer support for cross-platform or inter-language communication that is now supported by our BDS API as a key distinguishing feature on how we supplement the gap in this literature.

B. LiquidFiles

We model components of our BDS API using LiquidFile architectures. LiquidFiles is a web-based API that accepts medium to relational database files and splits them up into smaller blocks of data to be proccessed. These smaller blocks of data are normally around 100 megabytes in size. It is easily scalable and can work with all files sizes. It also uses XML file formatting to communicate with multiple languages and can built-in security features [16].

As LiquidFiles is ideally web-based, it assumes default Internet connectivity to maintain a persistent state . This feature may be a handicap for users without persistent Internet connection. Currently, LiquidFiles do not have support for unstructured data sets running on Hadoop. Our naïve toy experiment testing shows that liquidfiles persistent state connectivity to handle unstructured petabyte or exabyte file limits is unstable. This requires traditional developers to create customized connection mechanisms to these platforms which would handle these new data formats. This transition however is expensive both in terms of time, money and techincal infrastructure adoption unless someone can prove this to us otherwise, but we have not seen any new evidence to show this at least up to the time we had been preparing this paper.

C. BULK API

Our efforts were also motivated by looking at the Bulk API, which is used to query or modify a large number of records asynchronously. This was achieved by submitting batches that are running as background processes. It is REST-based[2] and is optimized for handling relatively large sets of structured data. Bulk API is designed to handle records when the data sets contain a couple thousands to millions of records up to mega-byte and even terabyte scale [17]. We reasonably argue that our BDS API enables a wider range of functionality than Bulk API that handle peta-byte scale data limits and beyond.

D. Google BigQuery

Google BigQuery is a tool, which enables users to right fast SQL-like queries (the official dialect used is BigQuery's SQL dialect) against large data sets using the processing power of Google [18]. BigQuery can be accessed by varying means such as: a browser tool, a command line tool, calling the BigQuery REST API or client libraries (JAVA, PHP and Python) [18]. Data however either has to be stored on Google's cloud server or streamed into the API for use.

The difference between our BDS API and the service offered by BigQuery is the scalable multiple language syntax support as compared to google BigQuery that adhere to a single native SQL dialect.

E. Oracle XQuery

Oracle XQuery for Hadoop is a transformation engine for semi-structured big data [19]. Oracle XQuery runs in the XQuery language and it transforms commands into a series of

[2] REST (Representational state transfer) is an architectural style consisting of a coordinated set of constraints applied to components, connectors, and data elements, within a distributed hypermedia system.

MapReduce jobs that are executed (in parallel) on the Hadoop clusters. With this solution, one can focus on data movement and transformation logic, instead of using Java or MapReduce; which carry their own levels of complexities, without sacrificing scalability or performance [19]. Unlike other Hadoop vendors, the data to be processed XQuery does not only have to be located on the Hadoop Distributed File System (HDFS), but it can also be stored in an Oracle NoSQL Database. Oracle's Big Data also contains advance R connectors for better statistical algorithm applications [20].

Oracle XQuery provides good vendor specific support for its user-driven queries running in NoSQL databases albeit Hadoop service enabled or not. We also realize that one of the limitations of the Oracle XQuery over Hadoop is its lack of support for non native language communication or process management APIs. These observations have driven our motivations to design the BDS API as an open source scalable API that improves on the Oracle XQuery functionalities.

VII. Future Work

The research team recognizes that there is a lot of further work to be done and as such have developed a roadmap for same.

A. Knowledge Base Constraints

Currently, the BDS API parser depends on knowledge of the different languages to properly function (as at present the current version only supports a limited set of language scalabilities). To address this, research will be undertaken to create a new binary language that runs the BDS API as its own embedded virtual machine (VM) operating system language. This virtual machine language API will autonomously convert binary data between non native language instances to ensure we strive for the ideal when we mention scalability and platform independence.

B. Restricted Local Customization Capabilities

Currently the system is incapable of allowing local customization of platforms via a suitable visualization front end. We mostly drive the system functions of the BDS API using command line interactivity and limited graphical end user interface (GUI) support. In other words it would be convenient for us to have a sophisticated front end, with intelligent human computer interactivity (HCI) to the BDS API. This would allow the end users of our tool to seamlessly visualize all the language interdependency communication scenarios across multiple language platforms that are interacting on this Hadoop framework using our BDS API. This approach invariably would also allow us to build access control functionalities into the GUI front end version of the BDS API as a security feature that will allows us to separate the API from direct manipulation while it provides customized and personalized interfaces for the end user interaction within the scalable language hadoop environments.

C. Mobile Computing and Communication Capabilities

We intend to extend the BDS API as a mobile application support feature. This opportunity allows us to expand the outreach of access to end users of our tool e.g. mobile companies.

D. Increased Platform Support

The current instance of the BDS API only supports Hadoop. We will generalize the API to support any arbitrary Big Data platforms in future.

E. Autonomy

Where our BDS API can run as its own autonomous agent, the future design should allow for dynamic state interaction with big data platforms, based on unsupervised goal setting on task. This could allows us to define autonomous MapReduce functionalities that will enable improved computational efficiencies on both unstructured data and meta data now generated in these large environments. It goes without saying that our tool represents an Internet of Signs (IOS) philosophy that should be able to differentiate large state data structures sets in real time, apply the suitable syntax and semantic data transaction analysis as a feature of the API parsing. These underlined new functionalities become relevant as Big Data platforms keep changing.

The suggestions described above fits into the realm of other ongoing work within our research group on autonomous interface descriptive languages (IDL); where we explore various deep learning techniques as a neural network language translation analysis service. These expectations allows to design the BDS API as a "dynamic learning service".

Parts of the current work on the BDS API will be available within an open stack so that contributing developers can reuse and test new embedded functionalities. This approach will allow us to gain constructive feedback on the approaches we have used, while we improve our existing versions.

VIII. Conclusion

Given the need for firms to extract value from the vast amounts of sparse flowing data that is currently being generated from within various ubiquitous software as service(SAAS) programming language applications environments found everywhere today, this need exacerbates the concerns that finding suitable tools that harness the power of existing and future Big Data platforms cannot be underestimated. In this paper, we presented the Big Data Services (BDS) API as a proof of concept tool to assuage the concerns around migration and leveraging contemporary unstructured data within evolving data center environments. The BDS API's contributory support for legacy and emergent languages running over a Hadoop framework minimizes the need for full-scale system migration to new independent language platforms. This work is the first of its kind anywhere in the current literature as far as we know.

Through the use of language translation analysis techniques which apply the use of data streams, the BDS API communicates seamlessly with such data streams (i.e. petabyte scale and greater). The BDS API is language independent by design as it allows inter-process communication between tiers of SAAS enabled legacy and emergent languages. The API has in-built fault tolerance, concurrency management and

bottleneck detection features that allows it to be an enabler for large unstructured data center environments.

REFERENCES

[1] P. Simon. *Too Big to Ignore: The Business Case for Big Data*. N.p.: Wiley, 2013. ISBN-13: 978-1118638170.

[2] H. Barwick, "Lack of in-house skills a barrier to big data adoption in A/NZ: report" CIO Magazine, November 28th, 2013. Retrieved from http://bit.ly/1dS41yD on February 19th, 2014.

[3] P. Simon, "Why Big Data In The Enterprise Is Mostly Lip Service," Information Week, Feb 18th, 2014. Retrieved from http://ubm.io/MeGug1 on Feb 19th, 2014.

[4] Oracle. "Oracle: Big Data for the Enterprise." http://www.oracle.com/us/products/database/big-data-for-enterprise-519135.pdf.

[5] M. Beyer, "Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data." Gartner. June 27, 2011. Retrieved from https://www.gartner.com/newsroom/id/1731916 on February 19th, 2014.

[6] D. Laney. "The Importance of 'Big Data': A Definition." Gartner. June 21, 2012. Retrieved from https://www.gartner.com/id=2057415 on February 19th, 2013.

[7] R. Natarajan, "Apache Hadoop Fundamentals - HDFS and MapReduce Explained with a Diagram," The Geek Stuff. Retrieved from http://bit.ly/MeI8OI on Feb 19, 2014 .

[8] D . Jeffrey, and S. Ghemwhat, "MapReduce: simplified data processing on large clusters." *Communications of the ACM - 50th anniversary issue*: 1958 - 2008 1 Jan. 2008: 107-13. Print..

[9] 3Scale. (n.d.). What is an API. In *3Scale*. Retrieved February 19, 2014, from http://www.3scale.net/wp-content/uploads/2012/06/What-is-an-API-1.0.pdf.

[10] Ross, D. (n.d.). How to Leverage an API for Conferencing. In *How Stuff Works*. Retrieved February 19, 2014, from http://money.howstuffworks.com/business-communications/how-to-leverage-an-api-for-conferencing1.htm

[11] Oracle. "Overview of STREAMS." http://docs.oracle.com/cd/E19683-01/806-6546/6jffu9853/index.html.

[12] A. Klapuri, "Semantic analysis of text and speech." Institute of Signal Processing, Tampere University of Technology. http://www.cs.tut.fi/sgn/arg/klap/introduction-semantics.pdf.

[13] E. T. Ray, *Learning XML*. 2nd ed. N.p.: O'REILLY http://oreilly.com/catalog/learnxml2/chapter/ch02.pdf.

[14] P. Hunlock, "Mastering JSON (JavaScript ObjectNotation)." http://www.ebooks.shahed.biz/JS/JSON/Mastering_JSON_%28JavaScript_Object_Notation%29.pdf.

[15] A. Thusoo, S. S. Joydeep, J. Namit, S. Zheng, C. Prasa, et al. *A Petabyte Scale Data Warehouse Using Hadoop*. N.p.: InfoLab Stanford http://infolab.stanford.edu/~ragho/hive-icde2010.pdf.

[16] LiquidFiles. "LiquidFiles is a Virtual Appliance that helps companies and organisations Send & Receive Large Files, Fast & Securely." http://www.liquidfiles.net/.

[17] SalesForce. *SOAP API Developer's Guide*. Vol. 30. N.p.: salesforce.com, 2014. http://www.salesforce.com/us/developer/docs/api/apex_api.pdf.

[18] Google Developers. "Google Big Query.". https://developers.google.com/bigquery/.

[19] Oracle. "What Is Oracle XQuery for Hadoop?." http://docs.oracle.com/cd/E51174_01/doc.24/e51161/oxh.htm#BDCUG526.

[20] A. Woodie, "Oracle Expands Use of Cloudera Hadoop in Big Data Kit." Data Nami. http://www.datanami.com/datanami/2013-11-13/oracle_expands_use_of_cloudera_hadoop_in_big_data_kit.html.