Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 170723, 10 pages
http://dx.doi.org/10.1155/2014/170723

Research Article

REST-MapReduce: An Integrated Interface but Differentiated Service

Jong-Hyuk Park,[1] Hwa-Young Jeong,[2] Young-Sik Jeong,[3] and Min Choi[4]

[1] *Department of Computer Engineering, Seoul National University of Science and Technology, 232 Gongneung-ro, Nowon-gu, Seoul 139-743, Republic of Korea*
[2] *Humanitas College, Kyunghee University, No. 26, Kyunghee-daero, Dongdaemun-gu, Seoul 130-701, Republic of Korea*
[3] *Department of Multimedia Engineering, Dongguk University, 30 Pildong-ro 1 Gil, Jung-gu, Seoul 100-715, Republic of Korea*
[4] *Department of Information and Communication Engineering, Chungbuk National University, 52 Naesudong-ro, Heungdeok-gu, Chungbuk, Cheongju 361-763, Republic of Korea*

Correspondence should be addressed to Min Choi; miin.chae@gmail.com

Received 16 March 2014; Accepted 3 April 2014; Published 11 June 2014

Academic Editor: Laurence T. Yang

With the fast deployment of cloud computing, MapReduce architectures are becoming the major technologies for mobile cloud computing. The concept of MapReduce was first introduced as a novel programming model and implementation for a large set of computing devices. In this research, we propose a novel concept of REST-MapReduce, enabling users to use only the REST interface without using the MapReduce architecture. This approach provides a higher level of abstraction by integration of the two types of access interface, REST API and MapReduce. The motivation of this research stems from the slower response time for accessing simple RDBMS on Hadoop than direct access to RDMBS. This is because there is overhead to job scheduling, initiating, starting, tracking, and management during MapReduce-based parallel execution. Therefore, we provide a good performance for REST Open API service and for MapReduce, respectively. This is very useful for constructing REST Open API services on Hadoop hosting services, for example, Amazon AWS (Macdonald, 2005) or IBM Smart Cloud. For evaluating performance of our REST-MapReduce framework, we conducted experiments with Jersey REST web server and Hadoop. Experimental result shows that our approach outperforms conventional approaches.

1. Introduction

With the fast deployment of cloud computing, MapReduce architectures are becoming the major technologies for mobile cloud computing. Nowadays, we are experiencing a major shift from conventional mobile applications to mobile cloud computing. The demand of Open API-based development stems from the increasing use of smartphone applications [1, 2]. Community portal companies are providing Open API service for access to their service. Within a few years, we can expect a major shift from traditional mobile application technology to mobile cloud computing [3]. It improves application performance and efficiency by off-loading complex and time-consuming tasks onto powerful computing platforms. By running only simple tasks on mobile devices, we can achieve a longer battery lifetime and a greater processing efficiency. This off-loading with the use of parallelism is not only faster but can also be used to solve problems related to large data sets of nonlocal resources. With a set of computers connected on a network, there is a vast pool of CPUs and resources, and you have the ability to access files on a cloud. In this paper, we propose a novel approach that realizes the mobile cloud convergence in a transparent and platform-independent way. Users need not know how their jobs are actually executed in a distributed environment and need not to take into account their mobile platforms are IPhone or Android. All they have to do is to make use of the REST interface, and need not to know the complex distributed computing API such as Hadoop [4].

The research of MapReduce using REST web service interface is underexplored and most research efforts are still at their initial state [5, 6]. MapReduce is a programming model and an associated implementation for processing and generating large data sets. In this work, we propose a concept of REST-MapReduce enabling users to use only the REST interface without using the MapReduce architecture; it is the MapReduce framework using REST web service Open API interface. We combine MapReduce and REST Open API into an integrated service as REST-MapReduce [7, 8]. This is because of the slower response time for accessing simple RDBMS on Hadoop than direct access to RDMBS [9, 10]. The slow response time stems from the fact that MapReduce was originally designed for analyzing large data, not for simple RDBMS lookup. Such a job, scheduling, initiating, starting, tracking, and management during MapReduce execution, is not a necessary task for REST Open API service execution. To avoid such a problem, REST-MapReduce framework provides an integrated interface with high performance that supports both REST Open API and MapReduce. At the same time, the REST Open API service is provided by a separated architecture for the REST Open API service with a separate architecture. Likewise, we can overcome such a slow response time of simple RDBM invocation on Hadoop by this integrated interface, but differentiated service.

The rest of this paper is organized as follows. Section 2 describes related works. Section 3 explores the architecture of MapReduce computation processes in our REST-MapReduce framework. Section 4 presents the platform independent implementation of application using the REST-MapReduce interface. Section 5 shows performance evaluation. Finally, we conclude and summary our work in Section 6.

2. Related Works

Before we go into more detail, we briefly introduce the REST Open API-based mobile application development approaches. To communicate with remote procedure call between client and server, the interface should be defined first. To this end, web service description language (WSDL) and remote procedure call (RPC) were used for the specification. But, these previous approaches are relatively complicated and highly overloaded. Recently, representational state transfer (REST) architecture was first introduced by Fielding. REST web service is becoming popular and explosively used in the field of application development of web and smartphone. Therefore, today's many Internet companies already provide their services by both traditional SOAP-based web service and RESTful web services [11, 12]. The main differences between REST web service and SOAP/WSDL web service are as follows: due to the complicated characteristics of SOAP-based web services, REST web service has not been introduced. REST web service removes the overhead from encoding/decoding of header and body during message transfer. The REST web service enables users and developers to easily use the web services at remote or local sites. We need not add additional communication layer or protocols for REST web service, but we can easily achieve scalability and performance. This research evaluates the performance of mash-up architectures through RESTful Open API web services on smart mobile devices. It provides the analytical experimental results for the performance evaluation of system models. Especially, we try to find an optimal number of parallel REST web server architectures under certain request arrival rates. We show the performance of the proposed architecture, especially the mean number of requests in the queue and the mean waiting time.

REST web service is a core technology for smartphone application development. This is because REST web service is the most appropriate way for accessing information through the Internet. Usually, a smartphone application needs information from several sources of (one or more) REST web services [13]. So, we need to utilize two or more REST web services composition to realize a target application [14, 15]. In this paper, we propose a server architecture for managing REST web services. This server is for managing web services so as to provide web server maintenance, especially on composition, deployment, and management of REST web services. It enables service developers to conveniently develop, deploy, upload, and run their composed web services with the use of general OOP languages [16].

In 2004, the concept of MapReduce [17] was first introduced as a novel programming model and implementation for a large set of computing devices. Map generates a set of intermediate key/value pairs and reduces merges all intermediate values associated with the same intermediate key, so that programs with this are automatically parallelized and executed on a large cluster of computing devices [18, 19].

Apache Hadoop has become the de facto standard for managing and processing hundreds of terabytes to petabytes of data. It is an open-source Java software framework that supports massive data processing across a cluster of servers. It can run on a single server, or thousands of servers. Hadoop uses a programming model called MapReduce to distribute processing across multiple servers. It also implements a distributed file system called HDFS [20] that stores data across multiple servers. Hadoop monitors the health of servers in the cluster and can recover from the failure of one or more nodes. In this way, Hadoop provides not only increased processing and storage capacity but also high availability. Hadoop [4] is actively used these days by Amazon/A9 [21], Facebook, Google, IBM [22], Joost, New York Times, PowerSet, Yahoo, and so on.

3. REST-MapReduce Framework Architecture

This research focuses on designing the concept of MapReduce using the REST Open API interface. This means that both interfaces of REST Open API and MapReduce are integrated into a REST Open API interface. We provide a higher level of abstraction by integrating those two different types of access methods, such as REST Open API and MapReduce. The abstraction by integration provides higher abstraction for both REST API and MapReduce. Users need not to recognize or differentiate how to use those two interfaces, respectively. This is good for user convenience, but it is known to have lower performance when simple RDBMS access occurs on MapReduce servers. This is because MapReduce was originally designed for analyzing large data through parallel execution among multiple cluster nodes. To avoid such an overhead, we proposed a novel architecture as follows.

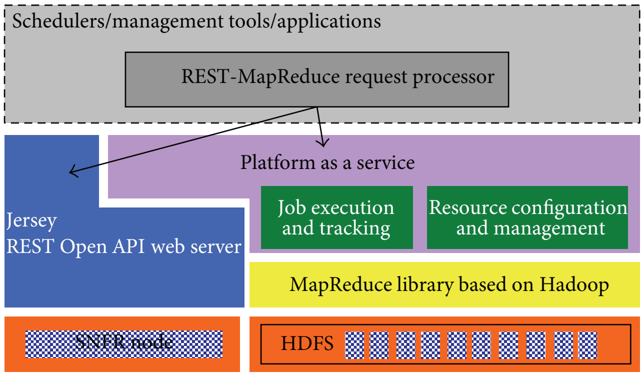
Schedulers/management tools/applications
REST-MapReduce request processor
Platform as a service
Jersey
REST Open API web server
Job execution and tracking
Resource configuration and management
MapReduce library based on Hadoop
SNFR node
HDFS

Figure 1: REST-MapReduce framework architecture.

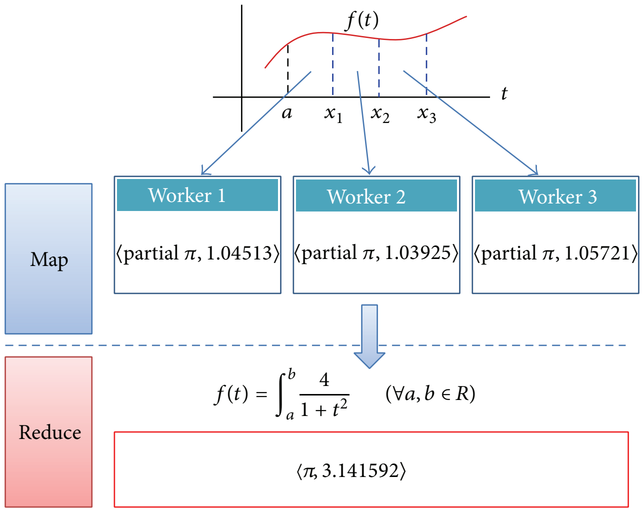

Figure 2: MapReduce computation process on our framework.

3.1. Architecture. Figure 1 depicts the architecture of our REST-MapReduce framework. It has five core components: applications, Jersey, platform as service, MapReduce library, and HDFS/S3. First, REST-MapReduce Request Processor acts as the role of a service differentiator in this framework. It determines whether the incoming request is for REST Open API or MapReduce. Then, it sends to Hadoop or Jersey depending on the request type. Second, Jersey is the open source JAX-RS (JSR 311) Reference Implementation [6] for building RESTful Web services. Jersey provides an API so that developers may extend Jersey to suit their needs. We make use of both Tomcat and Jersey to implement our systems. Platform as a Service is achieved by Hadoop. The MapReduce library, job execution, job tracker, and resource management schemes are from the Hadoop. Third, HDFS stands for Hadoop distributed file system, whereas SNFR stands for special node for fast responses [23].

The general concept is that a user submits a job to our REST-MapReduce framework. Then, the REST-MapReduce request processor determines whether the request is for REST Open API or MapReduce. Then, it sends it to either Hadoop or Jersey depending on the request type. Information about the type of the incoming request is necessary for the initial job placement to maximize resource utilization and also that of the entire system. This is because the most appropriate node to execute the task is determined by the type of request. If it is a REST API call, it is better to be forwarded to Jersey server due to its performance, whereas if it is a MapReduce request, it should be forwarded to Hadoop server because of its nature of the parallel execution. The user client can communicate with the PaaS components, such as Resource Configuration & Manager, using the client tool to first acquire a new connection and then submit the application to be run via ClientRMProtocol#submitApplication. As part of the ClientRMProtocol#submitApplication call, the client needs to provide sufficient information to the ResourceManager to “launch” the application’s first container, that is, the ApplicationMaster. You need to provide information such as the details about the local files/jars that need to be available for your application to run, the actual command that needs to be executed (with the necessary command line arguments), any Unix environment settings (optional), and so forth. Effectively, you need to describe the Unix process(es) that needs to be launched for your ApplicationMaster. Due to the integration, there are somewhat different features in requests through REST Open API from smartphones. Almost all requests are usually simple data lookup, whereas the rest of them are task/data parallel operations. Therefore, we focus on differentiating those two operations to increase response time.

Figure 2 shows the overall flow of a MapReduce computation process in our REST-MapReduce architecture. When a new job is submitted to a system, a global job scheduler selects the most preferable node for the job to be executed (mapping strategy). Then, the Hadoop JobTracker monitors the job by keeping track of change of job resource usage during the execution. Let us take a look at the procedure in detail. When the user program calls the REST Open API, the following sequence of actions occurs. $s^*(s^*1)$ The job execution and MapReduce module split the pi value calculation workload into multiple nodes. Then, it starts up on multiple workers of the Hadoop cluster. Our approach is different in terms of task parallelism and not data parallelism. Typically, previous researches in the field of big data processing on Hadoop cluster usually focus on data parallelism, distributing the data of 16 megabytes to 64 megabytes (MB) per piece through the Hadoop cluster. $s^*(s^*2)$ One of the workers (the workers run on nodes called DataNodes or slaves, interchangeably) has a special purpose. It is a master node. The master node reduces tasks to be assigned. The master picks idle workers and assigns to each one a map task or a reduce task. The rest are slave workers. The slaves are configured in conf/slaves of the Hadoop configuration. They initially join into the framework on system bootup. Once they have joined the framework, the master node sends a short heartbeat message to every worker periodically. If there is no response from a worker within a certain amount of time, the master checks the worker as failed. $s^*(s^*3)$ After completion of the distributed workload calculation, the Reduce worker iterates over the sorted intermediate data and, for each unique intermediate key encountered, passes the key and the corresponding set of intermediate values to the user's Reduce function [1]. In this work, we eliminate data dependency through workload parallelization, if any exists, between the workloads of slaves. This is because the data dependency leads to performance degradation, severely resulting in sequential execution. s*(s*4) Map phase generates computation result as a form of key-value pairs into log files (e.g., <partial pi>, <1.05721>). The Map function takes a log line, pulls out the timestamp field when the server finished processing the request, converts it into a minute-in-week slot, and then writes out in file systems. Reduce phase reads and sorts all intermediate data so that all occurrences of the same key are grouped together, resulting in the final result which is numerically added for all the same keys. This is the reason why we can see the final pi value as 3.141592 in Figure 2.

3.2. Task Parallelization Phase. In this section, we show a development procedure of the cloud-based applications on a mobile platform, especially π calculation. The first step in this procedure is to identify sets of tasks that can run concurrently and/or partitions of data that can be processed concurrently. The second step is to eliminate dependency, if any exists, between every computational phase in the algorithm. The dependency limit of the degree of parallelism results in performance degradation. π is a mathematical constant whose value is the ratio of any Euclidean plane circle's circumference to its diameter; this is the same value as the ratio of a circle's area to the square of its radius. Many formulas from mathematics, science, and engineering involve π, which makes it one of the most important mathematical constants. The simplest method to calculate π is circumference divided by diameter [24]. However, it is difficult to get the exact circumference using this simple method. As a result, there are other formulas to calculate π. These include series, products, geometric constructions, limits, special values, and pi iterations. To calculate π through mobile cloud convergence, we first need to convert the algorithm into a parallelized version. We present a π calculation with infinite series that puts forth a parallelization method for ease of application on the mobile cloud convergence. To calculate π, we first show the procedure of parallelizing the pi calculation as follows:

$$\begin{aligned} P_n(x) &= f(c) + f'(c)(x-c) \\ &\quad + \frac{f''(c)}{2!}(x-c)^2 + \cdots + \frac{f^{(n)}(c)}{n!}(x-c)^n, \end{aligned} \tag{1}$$

where $P_n(x)$ is defined by the Taylor series. $P_n(x) = \sum_{k=0}^{\infty}(f^k(0)/k!)$, especially on $c = 0$ is known as the Maclaurin series. So, we compute the Maclaurin series generated by $f(x) = \tan^{-1}(x)$. Since we need the nth order derivative of $f(x) = \tan^{-1}(x)$, we apply this expression to the Maclaurin series. Consider

$$\begin{aligned} P_n(x) &= 0 + x + 0 + \frac{x^3}{3} + 0 + \frac{x^5}{5} + \cdots \\ &= x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots . \end{aligned} \tag{2}$$

Finally, we get the following expression from $P_n(x) = \tan^{-1}(x)$:

$$\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots + (-1)^n \frac{x^{2n+1}}{2n+1} + \cdots . \tag{3}$$

But, there is still another problem such that a function to compute this based on the above form is not appropriate for parallelization. This is because each computed value is dependent on previously computed values. Assuming we distribute this workload on eight nodes, they should not be dependent on the previous iteration and the next iteration. That means the next term calculation requires the result of previous term calculation, resulting in serialized execution in a parallelized environment. For example, considering the following expression:

$$\tan^{-1}(x) = \frac{\pi}{4}, \tag{4}$$

it is necessary to calculate the following expression:

$$\tan^{-1}(x) = 1 - \frac{1^3}{3} + \frac{1^5}{5} - \frac{1^7}{7} + \cdots . \tag{5}$$

But, for computing $(-1^7/7)$, the partial term of $1 - 1^3/3 + 1^5/5$ should be calculated a priori. Again for computing $(+1^5/5)$, the partial term of $1 - 1^3/3$ should be calculated a priori. Thus, we need to come up with a parallelized solution for the π calculation.

In this paper, we propose such a parallelized solution to distribute the heavy workloads to multiple nodes. An independent form of this equation should be provided. Therefore, we convert the equation into an integral form that is suitable for parallelized execution on MapReduce framework. We first take the derivative from the expression (3) with respect to x

$$\frac{d}{dx}\tan^{-1}(x) = 1 - x^2 + x^4 - x^6 + \cdots + (-1)^n x^{2n} + \cdots . \tag{6}$$

We replace the variable x with t for the sake of convenience:

$$\frac{d}{dt}\tan^{-1}(t) = 1 - t^2 + t^4 - t^6 + \cdots + (-1)^n t^{2n} + \cdots . \tag{7}$$

At this time, expression (7) can be simplified by

$$\frac{1}{1+t^2} = 1 - t^2 + t^4 - t^6 + \cdots + (-1)^n t^{2n} + \cdots \tag{8}$$

to

$$\frac{d}{dt}\tan^{-1}(t) = \frac{1}{1+t^2}. \tag{9}$$

Integrate this formula to infinite

$$\int_a^b \frac{t}{dt}\tan^{-1}(t) = \int_a^b \frac{1}{1+t^2} \quad (\forall a, b \in R). \tag{10}$$

Integrating this equation for the interval a to b yields the integral form of $\tan^{-1}(t)$. By substituting $\pi/4 = \tan^{-1}(t)$ into this formula, we get the parallelized form that is executable on the MapReduce platform:

$$\tan^{-1}(t) = \int_a^b \frac{1}{1+t^2} \quad (\forall a, b \in R). \tag{11}$$

By $\tan^{-1}(\tan(\pi/4)) = \tan^{-1}(1)$, we get $\pi/4 = \tan^{-1}(1)$. Finally, we make use of (11) in this expression to get the following expression:

$$\pi = 4\tan^{-1}(t) = 4\int_a^b \frac{1}{1+t^2} = \int_a^b \frac{4}{1+t^2} \quad (\forall a, b \in R). \tag{12}$$

We approximately get the π value by integrating this equation for the interval $-1/2$ to $1/2$.

Unlike an infinite series representation, the integral form is fully parallelizable and it is easy to divide the problem into chunks/parts of the work. We distribute and map these tasks onto multiple clouding nodes. However, this equation cannot be executed on cloud computing which is highly parallelized and distributed in a computing environment. This is an example of task parallelization and partitioning and can be run on a mobile cloud convergence platform.

4. REST Application Interface

4.1. Persistent Storage. In this section, we examine local storages on HTML5 web applications. Usually, we make use of cookie and session for keeping information on desktop when network connection is not available. HTML5 provides more options than conventional web development methods. LocalStorage, sessionStorage, and WebDB are like that. While all of these functionalities are applicable to both the mobile and desktop worlds, in the world of desktops you generally have a lower rate of adoption. However, any mobile device released in the last 2-3 years will support most of these specs. Moreover, the explosive use of mobile devices such as smartphones requires the demand of using HTML5 due to one source multiuse (OSMU) development. So, there are big demands of persistent storage using HTML5. The persistent storage support was in demand in the world of desktops, but, with the rise of the mobile web and edge connections, support for offline capability has exploded. Everything from offline data storage to the actual application startup is already available and supported on a wide range of mobile platforms. The HTML5 brings us to the three storage mediums: localStorage, sessionStorage, and WebDB. Luckily, the Sencha Touch data package offers awesome wrappers around all three. We can use these persistent storages regardless of the network connection status. SessionStorage is not a persistent storage, meaning it gets wiped whenever the user leaves the page or closes the application. However, in case of one-page web apps where you stay on the same page the entire time, sessionStorage can be a perfect candidate for offline data access, especially in data-sensitive scenarios where you do not want the data persisted on the device after the user is done using the app. SessionStorage is generally limited to 5 MB in size and when that is exceeded, depending on the platform, either a JavaScript error might be thrown or a popup is presented to the user asking for permission to increase the available storage.

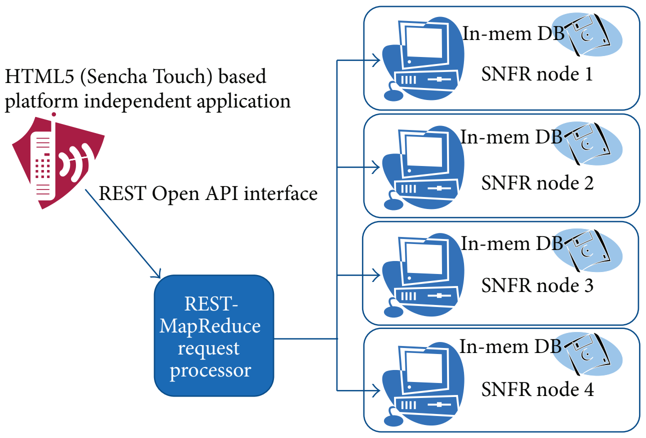

Figure 3: System integration and interactions between the components.

LocalStorage is essentially the same thing as sessionStorage, except that it is persistent. In other words, if you close the app and return, the data will still be there. localStorage is more suited for data that you want to be available when used in combination with the offline startup techniques discussed earlier. However, the localStorage still has a problem for being a perfect persistency. If you clean and delete your web cache from your browser, it will be removed from then on. So, we have to prepare a work around for compensating the cases. Sencha Touch configuration for localStorage looks almost identical to that of sessionStorage.

Finally, the web database is supported by almost every browser. Though specs call for 5 MB limit per origin, iOS has been known to allow up to 50 MB after multiple user prompts. Behind the scenes, it is an SQL database with a query-based language that many of us know and love. When it comes to configuring it in Sencha Touch, it is just as easy as with the other storage mediums.

4.2. System Components. We implemented our application exploiting 3 cutting edge technologies. Figure 3 shows the system architecture of our acquisition tax analysis application. Our system architecture consists of the following components: REST Open API server, HTML5 based Platform Independent Client Application, and Database Server. Figure 3 shows the system architecture of our platform independent application design and implementation for checking capital gain tax relief due to one house by one household.

In-Memory Database. For the high performance of database, we make use of the in-memory database in this project. Because of too many representative requests, we came up with a state-of-the-art technology for processing this type of short and high frequency requests. The best way to service these requests is the in-memory database.

REST Open API Web Service. The REST is a platform independent architectural style. REST ignores the details of

```
Object1 obj1;
String strSearchKeyword = getParameter(STR_PARAM_SEARCHKEYWORD);
String strWebSvcQuery = "http://openapi.naver.com/search?key=test&query=";
strWebSvcQuery += strSearchKeyword + "&target=adult";
URL text = new URL(strWebSvcQuery);
XmlObjectConversionFactory objCreator = XmlObjectConversionFactory.newInstance();
XmlObjectConverter xoConverter = objCreator.newConverter();
obj1 = xoConverter.setInput( text.openStream(), null );
if (obj1.getbAdult()) {
    return;
}
else
{
        try{
              strWebSvcQuery = "http://openapi.naver.com/search?key=test&query=";
              strWebSvcQuery += strSearchKeyword +
              "&display=10&start=1&target=webkr";
              URL text = new URL(strWebSvcQuery);
              String test = text.toString();
              XmlPullParserFactory parserCreator =
              XmlPullParserFactory.newInstance();
              XmlPullParser parser = parserCreator.newPullParser();
              parser.setInput( text.openStream(), null );
              String tag;
              int parserEvent = parser.getEventType();
              while (parserEvent != XmlPullParser.END_DOCUMENT ){
                    switch(parserEvent){
                    case XmlPullParser.TEXT:
                    tag = parser.getName();
                    break;
                    case XmlPullParser.END_TAG:
                    tag = parser.getName();
                    break;
                    case XmlPullParser.START_TAG:
                    tag = parser.getName();
                    break;
              }
        }catch( Exception e ){
              Log.e("dd", "Error in network call"+ e);
        }
}
```

Algorithm 1: Open API parser.

component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.

Sencha Touch (HTML5) Application. Sencha Touch is a representative HTML5 UI Framework in these days. Sencha Touch is a well-known user interface (UI) JavaScript library, or framework, specifically built for the Mobile Web. It can be used by Web developers to develop user interfaces for mobile web applications that look and feel like native applications on supported mobile devices. As shown in Algorithm 1, it is fully based on web standards such as HTML5, CSS3, and JavaScript. Sencha Touch aims to enable developers to quickly and easily create HTML5 based mobile apps that work on Android, iOS, Windows, Tizen, and BlackBerry devices and produce a native-app-like experience inside a browser.

4.3. System Implementation. Using these techniques, we developed our system and application which is platform independent one as shown in Figure 3. The application makes use of the AJAX request to the REST Open API web service as shown in Algorithm 2.

```
var button = new Ext.Toolbar({
                        cls: “calculator_button”,
                        height: 35,
                        items: [this.text,
                                    {xtype: ‘spacer’},
                                    {html: new Ext.XTemplate(‘<img style=“width:.5em;height:.5em;”
src=“resources/imgs/button.png”/>’).apply({name: ‘button’}),
                                    handler: function () {
                                              Ext.Ajax.request({
                                                    url: ‘/localhost:8080/Example/apis/example/,
                                                     params:{
                                                       action: ‘calculation’,
                                                       userid: ‘15’,
                                                       username: ‘MCHOI’,
                                                       username: ‘MCHOI’,
                                                       userDate: ‘20140315’,
userEtc: ‘etc’
HouseHolds: ‘2’
AreaofExcsSpace: ‘5’
AcquisitionPrice: ‘25000’
userContact: ‘01099695699’
                                                     },
                                                     success: function(xhr) {
                                                           var response =
Ext.decode(xhr.responseText);
                                                     }
                                              });
                                    }}
                                   ]
                });
```

Algorithm 2: A portion of our HTML5 application code.

Our application supports WebOS, Android, iOS, Window Phone, and BlackBerry. The application requires the only information of acquisition tax, the area of exclusive space, household numbers, and the location. Then, the application provides the capital gain tax result which is automatically calculated.

5. Experimental Results

We describe the experimental result for the REST-MapReduce in this section. This is because REST web service is one of the most convenient methods for accessing information through Internet. Usually, a smartphone application needs information from several sources of (one or more) REST web services. In this experiment, we adopt the Apache Tomcat 7.0 as a web application server, Jersey 1.8 for REST Open API Service Provider, and Hadoop 2.0.4 as MapReduce execution server. Apache Tomcat is an open software with Java Servlet and JavaServer Pages technologies. Apache Tomcat powers numerous large-scale web applications across a diverse range of industries and organizations. Jersey is the open source JAX-RS (JSR 311) Reference Implementation [14] for building RESTful Web services. Jersey provides an API so that developers may extend Jersey to suit their needs. We make use of both Tomcat and Jersey to implement our systems. We constructed eight-node Linux cluster of Core i5 machines, each with 4 G RAM. The machines were connected by network and managed by Hadoop [4]. Figure 4 shows an overview of our REST-MapReduce framework architecture.

Prior to evaluating the performance in detail, we present system model as a queueing network. The evaluation model of our REST-MapReduce architecture is presented in Figure 4. REST Open API Web Service is composed of three components comprising: (1) dedicated node for Jersey REST web service, (2) Hadoop cluster, and (3) Job schedule/tracker. As shown in Figure 4, there are a number of components (nodes) comprising several queues. Jersey REST web server manages web services instead of web, so as to provide web server maintenance service, especially composition, deployment, and management. Request traverses via the new job submission node and is received by the job scheduler, represented by the components at the left bottom of Figure 4. Our system model is a sort of open queueing network that has external arrivals and departures. The requests enter the system at “New Job Submission” and exit at “OUT” of Hadoop cluster and dedicated node for REST web server, respectively. The number of requests in the system varies with time. In analyzing an open system, we assume that the throughput is known (to be equal to the arrival rate) and we also assume that

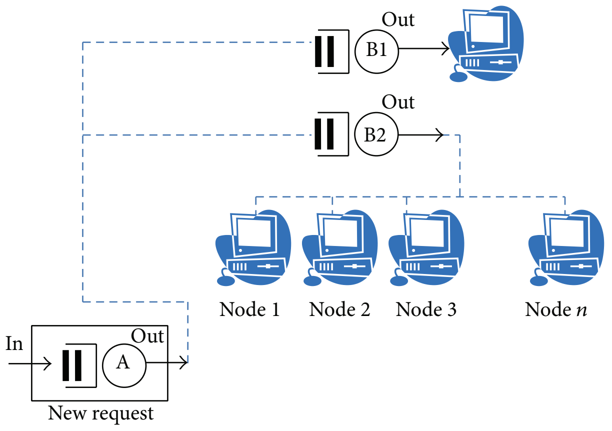

Figure 4: Evaluation model.

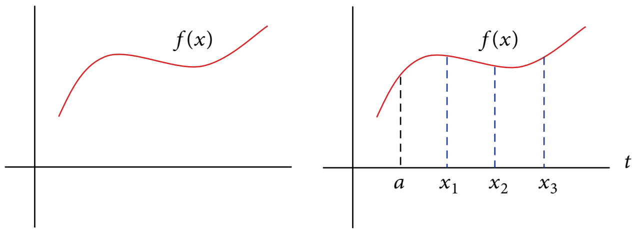

Figure 5: Task parallelization and job distribution among Hadoop clusters.

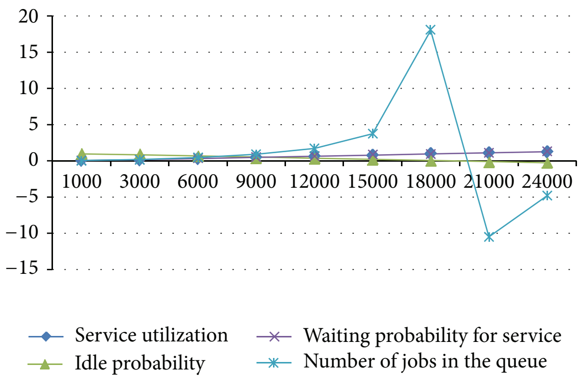

Figure 6: Experimental results by increasing service rates 1.

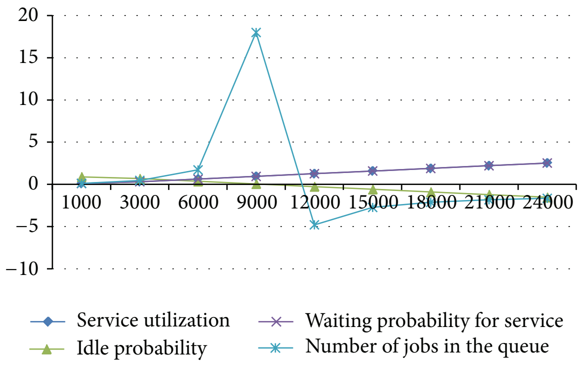

Figure 7: Experimental results by increasing service rates 2.

there is no probability of incomplete transfer in this system, so there is no retrial path to go back to Hadoop clusters. The initialization process for the request is done at the scheduler. Then, the job proceeds to the component, either "Hadoop cluster" network or Jersey REST web server, depending on the type of request; if the request is for the REST web server, it goes to the Hadoop cluster. If the request is for just web server, it goes to the web server.

A request may receive service at one or more queues before exiting the system. Jobs departing from the job scheduler arrive at either the Hadoop cluster or dedicated node for Jersey REST web service. All jobs submitted must first pass through the job scheduler/tracker for determining whether it is REST Open API request or MapReduce service. Requests arrive at the web server at an average rate of 1,000/s–15,000/s. Traffic intensity is calculated by the arrival rate over the service rate that means how fast the incoming traffic is serviced on the server.

The key feature of our design is to separate the Jersey web server onto a dedicated node. This feature isolate the performance that is not bound to the MapReduce computation. Hadoop clusters consist of multiple computing nodes. In order to get benefit from such multiple nodes and to handle the heavy load of MapReduce, we need to transform the problem into parallelizable form. To this end, we had the task parallelization phase in Section 3.2. Unlike an infinite series representation, the integral form is fully parallelizable and it is easy to divide the problem into chunks/parts of work. As shown in Figure 5, the total workloads is divided into three chunks so that we can integrate the formulae at different nodes in parallel. Thus, we can easily distribute and map these tasks onto multiple clouding nodes. We can approximately get the π value by integrating this equation for the interval $-1/2$ to $1/2$.

Figures 6 and 7 show the service utilization, idle probability, waiting probability for service, and number of jobs in the queue depending on increasing service rate. Since the service rate of each Hadoop node in this experiment is 19000 request/sec, the mean number of requests in the queue reaches up to the maximum on the total arrival rate which is increasing between 18000 and 21000. Then, it sharply falls down to the bottom right after the total arrival rate of 21000.

Figure 8 shows the system utilization depending on the change of performance of REST web service. The graph from Va10 to Va300 shows the system utilization by increasing workload on the REST web server. As mentioned above, incoming jobs proceed to the component, either "Hadoop cluster" network or Jersey REST web server, depending on the type of the request; if the request is for the REST web server, it goes to the Hadoop cluster.

If the request is for just web server, it goes to the web server. Thus, if there are large requests incoming for REST web service, then it is natural and there are relatively small requests for MapReduce. This is the reason why the utilization of MapReduce servers gets lower by increasing the server utilization of REST web server.

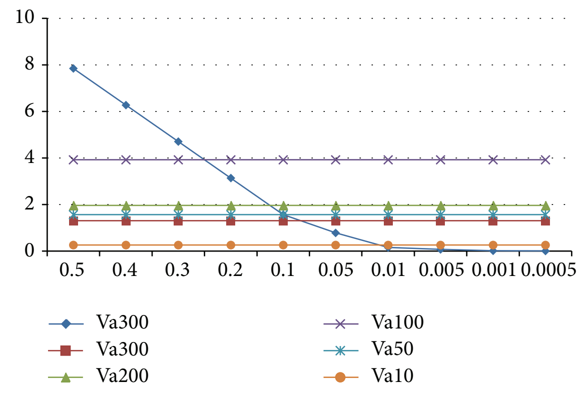

FIGURE 8: System utilization depending on the REST web server performance.

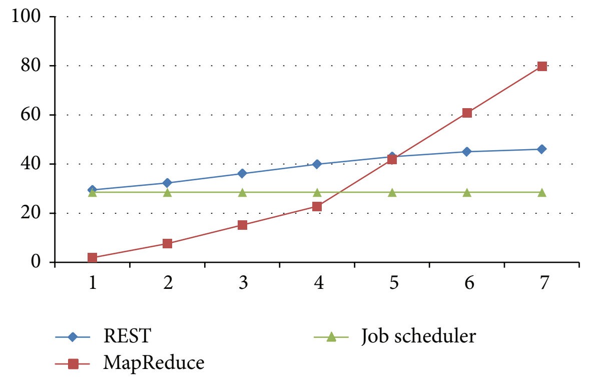

FIGURE 9: Utilization of REST web server, MapReduce clusters, and Job Scheduler.

Figure 9 shows the utilization of REST web server, MapReduce clusters, and Job scheduler. First, the utilization of job scheduler/tracker is constant because the performance change of the job scheduler/tracker is not very high. It is negligible. So, we did not care about the utilization of job scheduler. But we focus onto the utilization of REST web server and MapReduce clusters. By increasing workloads and the number of nodes, the system utilization of MapReduce clusters improves a lot. But, REST web server utilization is just a little bit increased by up to its internal processing limit.

6. Conclusion

In this work, we proposed a novel concept of REST-MapReduce, enabling users to use only the REST interface without using the MapReduce architecture. We make both MapReduce and REST Open API into an integrated service as REST-MapReduce. It is well known that there is slower response time for accessing simple RDBMS on Hadoop than direct access to RDMBS. The slow response time stems from the fact that MapReduce was originally designed for analyzing large data, not for simple RDBMS lookup. Such job scheduling, initiating, starting, tracking, and management during MapReduce execution are not necessary tasks for REST Open API service execution. To avoid such a problem, REST-MapReduce framework provides an integrated interface with high performance that supports both REST Open API and MapReduce. At the same time, the REST Open API service is provided by a separated architecture. Likewise, we can overcome such a slow response time of simple RDBM invocation on Hadoop by this integrated interface, but differentiated service. Surely, we have only focused on the task parallelism such as pi value calculation. But, generally we need to prepare various types of requests for simple DB lookup. So, future work of this research involves trying to make faster DB lookup request on Hadoop framework physically.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work is supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012-0008105).

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplied data processing on large clusters," in *USENIX International Conference on OSDI*, 2004.

[2] A. C. Murthy, C. Douglas, M. Konar et al., "Architecture of next generation apache hadoop MapReduce framework," Tech. Rep., 2013.

[3] "Processing and Loading Data from Amazon S3 to the Vertica Analytic Database," White Paper, Amazon Web Service, 2013.

[4] Hadoop, http://hadoop.apache.org/.

[5] *Amazon Elastic MapReduce Developer Guide*, Amazon Web Service, 2009.

[6] *Getting Started with Amazon Elastic MapReduce*, Amazon Web Service, March 2009.

[7] I. Macdonald, "Ruby/Amazon & Amazon web services," *Dr. Dobb's Journal*, vol. 30, no. 2, pp. 30–34, 2005.

[8] R. Hussain and H. Oh, "Cooperation-aware VANET clouds: providing secure cloud services to vehicular ad hoc networks," *Journal of Information Processing Systems*, vol. 10, no. 1, pp. 103–118, 2014.

[9] S. Islam, R. Rahman, A. Roy, I. Islam, and M. R. Amin, "Performance evaluation of finite queue switching under two-dimensional M/G/1(m) traffic," *Journal of Information Processing Systems*, vol. 7, no. 4, pp. 679–690, 2011.

[10] R. Pan, G. Xu, B. Fu, P. Dolog, Z. Wang, and M. Leginus, "Improving recommendations by the clustering of tag neighbours," *Journal of Convergence*, vol. 3, no. 1, pp. 13–20, 2012.

[11] H. Zhao and P. Doshi, "Towards automated RESTful Web service composition," in *Proceedings of the IEEE International Conference on Web Services (ICWS '09)*, pp. 189–196, July 2009.

[12] X. Zhao, E. Liu, G. J. Clapworthy, N. Ye, and Y. Lu, "RESTful web service composition: extracting a process model from linear

logic theorem proving," in *Proceedings of the 7th International Conference on Next Generation Web Services Practices (NWeSP '11)*, pp. 398–403, October 2011.

[13] Z. Li and L. O'Brien, "Towards effort estimation for web service compositions using classification matrix," 2010.

[14] C. Pautasso, O. Zimmermann, and F. Leymann, "RESTful web services vs. "Big" web services: making the right architectural decision," in *Proceedings of the 17th International Conference on World Wide Web (WWW '08)*, pp. 805–814, April 2008.

[15] R. Alarcon, E. Wilde, and J. Bellido, "Hypermedia-driven RESTful service composition," *Service-Oriented Computing*, Springer, vol. 6568, pp. 111–120, 2011.

[16] M. Yoon, Y. K. Kim, and J. W. Jang, "An energy-efficient routing protocol using message success rate in wireless sensor networks," *Journal of Convergence*, vol. 4, no. 1, 2013.

[17] "The Internet of Things: In action, The Next Web,"http://thenextweb.com/insider/2013/05/19/the-internet-of-things-in-action/.

[18] J. Rao and X. Su, "A survey of automated Web service composition methods," in *Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC '04)*, pp. 43–54, July 2004.

[19] J. Dean and S. Ghemawa, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation*, 2004.

[20] C. Pautasso, "RESTful web service composition with BPEL for REST," *Data and Knowledge Engineering*, vol. 68, no. 9, pp. 851–866, 2009.

[21] F. O. Catak and M. E. Balaban, "CloudSVM: training an SVM classifier in cloud computing systems," in *Pervasive Computing and the Networked World*, pp. 57–68, 2013.

[22] IBM Smart Cloud, http://www.ibm.com/cloud-computing/us/en/.

[23] M. de Kruijf and K. Sankaralingam, *MapReduce for the Cell B.E. Architecture*, Vertical Research Group. Department of Computer Sciences, University of Wisconsin-Madison, 2010.

[24] M. Choi, J. Park, and Y.-S. Jeong, "Mobile cloud computing framework for a pervasive and ubiquitous environment," *The Journal of Supercomputing*, vol. 64, no. 2, pp. 331–356, 2013.