Process Streaming Healthcare Data with Adaptive MapReduce Framework

Fan Zhang, Junwei Cao, Samee U. Khan, Keqin Li and Kai Hwang

Abstract As one of the most widely used healthcare scientific applications, body area network with hundreds of interconnected sensors need to be used to monitor the health status of a physical body. It is very challenging to process, analyze and monitor the streaming data in real time. Therefore, an efficient cloud platform with very elastic scaling capacity is needed to support such kind of real-time streaming data applications. The state-of-art cloud platform either lacks of such capability to process highly concurrent streaming data, or scales in regards to coarse-grained compute nodes. In this chapter, we propose a task-level adaptive MapReduce framework. This framework extends the generic MapReduce architecture by designing each Map and Reduce task as a scalable daemon process. The beauty of this new framework is the scaling capability being designed at the Map and Reduce task level, rather than being scaled at the compute-node level, as traditional MapReduce does. This design is capable of not only scaling up and down in real time, but also leading to effective use of compute resources in cloud data center. As

F. Zhang (✉)
IBM Massachusetts Laboratory, Littleton, MA 01460, USA
e-mail: fzhang@us.ibm.com

J. Cao
Research Institute of Information Technology, Tsinghua University, Beijing 100084, China
e-mail: jcao@tsinghua.edu.cn

S.U. Khan
Department of Electrical and Computer Engineering, North Dakota State University, Fargo, ND 58108-6050, USA
e-mail: samee.khan@ndsu.edu

K. Li
Department of Computer Science, State University of New York, New Paltz, New York 12561, USA
e-mail: lik@newpaltz.edu

K. Hwang
Department of Electrical Engineering and Computer Science, University of Southern California, Los Angeles, CA 90089, USA
e-mail: kaihwang@usc.edu

S.U. Khan et al. (eds.), *Handbook of Large-Scale Distributed Computing in Smart Healthcare*, Scalable Computing and Communications,
DOI 10.1007/978-3-319-58280-1_3

a first step towards implementing this framework in real cloud, we have developed a simulator that captures workload strength, and provisions the just-in-need amount of Map and Reduce tasks in realtime. To further enhance the framework, we applied two streaming data workload prediction methods, smoothing and Kalman filter, to estimate the workload characteristics. We see 63.1% performance improvement by using the Kalman filter method to predict the workload. We also use real streaming data workload trace to test the framework. Experimental results show that this framework schedules the Map and Reduce tasks very efficiently, as the streaming data changes its arrival rate.

Keywords Adaptive Mapreduce · Big data · Healthcare scientific applications · Kalman filter · Parallel processing

1 Introduction

Big-data technology has been a driving force for the state-of-art healthcare science. Most of the healthcare applications are composed of processes that need to manage Gigabytes of real-time and streaming data. For example, the Body Area Network [1] that is widely recognized as a medium to access, monitor, and evaluate the real-time health status of a person, has long been notorious for its computing intensiveness to process Gigabytes of data [2, 3] in real-time. Such data are collected from well-configured sensors to sample the real-time signals of body temperature, blood pressure, respiratory and heart rate, chest sound, and cardiovascular status, to name a few among others.

To process stream big-data in real-time, traditional parallelized processing frameworks, such as Hadoop MapReduce [4], Pregel [5], and Graphlab [6, 7], are structurally constrained and functionally limited. The major difficulty lies in their designs, which are primarily contrived to access and process the static input data. No built-in iterative module can be used when the input data arrives in a stream flow. Moreover, the existing frameworks are unable to handle the scenarios when the streaming input datasets are from various sources and have different arrival rates. Streaming data sample rates are consistently changed while the healthcare scientific applications are running. For example, the data collected when a person is sleeping is usually far less than the data collected when the person is running or swimming.

Cloud computing, with most of its few open-source tools and programming models, have provided a great opportunity to process such time-varied streaming data healthcare applications. Amazon Elastic MapReduce (EMR) framework [8], as an example, is typically represented by its compute instances being automatically scaled up or down and scaled in or out when workload changes. However, the

granularity of the scaling is too coarse for most of the healthcare applications, meaning we need more fined-grained scaling objects, such as CPU core, memory size or active processing tasks, to be used in order to be able to scale in real time. In our early research paper [9], we have discovered and identified an issue when scaling large number of compute instance, named the large-scale limitation issue. This issue demonstrates the most MapReduce applications fail to promise its scaling capability when the number of the compute instances exceeds the actual need. Therefore, the scalability would find itself more tractable when one the MapReduce application scales at a task level—increasing or reducing the Map and Reduce task number when a variation of the workload is predicted.

Tools with such fined-grained scaling capability are really rare to find. For example, the number of the Map tasks and Reduce tasks in a launched MapReduce job is fixed and can never be changed over time. However, such a fined-grained processing is widely needed. It is also very possible to do so since the number of the Map tasks are usually related the number of chunks of the input dataset sizes, and the number of the Reduce tasks is related to the hashing algorithm used for the intermediate keys. There is no strong constraint between the number of the Map Tasks and Reduce Tasks. All these make the scaling of the Map and Reduce tasks possible. In the next section, we will survey a few commercial tools that have been widely used in business for similar purposes.

Towards that end, we propose a full-fledged MapReduce framework that is tailored for processing streaming data in healthcare scientific applications. The framework goes beyond the traditional Hadoop MapReduce design, while also providing a much more generic framework in order to cover a wider group of real-time applications. Traditional Hadoop MapReduce requires the Map and Reduce number to be fixed, while this new MapReduce framework doesn't require so. Furthermore, Each Map and/or Reduce task is mandatory to reside on its own JVM in traditional Hadoop MapReduce. In the new framework, one such a Map and/or Reduce task can be specified much differently, whether it be in a JVM, a local thread or process, or even an entire compute node, to name a few among others. In other words, this framework absorbs the MapReduce design primitive, that the Map tasks outputting data to all the Reduce tasks, but in a way that requires much less constraint. It's our expectation that this new framework is supposed to streamline the streaming data processing, which has never been implemented in traditional Hadoop MapReduce at all. The major contribution of this chapter is summarized below.

(1) A unique task-level and adaptive MapReduce framework is proposed in order to process rich and varied arrival rate streaming data in healthcare applications. This framework enriches the traditional Hadoop MapReduce design in a way that specifically addresses the varied arrival rate of streaming data chunks. The framework is mathematically grounded on quite a few theorems and corollaries, in order to demonstrate its scaling capability. A full stack simulator is also developed with extensive experiments to validate the effectiveness.

(2) In order to better process streaming data and better use compute resources in a scaling process, a workload arrival rate prediction mechanism is therefore needed. This chapter covers two innovative workload prediction algorithms. Real-life healthcare experiments are used to compare the performance of them.
(3) Finally, we demonstrate more experiment results by showing the close relationship between the active numbers of the Map/Reduce tasks with the number of the streamed workload tasks. This reveals the adaptively of framework, as well as the correctness of the mathematical foundation.

This chapter is organized as follows. The first section introduces a real model of healthcare scientific applications, and necessitates the requirement of processing stream-style big-data. In the second section, we report the related work, and also introduce our unique approach. A real-life healthcare application study is followed in section three. After that, we reveal the methodology of the proposed task-level adaptive MapReduce framework. Two innovative methods for predicting the streaming data workload are proposed in the next section. Experiment settings and the results are introduced in the fifth section. Finally, we conclude the work and summarize a few aspects of directions to continue the work.

2 Related Work and Our Unique Approach

There is an escalating interest on leveraging the state-of-art big-data platform to process stream data in real-time. In this section, we investigate previous publications in this area. Thereafter, we briefly describe our unique approach to show the advantage among other solutions.

2.1 Related Work

Health Information System [10] was originated and further extended from the hospital information system [11] that addresses what is called the health informatics issues. The major challenge involves the shift from paper-based to computer-based, and further to the Internet-based data storage processing. Patients, healthcare consumers, and professionals are more involved into a collaboration phase from a traditional in- or out-patient medication, to a widely acceptable online on-demand treatment. Such a shift requires a significantly powerful interconnection compute network and highly scalable compute nodes for both computing and big-data processing.

MapReduce is a simple programming model for developing distributed data intensive application in cloud platforms. Ever since Google initially proposed it on a cluster of commodity machines, there have been many follow-up projects. For instance, Hadoop [12] is a Mapreduce framework developed by Apache, and Phoenix [13] is another framework designed for shared memory architecture by Stanford University. Pregel [5] is a message-based programming model to work on real-life applications that can be distributed as an interdependent graph. It uses vertex, messages, and multiple iterations to provide a completely new programming mechanism. GraphLab [6, 7] is proposed to deal with scalable algorithms in data mining and machine learning that run on multicore clusters.

The above-mentioned tools have a wide impact on the big-data community and have been extensively used in real-life applications. Along those lines, other research efforts addressing streaming data have been proposed. Nova [14], due to its support for stateful incremental processing leveraging Pig Latin [15], deals with continuous arrival of streaming data. Incoop [16] is proposed as an incremental computation to improve the performance of the MR framework. Simple Scalable Streaming System (S4) [17], introduced by Yahoo!, is universally used, distributed, and scalable streaming data processing system. As one of its major competitor, Twitter is using Storm [18] that has also gained momentum in real-time data analytics, online machine learning, distributed remote procedure call, ETL (Extract, Transform and Load) processing, etc. Other companies, such as Facebook, Linkedin and Cloudrera, are also developing tools for real-time data processing, such as Scribe [19], Kafka [20], and Flume [21]. Even though the programming languages are different, they all provide highly efficient and scalable structure to collect and analyze real-time log files. Complex Event Processing systems (CEP) are also gaining interest recently. Popular CEP systems include StreamBase [22], HStreaming [23], and Esper&NEsper [24]. Essentially the CEP systems are primary used in processing inter-arrival messages and events.

Different from these research and commercial products, our work goes beyond a programming model framework, but also serves as a simulator to help users identify how their compute resources can be effectively used. Secondly, the framework is still based on a generic MapReduce, but not entirely a Hadoop MapReduce framework. We do not intend to design a completely new framework, but we aim to extend a widely acceptable model to allow it to seamlessly process streaming data. Our work may aid the programmers to manipulate the streaming data applications to process such kinds of flow data in a more scalable fashion.

2.2 Our Unique Approach

In a nutshell, our approach implements each Map and Reduce task as a running daemon. Instead of processing local data in Hadoop Distributed File System

(HDFS) as what traditional Hadoop usually does, the new Map tasks repeatedly pull the cached stream data in the HDFS, generate the Map-stage intermediate key-value pairs and push them to the corresponding Reduce tasks. These Reduce tasks, quite similar to what the Map tasks, are also implemented in such a way. These Reduce tasks repeatedly pull the corresponding data partitions from the entire Map task output, generate the Reduce-stage intermediate key-value pairs and push them to the local disk cache. In this way, each Reduce task has to cache the intermediate status of all the output key-value pairs when the application is on going.

Take a simple example to illustrate the scenario. Multiple users are collaborating by adding/removing/updating words, sentences and files to HDFS as data streams. An enhanced WordCount application requires obtain the real-time count of each word when the texts are consistently updated. Map tasks are implemented in a stateless manner, meaning that they just simply process the corresponding input data and produce output without having to worry about previous data that they have processed. However, the Reduce tasks must be implemented in a stateful way. This means that each of the Reduce task has to save the real-time count of each word and adaptively add or reduce the count whenever there is a change in the HDFS.

The essence of our approach, as we can see from the analysis, lies in the seamless connection to the MapReduce implementation. Users write data streaming applications as they did in writing traditional MapReduce applications. The only difference, however, is that they need to write such a Map and Reduce daemon function. Secondly, our approach can be implemented to scale the Map and Reduce task number separately. Traditional MapReduce framework which scales compute nodes usually leads to low compute resource usage when the active running tasks cannot utilize these compute nodes effectively.

An example is used here to illustrate the adaptive MapReduce application that calculates the real-time occurrence of each word in a set of documents. Multiple people consistently update these documents, and therefore the statistics of each word count differ from time to time.

The Map tasks below are continuously fed by input data stream, and enter into a loop that would not stop until the data stream update ends. For each of the loop, all the words are extracted and emitted key value pairs as tradition WordCount does. The Reduce tasks, also being launched in a loop, are fed continuously by the intermediate data produced by all of the Map tasks. The only difference here is the result, which needs to be fetched from HDFS. Because for each result that has been calculated, it needs repeatedly updating. Therefore, Reduce tasks should be able to not only write data back to the HDFS, but also retrieve data back from HDFS for updating.

```
Map Function: map(String k, String v):
// k: doc name in a streaming data
// v: doc contents
While(HasMoreData)
    value = GetStreamDataContent();
    for each word w in value:
        EmitIntermediate(w, "1");

Reduce Function: reduce(String k, Iterator vs):
// k: a word
// vs: a list of counts
While(HasMoreIntermediateData)
    int result = getResultFromHDFS;
    for each v in vs:
        result += ParseInt(v);
        Emit(AsString(result));
```

3 Problem Formulation of a Real-Life Healthcare Application

In Fig. 1, we illustrate a case study of the body area network as a real-life healthcare application. Health status regarding the respiration, breath, cardiovascular, insulin, blood, glucose and body temperature, which are consistently collected by sensors deployed all over the human body. This is wearable computing, which sends data periodically to a mobile phone via local network. The sample frequency is determined by the capacity of the sensors as well as the processing rate of the mobile device.

Because most of the mobile devices nowadays are equipped with advanced processing unit and large memory space, data can be continuously transferred to a mobile device and even processed within the mobile device. Therefore, the various sources of input data can be even locally analyzed before moving to the remote data center. The data center has information on various disease symptoms and the corresponding value threshold in regards to the insulin pump and glucose level, etc. The purpose of the follow-up data transferring is to compare the collected data with those in the database, and quickly alert the user the potential symptom he/she is supposed to see, and provide a smart medical suggestion in real-time.

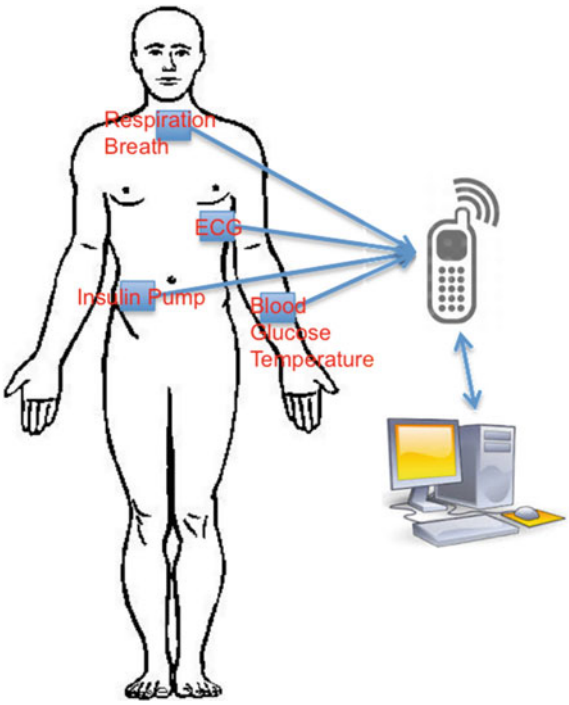


Fig. 1 As a case study of the body area network, data streams collected from various sensors are pushed to a mobile device and backend data center for real-time medical treatment

Data sampled within a wearable computing can usually go up to GB per minutes. With a high sample rate, more accurate data can therefore be used, and the diagnosis can be more in real time, and the alert can be better in use. Therefore, it is strongly needed that the data center can support thousands of users' real-time data access, as well as computing across multiple dimensions of syndromes and features to be collected.

4 Task-Level Adaptive MapReduce Framework

In this section, we brief an overview of the Hadoop MapReduce framework as a start. Thereafter, the task-level provisioning framework is introduced in the subsequent text.

4.1 Preliminary of Hadoop MapReduce Framework

The standard Hadoop MapReduce framework is depicted in Fig. 2. There are *4* parallel Map tasks and *3* parallel Reduce tasks, respectively. Because the total

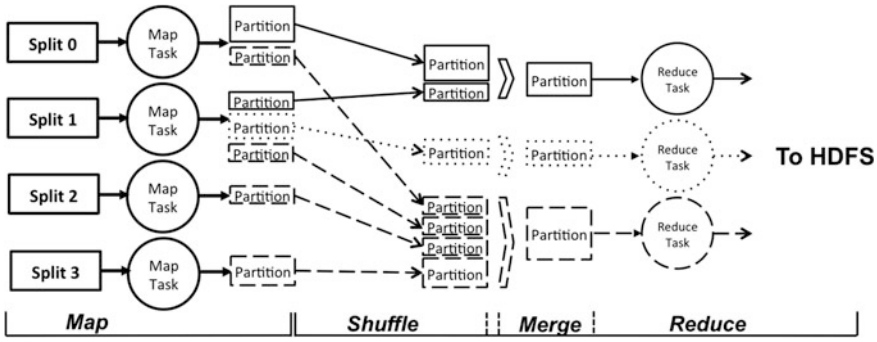


Fig. 2 A MapReduce framework splits the input file into 4 segments, and each segment corresponds to one Map task. Map Tasks output data partitions, which are further shuffled to the corresponding Reduce tasks. There are *3* reduce tasks which generate *3* separate outputs

number of the Map tasks normally equals to the number of the input data splits, there are four data splits as well. Each Map task performs a user-defined Map function on the input data that resides in the HDFS and generates the intermediate key-value pair data. These intermediate data are organized on the *partition* basis. Each of the partition consists of certain key-value data pairs, whose keys can be classified into one group. The simplest classification method is a hash function. Within such a hash capability, data partition belonging to the same group are shuffled across all the compute nodes and merged together locally. There are three data partitions shown in the figure. These merged data partitions, as indicated by three different framed rectangular boxes, are consumed by three Reduce tasks separately. The output data generated by all the Reduce tasks are written back to the HDFS.

Each Map task resides in a Map slot of a compute node. Usually Two Map slots reside in one compute node. The slot number per node can be adjusted in the configuration file. The total number of the Map slots determines the degree of parallelism that indicates the total number of Map tasks that can be concurrently launched. The rational for the Reduce task and Reduce slot is the same. The whole Hadoop MapReduce workflow is controlled in a JobTracker located in the main computer node, or what is called the NameNode. The Map and Reduce tasks are launched at the TaskNodes, with each task corresponding to one TaskTracker to communicate with the JobTracker. The communication includes heart-beat message to report the progress and status of the current task. If detecting a task failure or task straggler, the JobTracker will reschedule the TaskTracker on another Task slot.

As we can see from Fig. 2 above, the Hadoop MapReduce is essentially a scheduling framework that processes data that can be sliced into different splits. Each Map task is isolated to process its input data split and no inter-Map communication is needed. The Hadoop MapReduce framework can only be applied to process input data that already exist. However, real-life big-data applications typically require the input data be provisioned in streaming and be processed in

real-time. Therefore, an enhanced MapReduce framework is required to cater for such a need. That is the motivation behind out design of the task-level adaptive MapReduce framework.

4.2 Task-Level Adaptive MapReduce Framework

An adaptive MapReduce framework is proposed to process the streaming data in real-time. One of the most significant challenges here is how to process the streaming data with varied arrival rate. Real-life applications entail workloads of a variety of many patterns. Some of the workloads show a typical pattern of periodical and unpredictable spikes, while others are more stable and predictable. There are four technical issues that we should consider when designing the adaptive framework.

First, the framework should be both horizontally and vertically scalable to process a mixture of such varied workloads. In other words, the scheduling system should either be able to manage compute node count, but also types. For some Hadoop MapReduce applications, merely managing the number of compute nodes is not necessarily sufficient. Certain kinds of workloads require large CPU-core instances while others need large-memory instances. In a nutshell, scaling in a heterogonous system is one of the primary principles.

Second, the number of the active Map and Reduce tasks should be in accordance with the cluster size. Even though the Map and Reduce count determines the overall performance of the whole system, it still doesn't perform that desirable if the cluster is either over provisioned or under provisioned.

Third, scaling the Reduce tasks is very tedious. This is determined by the design of the Reduce phase. If the number of the Reduce task increases, the hash function that maps a particular Map output partition data to a Reduce function will change. Take modular operation as a hash function as an example. Increasing the Reduce count from *r* to *r'* leads to *key mod r* to *key mod r'* as the corresponding new hash function. A reorganization of the Reduce output will be added to the Reduce phase when the number of the Reduce task has changed.

Fourth, we also need to consider the heterogeneity of the processing capabilities of different Map tasks. Some of the Map tasks may be scheduled on a slow node while others are on much faster nodes. An appropriate load balancing mechanism can further improve the rescheduling philosophy implemented in the traditional Hadoop MapReduce. The purpose is to coordinate the progress of the entire task without leading to skew task execution time.

Fifth, the optimal runtime of Map and Reduce task count should be specified. Traditionally, the initial Map task number depends on the input dataset size and the HDFS block size. The Reduce task count is determined by the hash function. The new framework requires a redesign of the Map and Reduce Task scheduling policy by considering the input data arrival rate instead of their sizes instead.

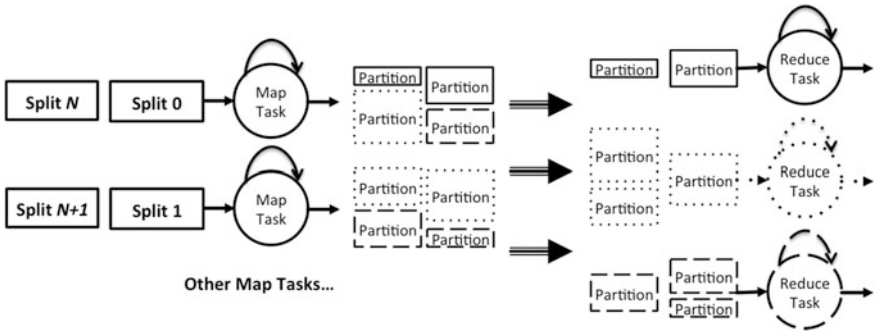


Fig. 3 A demonstration of task-level adaptive MapReduce framework which processes streaming data. Each Map and Reduce task has a non-stop running daemon function which continuously processes the input data

To satisfy all the purposes above, we demonstrate a task-level adaptive MapReduce framework in Fig. 3. Two Map tasks are used as representatives of the Map stage. Each Map task, different from the Map task of the traditional Hadoop MapReduce, defines a loop function as shown in the loop-back arrow. The two Map tasks are launched as special runtime daemons to repeatedly process the streaming data. Each of the Map task produces two continual batches of output data partitions. Similarly, the Reduce tasks are also scheduled in such a loop-like daemon that continuously processes their corresponding intermediate data produced by all the Map tasks.

In this new task-level adaptive MapReduce framework, the JobTrackers need to be redesigned to maintain a pool of TaskTrackers, and the TaskTracker count may change as the workload changes.

There are two ways to feed data streams into the Map tasks. A proactive strategy caches streaming data locally first and pushes them every fixing period of time, say 1 min. As an alternative option, data splits can also be pushed in a reactive way. In other words, a cache size is defined in HDFS before the input data starts to move in, whenever the cache usage hits a ratio, say 85%, the data splits begin to be pushed to the Map tasks.

4.3 Adaptive Input Data Split Feeding

The adaptive MapReduce framework starts from a novel runtime scheduler that feeds different Map tasks with different number of data splits. In Fig. 4a, we show a study case of the adaptive input data split feeding. As a start, six splits of input data arrive. The scheduler, without knowing the processing capability of each Map task, distributes the data splits evenly to the two Map tasks, which results to each Map task having three data splits. Suppose the first Map task is executed on a faster

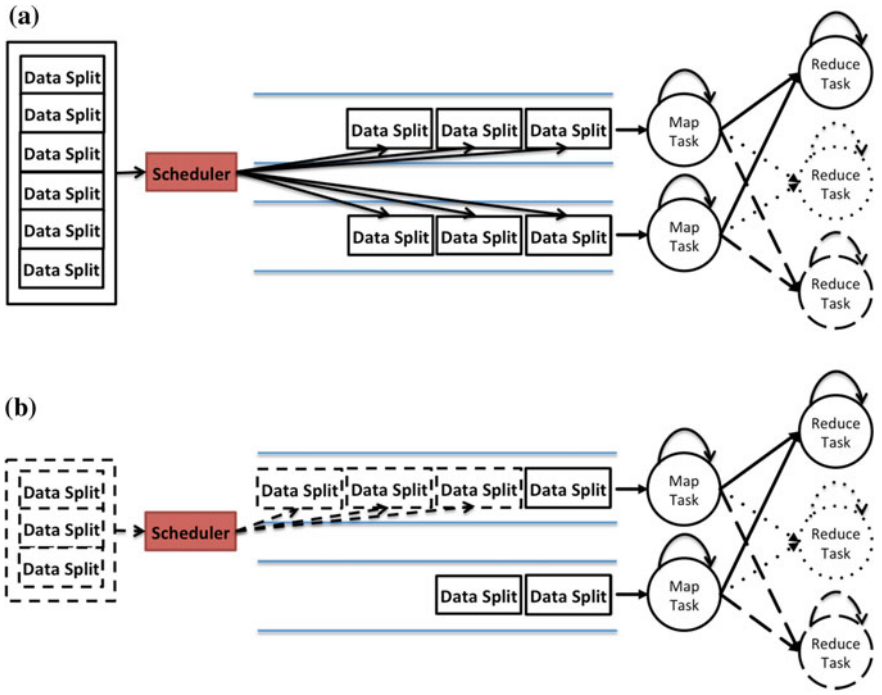


Fig. 4 A demonstration of the adaptive input data split feeding. **a** Initially six data splits arrive. Without knowing the compute capacity of each Map task, scheduler divides the workload evenly between the two queues, each one having three data splits. **b** After being aware of the processing capacity of each Map task, the scheduler sends three data splits to Map task one which shows twice the processing capacity at the consecutive scheduling period

compute node and has processed two splits of the input data while the second Map task has processed only one. Being aware of such a skewed processing capability, the scheduler sends the newly arrival three data splits adaptively to balance the workload in Fig. 4b. This leads to Map task one has four data splits while Map task two has two, and the total execution time of the Map stage is minimized.

In this case, processing the three newly arrival data splits doesn't result in an increase of Map task count, but trigger the scheduler to dispatch them fairly to all the Map tasks. Scheduler caches the input data locally in HDFS and regularly sends them to different Map tasks. The time interval is also adaptively determined by the arrival rate of the data splits.

To refresh readers' memory and ease difficulty in understanding all the mathematics below, we plot table one below, which summarizes all the symbols and explain their meanings (Table 1).

Suppose there are m Map tasks and the task queue length of each Map task be Q. In the Fig. 4, we set m equal to 2 while Q equal to 4. Suppose as a start the input data has n_0 data splits. As long as n_0 be less than $m * Q$, each Map task gets approximately n_0/m data splits (Fig. 5).

Table 1 Symbols, notations and abbreviations with brief introduction

Notation	Brief definitions with representative units or probabilities
m	The total number of available Map tasks
Q	The total number of data splits that can be accommodated in each Map task
n_0	The total number of data splits arrives at the start time
t	The scheduling period, denoting the data feeding frequency from the scheduler to all Map tasks
dMapTaskN(j)	The number of data splits that remained in the queue of Map task j at time t_i
dMapTaskN'(j)	The number of data splits that remained in the queue of Map task j at time t_{i+1}
eMapTaskC(j)	The estimated data processing capacity for Map task j
addedDataSplit (j)	The number of data splits that needs to be added to Map task j after new stream data arrives
TT	Estimated finish time of all the Map tasks
α	Upper bound percentage threshold used when Map task number above $\alpha * Q$ in a queue, Map tasks are over provisioned
β	Lower bound percentage threshold used when Map task number below $\beta * Q$ in a queue, Map tasks are under provisioned

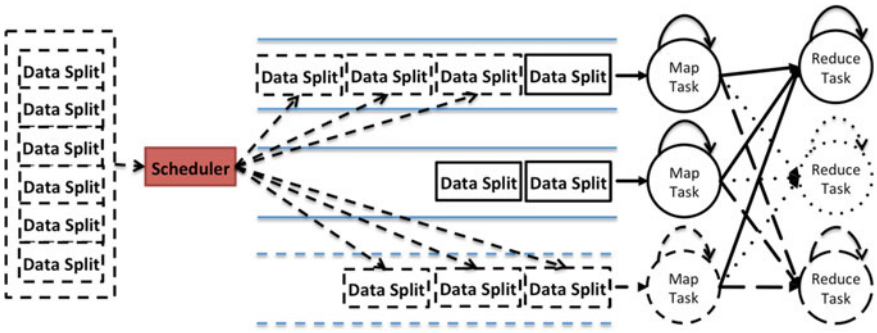


Fig. 5 Demonstrates of adding an adaptive Map task. Continued from the previous example, if the input data split count is six, the scheduler adaptively launches one Map task instead of feeding all the data splits to the queues. The other three data splits are moved to the newly added Map task

The scheduling period, namely the time interval between two data feeding periods is t. In other words, every t units (seconds or minutes) of time, scheduler feeds one batch of the cached data into the Map queues. Suppose at time t_i, the data splits count of each Map task queue equals to [*dMapTaskN(0), dMapTaskN(1), …, dMapTaskN(m − 1)*] after the newly arrived data splits have been pushed into the queues. After time t at t_{i+1}, the remaining task count becomes [*dMapTaskN'(0), dMapTaskN'(1), …, dMapTaskN'(m − 1)*]. The estimated processing capacity of each Map task is estimated as [(*dMapTaskN'(0) − dMapTaskN(0)*)/*t*, (*dMapTaskN'(1) – dMapTaskN(1)*)/*t*,…, (*dMapTaskN'(m − 1) – dMapTaskN(m − 1)*)/*t*].

Suppose n_{i+1} data splits arrive at time t_{i+1}. We consider a scheduling algorithm that effectively distributes all these data splits to all the Map tasks in Theorem 1. Second, we consider in Corollary 1 that whether the Map task number should remain the same or need to change. Third, if the Map task number needs to change, we calculate the variation of the Map tasks in Theorem 2. We separate the analysis into two different sections. In this section we discuss a scenario that workload doesn't have to trigger the change of the Map tasks. In the following section, we continue to discuss scenarios that Map task number needs to change.

Theorem 1 ***Condition***: *Suppose there are m_i Map tasks being actively used at time $t + 1$. As a new stage, N_{i+1} new data splits arrive. For any Map task j, dMapTaskN (j) data splits are in its queue. Its estimated data processing capacity is eMapTaskC (j).*

Conclusion*: The new data split count to be added to its queue is represented by:*

$$\begin{aligned} addedDataSplit(j) = {} & eMapTaskC(j) \\ & * (N_{i+1} + SUM(dMapTaskN(:))) \\ & / SUM(eMapTaskC(:)) \\ & - dMapTaskN(j) \end{aligned} \tag{1}$$

SUM(dMapTaskN(:)) denotes the total number of data splits across all the queues. SUM(eMapTaskC(:)) denotes the aggregated processing capacity of all the Map tasks.

Proof The scheduling target is to make sure all the tasks of the Map queues be finished almost at the same time, and let that task time be an unknown value *TT*. For any Map task *j*, $TT = (dMapTaskN(j) + addedDataSplit(j))/eMapTaskC(j)$, $j \in [0, m_i - 1]$. Note that $\Sigma\ dMapTaskN(j) = N_{i+1}$. Solving a total of $m_i - 1$ equations leads to the proof of theorem 1. **Q.E.D**

Corollary 1 *Let Q be the queue length of each Map task, namely the total number of data splits that can be accommodated in one Map task queue. Other conditions are the same as in the Theorem 1. Then new Map tasks need to be added if $\exists\ j \in [0, m_i - 1]$, $dMapTaskN(j)) + addedDataSplit(j) > \alpha * Q$. Similarly, Map task number needs to be reduced if $\forall\ j \in [0, m_i - 1]$, $dMapTaskN(:) + addedDataSplit (j) < \beta * Q$. Symbol $\alpha \in [0, 1]$ is a preset threshold to determine how full the Map task queues are allowed. Similarly, $\beta \in [0, 1]$ is preset to determine how empty the Map task queues are allowed.*

Proof If $\exists\ j \in [0, m_i-1]$, $dMapTaskN(j)) + addedDataSplit(j) > \alpha * Q$, this means the Map task number of one Map task queue will be above threshold if the new data splits were added. It automatically triggers a Map task increase request to the scheduler. Similarly, if $\forall\ j \in [0, m_i-1]$, $dMapTaskN(j) + addedDataSplit (j) < \beta * Q$ holds, this indicates the Map task count of each Map queue is less than a preset value, which means sufficient resources have been provided. A request is therefore sent out to reduce the Map task count. **Q.E.D**

In a nutshell, the purpose of designing such an adaptive scheduler is to leverage the processing capability of all the Map tasks and balance the start time of all the Reduce tasks.

4.4 Adaptive Map Task Provisioning

In the previous section, we focus on discussing the Map task provisioning mechanism that determines the time Map task needs to be updated. A natural extension along that line requires answer a provisioning mechanism—how many Map tasks need to be added or reduced in order to efficiently process the new stream data splits. If adding Map task is required, how to distribute the stream data splits across all the Map tasks, including the new ones. On the contrary, if reducing Map task is required, how to distribute the stream data splits, as well as the data splits in the queues that are supposed to remove, to all the remaining Map task queues.

Theorem 2 *Given the condition in Theorem 1 and* $\exists\, j \in [0, m_i - 1]$, *dMapTaskN (j)) + addedDataSplit(j)* $> \alpha * Q$, *the number of the new Map tasks that is needed is given below:*

$$\lfloor (N_{i+1} - aQ\ SUM(eMapTaskC(:)) / eMapTaskC(j^*) + SUM(dMapTaskN(j))) * eMapTaskC(j^*) / aQ \rfloor + 1 \tag{2}$$

For the Map task j, the new data splits count added to its queue equals to the formula below when $j \in [0, m_i - 1]$.

$$aQ\ eMapTaskC(j) / eMapTaskC(j^*) - dMapTaskN(j)) \tag{3}$$

Suppose the default estimated computing capacity of each new Map task is eMapTaskC. For the new added Map task, each is allocated an initial number of data splits in their queues. The data split number is given below:

$$aQ\ eMapTaskC / eMapTaskC(j^*) \tag{4}$$

Proof Suppose Map task j^* has the maximum computing capacity across all the Map tasks: $eMapTaskC(j^*) > eMapTaskC(j)$ for all $j \in [0, m_i - 1]$. Then the maximum data splits count allowed to be added to its queue equals to $\alpha Q - dMapTaskN(j^*))$. Proportionally compared, the maximum data split count of the jth Map task queue equals to $\alpha Q\ eMapTaskC(j)/eMapTaskC(j^*) - dMapTaskN(j))$ and Eq. (3) is proven. Therefore, aggregating all the data splits that are allocated to Map task j equals to $\alpha Q\ SUM(eMapTaskC(:))/eMapTaskC(j^*) - SUM(dMapTaskN(:))$. Since we assume that all the Map tasks can be finished within $\alpha Q/eMapTaskC(j^*)$, then given the default processing capacity of all the new Map tasks for the

remaining data splits, the needed Map tasks count is calculated by dividing the remaining data split count over the expected Map task finish time and Eq. (2) is therefore proven. Equation (4) is calculated by multiplying the predicted Map task execution time with the default processing capacity of each Map task. **Q.E.D**

Theorem 3 *Given the condition in Theorem 1 and* $\forall\, j \in [0, m_i - 1]$, *dMapTaskN (j) + addedDataSplit(j)* $< \beta * Q$ *and Suppose dMapTaskN(0) > dMapTaskN (1) > … > dMapTaskN*$(m_i - 1)$, *the Map Task set {MapTask_0, MapTask_1, …, MapTask_k} needs to be removed if:* $\forall\, j \in [k, m_i - 1]$, *dMapTaskN (j) + addedDataSplit(j, k)* $< \beta * Q$ *and* $\exists\, j \in [k + 1, m_i - 1]$, *dMapTaskN (j)) + addedDataSplit(j, k + 1)* $> \alpha * Q$. *After removing the Map tasks, the remaining Map task j (*$j \in [k + 1, m_i - 1]$*) adds data split count: addedDataSplit (j, k + 1) A more general term is defined as follows.*

$$\begin{aligned} addedDataSplit(j, p) = {} & eMapTaskC(j) \\ & * (N_{i+1} + SUM(dMapTaskN(:))) \\ & / SUM(eMapTaskC(p{:}\, m_i - 1)) \\ & - dMapTaskN(j) \end{aligned} \tag{5}$$

Proof A descending order of the remaining data splits leads to removing the Map task starting from the slowest one. The slower one Map task is, the slower that Map queue tasks to finish. We start to remove *MapTask_0* and add its queued data splits to N_{i+1}. Reallocating the total N_{i+1} + *dMapTaskN(0)* data splits to the remaining $m_i - 1$ Map tasks. If the data split count of each these remaining Map task still lower than $\beta * Q$, the procedure moves on. This procedure stops until when at least there is one Map task has its queued data split count larger than $\alpha * Q$. **Q.E.D**

4.5 Adaptive Reduce Task Provisioning

Adaptive provisioning of the Reduce tasks is far less straightforward than provisioning the Map tasks. Since Hadoop MapReduce is a framework primarily designed to scale the Map stage by involving embarrassingly parallel Map tasks, the Reduce tasks require network resource and an *m* to *r* data shuffling stage. In Fig. 6, we identify a scaling scenario of adding a new Reduce task to the original three tasks. The new added Reduce task should have no impact on saving the network usage since all the intermediate data still have to be moved among all the compute nodes. The only difference is the degree of parallelism in the Reduce stage, that each Reduce task can process less data partitions as well as move less output data back to HDFS.

As aforementioned, the Map tasks can be added incrementally one by one. However, this doesn't necessarily guarantee the best scheduling performance if

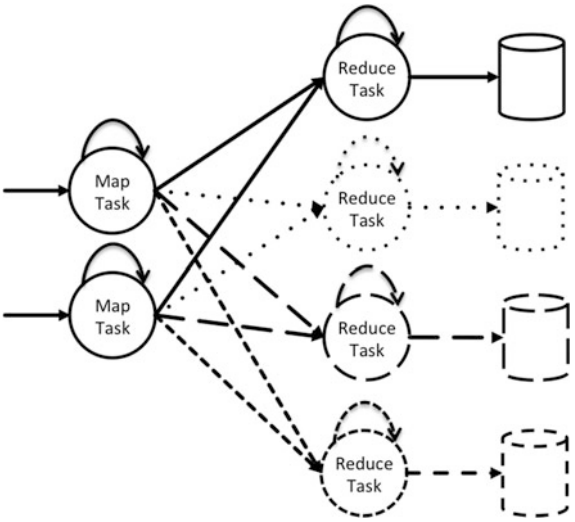


Fig. 6 A graphical illustration shows one parallel Reduce task being added. This added Reduce task brings no benefit in the data shuffling stage but results to a reduced data volume to be processed/outputted for each Reduce task

Reduce tasks were to be added in the same way. This is because there is no strict demand that one input data split should go to a particular Map task. The Reduce tasks, however, only accepts their partition data in need. Adding one Reduce task would inevitable change the hash function, which accordingly leads to the data partition changed, and mess the shuffling process.

Take an example here. Suppose the key set of the dataset is an nine-number set [0, 1, 2, …, 8]. There are three Reduce tasks R1, R2 and R3 as shown in Fig. 6. The hash function is a simple modular operation, e.g. key mod 3. Therefore, R1 gets partition data whose keys are [0, 3, 6]; R2 gets partition data whose keys are [1, 4, 7]; R3 gets partition data whose keys are [2, 5, 8]. Adding one Reduce task leads to the partition be [0, 4, 8] for R1, [1, 5] for R2, [2, 6] for R3 and [3, 7] for R4. In all, there are six keys that are either moved to R4 or being exchanged among inside R1, R2, and R3. Similar conclusion applies to the case that five Reduce tasks are used. However, if the Reduce task number doubled to 6, then [0, 6] will be for R1; [1, 7], [2, 8], [3], [4], [5] are keys for R2 to R6 respectively. Then there are data associated with only three keys, [3, 4] and [5], that needs to moved.

In such a case, a workaround would be replacing the hash function with an enumerated list of the keys as a lookup table. For each intermediate key-value pair needs to be shuffled, the corresponding Reduce task number is searched through the list. For example, the list can be like this [R1, 0, 1, 2], [R2, 3, 4, 5], [R3, 6, 7, 8]. If a new Reduce task R4 is added, we can simply create a new entry as [R4, 2, 8], and remove the keys [2] and [8] from their corresponding list.

The downside of the workaround approach can be easily identified. The search operation might involve I/O data accessing, which is far less efficient than calculating the hash function. We can put the mapping list in memory instead if the total number of the keys is not very large.

5 Experimental Studies

In this section, we first propose two methods for stream data workload prediction. After that, we show our experimental results of the prediction performance of the methods and the makespan of using these methods. Last, we report our task-level adaptive experimental results in terms of the Map and Reduce count in runtime when workload changes.

5.1 Workload Prediction Methods

For stream data applications, adaptive MapReduce task provisioning strategy should align with the workload variation. However, workloads are normally unknown in advance. In this section, we investigate two widely used prediction methods first and compare them prediction performance using real workload in the next section.

Stochastic control, or learning-based control method, is a dynamic control strategy to predict workload characteristics. There are numerous filters that can be applied. For example, smooth filter, or what we normally call as smoothing technique, predicts real-time workload by averaging the workload of a previous time span. The basic assumption here is that workload behaves reactively and not subject to significant variation in a short period of time. The average of the past one period would best represents the future workload.

There are many further improvements on the smoothing technique. For example, weighted smoothing gives higher weights to more recent workload than those that are old. The assumption here is that more recent workload would show higher impact on the real-time workload than older ones. Other prediction methods include AR method, which applies polynomial functions to approximate the workload. Among others, we want to bring forward the Kalman filter [25], also named as linear quadratic estimation, which is also widely used in workload prediction. Kalman filter works on a series of historical data stream of noise, updates and predicts future trend with statistically optimal estimations.

5.2 Experimental Settings

We use SimEvent [26] to simulate the experiments. This is a toolkit component included in Matlab. Each map/reduce task is simulated as a queue. During the runtime, the Map and Reduce tasks serve workload at different capacity, therefore the proposed task-level scheduling framework fits into such a need.

Both the Kalman Filter and the Smooth Filter are used to predict the workload. Two metrics, workload prediction accuracy and makespan, are both used for the three methods. The workloads we use were primary produced from real body area network data trace [27, 28]. The workload fluctuation amplitude, on the other hand, applies the web data trace from the 1998 Soccer World Cup site [29]. This workload trace has the average arrival rate data on each single minute over a 60-min duration as shown in Fig. 7a, d and g. We carefully choose three typical and varied stream data workload types: small, intermediate and strong, for the purpose of simulation. Small workload typically generates 20–60 data splits per minute. Intermediate workload generates 30–150 data splits per minute while strong workload generates 160–1180 data splits per minute.

5.3 *Experimental Results*

In Fig. 7b, e and h, the Kalman filter shows no more than to 19.97% prediction error compared to 50% of that when using the Smooth filter method in the light workload

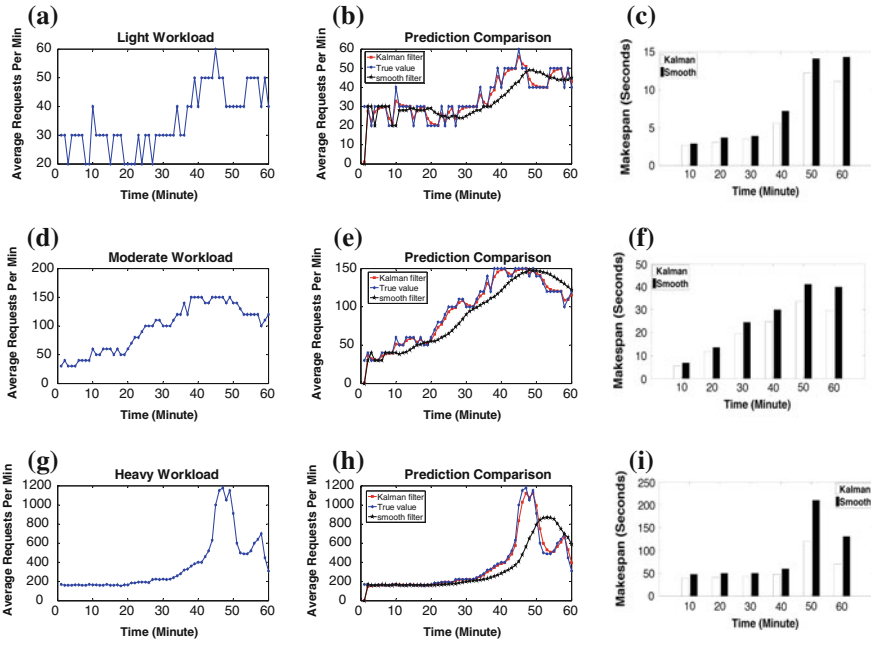
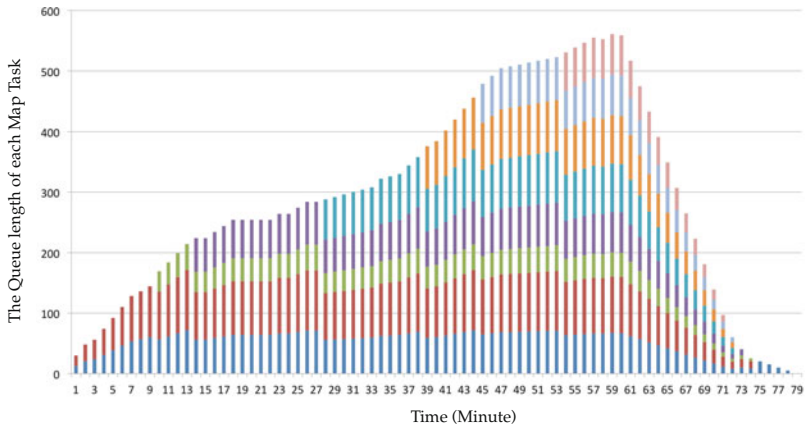
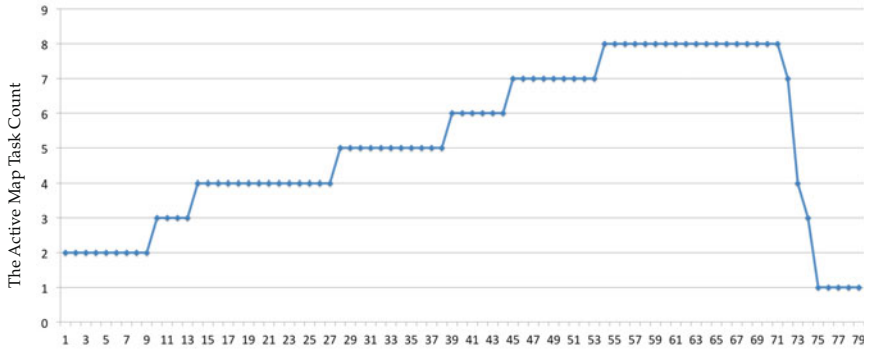


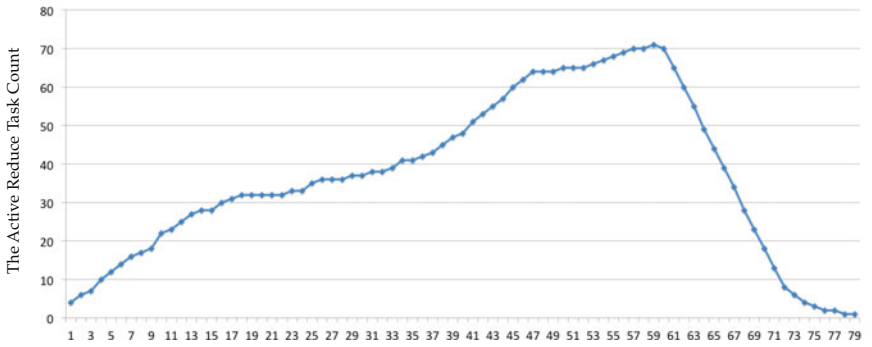
Fig. 7 Comparison under three types of workloads. Figure **a**, **d** and **g** are light, moderate and heavy workload respectively; Figure **b**, **e** and **h** demonstrate the workload prediction accuracy of the two method: Smooth filter and Kalman filter; Figure **c**, **f** and **i** report the makespan of using the two prediction methods. Kalman filter based workload prediction performs better than the Smooth filter based prediction method



(a) Data split count for each Map task queue in each minute



(b) Map task count in each minute



(c) Reduce task count in each minute

Fig. 8 Demonstration of the Map task queue, Map task number and Reduce task number in each minute of the light workload case. The Map and Reduce count adaptively follow the workload trend firmly

case. Under the intermediate workload, the prediction errors of these two filters are constrained to 14.1% and 35% respectively. For the strong workload scenario, these values turned to 27.2% and 90.3% for the two methods. Comparing the prediction error of the two methods across all the workload cases, the maximum margin is the strong workload case, which is typically 63.1% less by using the Kalman filter method.

From Figure 7c, f and i, we can see that under the three types of workloads, the Kalman based workload prediction based method outperforms the smooth filter based method over up to 28, 34 and 85%. All these results indicate that a good prediction method only gives a satisfactory estimation of the workload trend, but also improves the scheduling performance.

In Fig. 8, we demonstrate the scheduling effect of the proposed framework. Figure 8a illustrates the data split count of each Map task queue every minute. Figure 8b demonstrates the Map task count that are actively running. Theorems 2 and 3 calculate these Map task count. The count of each Reduced task every minute is reported in Fig. 8c. The Reduce task counts are calculated by the total data partition for all the Reduce tasks over the processing capacity.

We can see that as the workload increases, the Map task count also increases accordingly. As a result, each Map task has more data splits waiting in its queue, and so do the Reduce tasks. In Fig. 8a, the rising trend becomes less significant when hitting the 61th minute since no more follow-up stream data splits arrived. However, it is not until the 71th minute when the Map task number starts to noticeably reduce as reported in Fig. 8b. The reason is that the data splits in each Map task is accumulating in the previous 61 min. Until the 71th minute the data splits of each Map task queue are sufficiently short and the Theorem 3 starts to reduce the total Map tasks.

6 Conclusions and Future Work

We proposed a task-level adaptive MapReduce framework in this chapter. We conclude three major aspects of contribution, and then illustrate the future work that should extend the work.

6.1 Conclusions

A significant amount of scientific applications require effective processing of streaming data. However, there's a gap between the state-of-art big data processing frameworks Hadoop MapReduce for such a need. Since Hadoop MapReduce has gained its dominance in big-data processing domain for years, even though we have seen many existing streaming data processing toolkits, such as Storm, Spark Streaming, we still believe that, it would benefit the whole MapReduce community

if the framework could be adapted for the need of processing the streaming data, without having to move to a new framework. Therefore we propose such an adaptive Hadoop MapReduce framework, which is built on top of MapReduce, but could also be easily implemented in a virtualized cloud platform. We conclude the contribution of this work in the following three aspects.

(1) **Proposed a task-level adaptive MapReduce framework**. Traditional Hadoop MapReduce fixes the number of the Map and Reduce tasks. In this new framework, we suggest a framework that removes such a constraint, which allows the Map and Reduce task number to be adaptive given the runtime workload. Users don't have to change their programming habit in traditional MapReduce, and this framework allows such a transition from processing fixed dataset to streaming data seamlessly.
(2) **Runtime Map and Reduce task estimation**. The workload, as well as the queuing length of each task determines the runtime Map/Reduce task count. We have created a full-fledged mathematical model to estimate the real time task number, in order to optimize the streaming data processing rate, as well as keeping the cost of using compute resource at an acceptable level.
(3) **Adaptive task simulator**: With a simulator being used not only as a workload prediction toolkit, but also mathematically calculates the active number of Map/Reduce tasks in real time as the workload changes. This simulator implements the mathematically model we propose in this chapter, and estimates the workload in a way we proposed in the subsequent sections.

6.2 Future Work

We suggest extending this work in the following two directions:

(1) **Coherent scaling of Map/Reduce tasks and compute resources.** Scaling Map or Reduce task only is mainly investigated in this chapter. However, the scaling needs to be multi-tier, meaning the number of compute nodes also needs to align with the existing number of Map or Reduce tasks. It would be significantly useful if the framework supports the coherent scaling of compute resources, as well as the Map and Reduce tasks, from both theoretical aspect and implementation.
(2) **Continue the framework in large-scale heterogeneous cloud systems.** In a large cloud platform, the framework can be way complicated than our experimental scale. Lots of runtime issues, such as resource contention, virtual resource scaling cost etc., would happen during the course of scaling. This will bring other concerning issues that go beyond the description of the mathematical framework we propose above.
(3) **Release the simulation toolkit**. The simulation toolkit should be packaged into a software library in Hadoop MapReduce online in order to make sure the

service be available for such adaptive MapReduce applications online. For larger-size virtualized cloud platform, it can be deployed online and expose its API for public use. Furthermore, we plan to implement the adaptive scaling framework in Spark, in order to see the effectiveness within in-memory computing.

Acknowledgements This work was supported in part by the National Nature Science Foundation of China under grant No. 61233016, by the Ministry of Science and Technology of China under National 973 Basic Research Grants No. 2011CB302505, No. 2013CB228206, Guangdong Innovation Team Grant 201001D0104726115 and National Science Foundation under grant CCF-1016966. The work was also partially supported by an IBM Fellowship for Fan Zhang, and by the Intellectual Ventures endowment to Tsinghua University.

References

1. S. Ullah, H. Higgins, B. Braem, et al. A Comprehensive Survey of Wireless Body Area Networks. Journal of Medical Systems 36(3)(2010) 1065–1094.
2. M. Chen, S. Gonzalez, A. Vasilakos, et al. Body Area Networks: A Survey. ACM/Springer Mobile Networks and Applications. 16(2)(2011) 171–193.
3. R. Schmidt, T. Norgall, J. Mörsdorf, et al. Body Area Network BAN–a key infrastructure element for patient-centered medical applications. Biomed Tech 47(1)(2002)365–8.
4. J. Dean and S. Ghemawat, Mapreduce: Simplified Data Processing On Large Clusters, in: Proc. of 19th ACM symp. on Operating Systems Principles, OSDI 2004, pp. 137–150.
5. G. Malewicz, M. H. Austern, A. J. C. Bik, et al. Pregel: A System for Large-Scale Graph Processing, in: Proc. of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD 2010, pp. 135–146.
6. Y. Low, J. Gonzalez, A. Kyrola, et al, GraphLab: A New Framework for Parallel Machine Learning, in: Proc. of the 26th Conference on Uncertainty in Artificial Intelligence, UAI 2010.
7. Y. Low, J. Gonzalez, A. Kyrola, et al, Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud, Journal Proceedings of the VLDB Endowment, 5(8) (2012), pp. 716–727.
8. http://aws.amazon.com/elasticmapreduce/.
9. F. Zhang, M. F. Sakr, Cluster-size Scaling and MapReduce Execution Times, in: Proc. of The International Conference on Cloud Computing and Science, CloudCom 2013.
10. R. Haux, Health information systems–past, present, future, International Journal of Medical Informatics, 75(3–4)(2006), pp. 268–281.
11. P. L. Reichertz, Hospital information systems—Past, present, future, International Journal of Medical Informatics, 75(3–4)(2006), pp. 282–299.
12. http://hadoop.apache.org/.
13. J. Talbot, R. M. Yoo and C. Kozyrakis, Phoenix++: modular MapReduce for shared-memory systems, in: Proc. of the second international workshop on MapReduce and its applications, MapReduce 2011, pp. 9–16.
14. O. Christopher, C. Greg and C. Laukik, et al, Nova: Continuous Pig/Hadoop Workfows, in: Proc. of the 2011 ACM SIGMOD international conference on Management of data, SIGMOD 2011, pp. 1081–1090.
15. C. Olston, B. Reed, U. Srivastava, et al, Pig latin: a not-so-foreign language for data processing, in: Proc. of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD 2008, pp. 1099–1110.

16. P. Bhatotia, A. Wieder and R. Rodrigues, et al, Incoop: MapReduce for incremental computations, in: Proc. of the 2nd ACM Symposium on Cloud Computing, SoCC 2011.
17. L. Neumeyer, B. Robbins and A. Nair, et al, S4: Distributed Stream Computing Platform, in: Proc. of the International Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms, KDCloud 10, pp. 170–177.
18. http://storm.incubator.apache.org/.
19. http://www.scribesoft.com/.
20. J. Kreps, N. Narkhede, J. Rao et al. Kafka: A Distributed Messaging System for Log Processing. in: Proc. of 6th International Workshop on Networking Meets Databases NetDB 2011.
21. http://flume.apache.org/index.html.
22. http://www.streambase.com/.
23. http://www.hstreaming.com/.
24. http://esper.codehaus.org/.
25. R. E. Kalman, A new approach to linear filtering and prediction problems, Journal of Basic Engineering 82(1)(1960), pp. 35–45.
26. http://www.mathworks.com/products/simevents/.
27. C. Otto, A. Milenković, C. Sanders and E. Jovanov, System architecture of a wireless body area sensor network for ubiquitous health monitoring, 1(4)(2005), pp. 307–326.
28. E. Jovanov, A. Milenkovic, C. Otto1 and P. C de Groen, A wireless body area network of intelligent motion sensors for computer assisted physical rehabilitation, Journal of NeuroEngineering and Rehabilitation, 2(6)(2005), pp. 1–10.
29. M. Arlitt, T. Jin, Workload characterization of the 1998 World Cup Web Site (Tech. Rep. No. HPL-1999-35R1). Palo Alto, CA: HP Labs.