A Step towards Hadoop Dynamic Scaling

Qiaobin Fu
Department of Computer Science
Boston University
Boston, USA
qiaobinf@bu.edu

Nicholas Timkovich
University of Chicago
Chicago, USA
npt@uchicago.edu

Pierre Riteau
University of Chicago
Chicago, USA
priteau@uchicago.edu

Kate Keahey
Argonne National Laboratory
Lemont, USA
keahey@mcs.anl.gov

***Abstract*—Many application portals successfully manage to scale elastically in order to provide a stable response time by integrating on-demand cloud resources. This is more challenging for applications that have to manage a dynamic configuration. Our paper investigates the question: under what circumstances (if any) dynamically adding more nodes to the Hadoop computation will result in performance improvement? On one hand, if we add more nodes to a Hadoop computation, the computation will potentially finish faster since more computational power will be brought to bear on the problem. On the other hand, ensuring that we can use those nodes effectively may require data redistribution, thus creating additional overhead which may obviate any performance advantages. In this paper, we identified the container allocation as a key factor that affects Hadoop performance. Moreover, to mitigate the overhead, we describe and evaluate three methods for data redistribution in this use case and discuss their advantages and disadvantages.**

***Keywords*-Hadoop, Dynamic scaling, Geospatial processing, Cloud computing**

I. Introduction

Many applications today are provided via the software-as-a-service (SaaS) paradigm where a user can request execution of an application through a portal, and the application provider takes responsibility for allocating resources, deploying the application, and returning results to the user. This has become popular in business as in science where the science gateway paradigm – as exemplified by the CyberGIS or NanoHUB platforms – is increasingly popular [15]. One challenge that arises in this context is how a portal can adapt to a varying number of users/requests with varying workloads, and provide a predictable response time for all requests. By providing on-demand resources, cloud computing offers one possible answer to this question: if we can elastically expand the platform backing up the portal and effectively integrate the additional resources into the computation, we can manipulate the response time on demand.

Although integrating resources dynamically into an ongoing computation has proven effective in the case of e.g., high throughput computing (HTC) workloads, it is much more challenging for applications that have to manage a dynamic configuration, such as data distribution, targeting a fixed number of nodes. In particular, in the case of dynamically scaling Hadoop applications the overhead of making the application aware of additional resources can incur more cost than it brings benefit if not done carefully.

Over the past decade, many excellent methods [2], [5]–[9], [11], [12], [14], [16], [18], [20], [22], [25] have been proposed to translate Hadoop application requirements into an optimal resource allocation and data distribution pattern. However, most of these efforts focus on optimal static resource allocation; only few approaches address dynamic environments. In particular, state-of-the-art method discussed in [7] can dynamically determine the resources required to successfully complete Hadoop jobs. However, dynamically reconfiguring the Hadoop cluster to reach that optimal node allocation for a specific operation involves a data redistribution overhead that can make the overall operation too costly; this aspect of dynamic resource adjustments has not been studied as extensively.

Our paper investigates the scalability pattern of Hadoop in such dynamic situations. In particular, we sought to answer the question: Under what circumstances (if any) dynamically adding more nodes to the Hadoop computation will result in performance improvement? If we add more nodes to a Hadoop computation, there is potential that the computation will finish faster since more computational power will be brought to bear on the problem. However, ensuring that we can use those nodes effectively may require data redistribution, thus creating additional overhead which may obviate any performance advantages.

Our contributions in this paper are threefold:

- We investigated the trade-off between dynamically adding more nodes to the Hadoop computation and its overall performance improvement.
- We identified the Hadoop container allocation as a key factor that affects performance.
- We evaluated three methods for data redistribution and analyzed their advantages and disadvantages.

The remainder of this paper shows the background on related techniques (section II), conducts extensive experiments on a geospaital application (section III), reviews the related work on cloud auto-scaling techniques (section IV), and then concludes (section V).

II. Background

A. Hadoop Overview

Apache Hadoop is an open-source project, aiming for parallel processing of large-scale data sets on distributed systems.

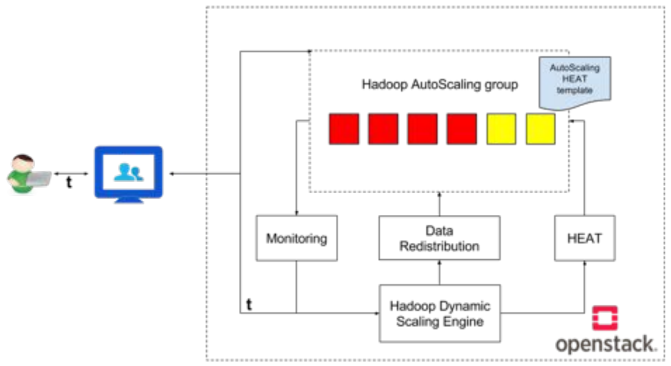
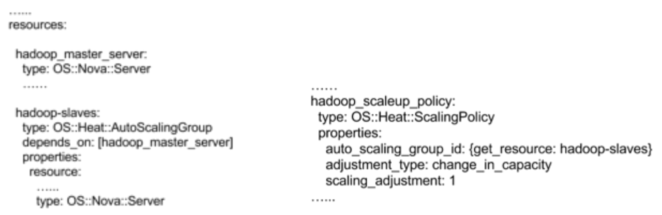


Fig. 1. Overview of the Hadoop dynamic scaling system



```
......
resources:

 hadoop_master_server:
  type: OS::Nova::Server
  ......

 hadoop-slaves:
  type: OS::Heat::AutoScalingGroup
  depends_on: [hadoop_master_server]
  properties:
   resource:
    ......
    type: OS::Nova::Server
```

```
......
hadoop_scaleup_policy:
 type: OS::Heat::ScalingPolicy
 properties:
  auto_scaling_group_id: {get_resource: hadoop-slaves}
  adjustment_type: change_in_capacity
  scaling_adjustment: 1
......
```

(a) Hadoop cluster nodes (b) Hadoop scaling policy

Fig. 2. HEAT template for launching stacks

It has three key building blocks: *Hadoop Distributed File System (HDFS)*, *Hadoop YARN*, and *Hadoop MapReduce* [23]. HDFS is a distributed file system that provides high-throughput access to application data. Hadoop YARN is a generalized container oriented framework for job scheduling and cluster resource management, which is adopted by the MapReduce version 2 [24]. MapReduce is a YARN-based system for parallel processing of large data sets. The MapReduce programming paradigm includes two stages: map stage and reduce stage. The map stage has 6 phases: compute, collect, sort, spill, combine, and merge spills; and the reduce stage has 3 phases: shuffle, sort, and reduce.

B. YARN Architecture

The Hadoop YARN splits up the functionalities of resource management and job scheduling/monitoring into separate daemons: (1) the *ResourceManager* with its core component – *Scheduler* is responsible for allocating resources to the various running applications in the cluster, according to the constraints like capacities, queues, etc.; (2) the *NodeManager* is the per-machine framework agent who is responsible for containers, monitoring their resource usage and reporting the same to the ResourceManager; (3) the per-application *ApplicationMaster* negotiates resources from the ResourceManager and works with the NodeManager to execute and monitor the tasks. Note that, the resource *container* is a general resource model for applications. An application (via the ApplicationMaster) can request resources with specific requirements such as: memory, CPU, etc.

C. Implementation

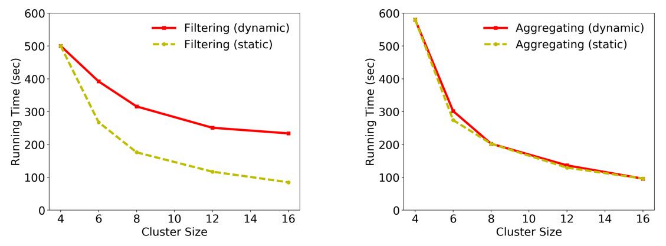
Figure 1 gives an overview of the Hadoop dynamic scaling system: at its heart is a scaling engine that adds nodes to the Hadoop computation as needed to achieve service level objectives. In our implementation nodes are added using OpenStack [27], in particular the Heat and Nova components. First, we launch a new stack on Chameleon testbed using *HEAT* template as shown in Figure 2. Due to the master slave architecture of Hadoop framework, we define three important resources: (1) a Hadoop master server with resource type of *OS::Nova::Server*; (2) Hadoop slaves with resource type of *OS::Heat::AutoScalingGroup*, which depends on the Hadoop master server; (3) Hadoop scaling up policies with resource type of *OS::Heat::ScalingPolicy*. For both (1) and (2), we specify the Hadoop installation scripts in the user data template. Notice that, we cannot simply define the scaling down policies as the way we define scaling up policies, since it's possible that HEAT shuts down all the nodes that contain the same copy of a data chunk, and lose that data chunk forever in Hadoop setting. One needs to ensure there is no data loss when removing Hadoop nodes, etc. Considering these factors, we decide to remove slave nodes on the fly by using the *HDFS exclude* file as specified in the *dfs.hosts.exclude* attribute at *hdfs-site.xml* file. Then, we can add nodes that need to be decommissioned to that file, and call HDFS commands to decommission the nodes. Finally, we can safely delete the node instances via *nova* in OpenStack.

After configuring the Hadoop cluster, a user can specify the the desired response time (t) of the Hadoop application. The *Hadoop dynamic scaling engine* takes t as an input parameter, and then derives the desired number of containers allocated to the application based on the existing work [7]. The *monitoring module* monitors the running state of the Hadoop cluster, and reports the state to the scaling engine every second. For now, we identified the number of containers as one key factor that influences the application performance. So, we simply monitor *runningContainers* for the running application via the ResourceManager REST API. However the monitoring module is flexible and scalable and it can easily incorporate various other metrics (e.g., cluster metrics, application request attempts, or system metrics) as needed. Based on the monitoring data, the scaling engine calculates the number of nodes that need to be added to or removed from the Hadoop cluster, and then it issues a Heat command to execute the corresponding policy defined in the HEAT template.

Finally, the *Data Redistribution* module monitors the change of the cluster size. Once it detects some change, e.g., more new nodes are added to the cluster, it issues data redistribution operation on the Hadoop cluster, so that the application can better utilize the resources available on the cluster. This module takes one of the three data redistribution methods discussed in section III-C.

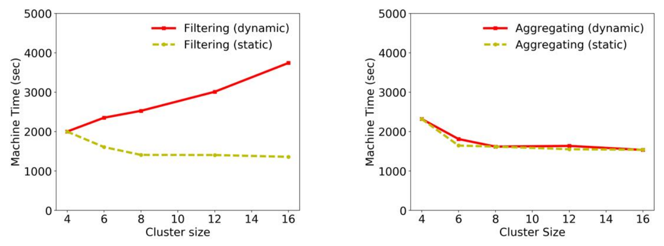
D. UrbanFlow

We conducted experiments in the context of a geospatial application, UrbanFlow [21], to understand its scalability patterns. UrbanFlow integrates geolocated Twitter data with



(a) Filtering performance (b) Aggregating performance

Fig. 3. UrbanFlow performance (static vs dynamic)



(a) Filtering economic cost (b) Aggregating economic cost

Fig. 4. UrbanFlow economic cost (static vs dynamic)

detailed landuse map (parcel level) to detect and analyze individual human mobility patterns. It runs common geospatial analysis operations like point in/nearest polygon.

In terms of implementation, UrbanFlow includes two main MapReduce jobs in its pipeline: (1) a filtering job, that filters tweets based on text, spatial and temporal constraints, and (2) an aggregating job that performs integration of Twitter data and the secondary dataset. The filtering job parallelizes the processing of the dataset via multiple mapper tasks. Each mapper task will process a chunk of data stored on the local or remote machine.

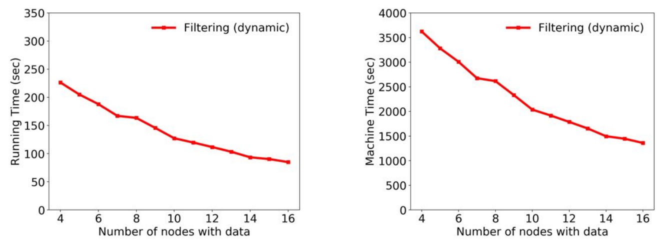
This split – into map and reduce operations – typifies the general class of MapReduce applications. In this paper, we use UrbanFlow as an exemplar of such an application; our results generalize to a larger class of applications corresponding the the map-reduce pattern. Thus, our references to the filtering component are synonymous with the map component of the application and our references to aggregating component are synonymous to reduce.

III. Experiments

For our experiments we used the Chameleon testbed [4], a large-scale configurable experimental environment. Each node in our experiments is configured with 24 cores, 128 GB RAM, 230 GB disk space, and 10 Gbps network, CentOS 7 and Hadoop version 2.7.1. Unless specified otherwise, we use the default Hadoop configuration, such as block size as 64 MB. In the experiments, we investigate two metrics: UrbanFlow performance in terms of running time, and UrbanFlow economic cost defined as the resources used multiplied by the time we used them for. We compare the above metrics in two modes: (1) in static mode, the application runs on a Hadoop cluster with optimal data distribution for the specific number of nodes; (2) in dynamic mode, the application starts out with data distributed optimally for a small cluster and subsequently nodes are dynamically added during the Hadoop cycle without distributing the data. We measure runtime as well as economic cost (a.k.a., Machine Time) defined as *number of nodes used* $\times$ *runtime*.

A. Establishing a Baseline

First, we conduct experiments on a static cluster to establish a baseline. The dashed yellow lines in Figure 3 show the running time of the filtering job and aggregating job respectively under various cluster sizes in the static mode, up to 16 nodes.



(a) Filtering performance (b) Filtering economic cost

Fig. 5. UrbanFlow performance and economic cost on 16-node cluster

As the size of the static cluster increases, both UrbanFlow jobs (filtering and aggregating) finish faster. As expected, the application scales well, i.e., the running time is almost halved as we double the cluster size each time.

Then, we show what happens in response to dynamically adding new nodes to a 4 node Hadoop cluster by the red lines in Figure 3. In the dynamic mode, we add 2, 4, 8, 12 nodes in turn to the cluster. As we can see, for the filtering application (map – which relies on data distribution on the nodes) the running time in the dynamic mode will also decrease with more nodes, but it takes longer time than a static configuration on the same number of nodes. By comparing the results, we conclude that the performance degradation is caused by factors inherent to adding nodes dynamically and hypothesize that the main factor has to do with data distribution. The aggregating application (reduce) is unaffected by data distribution.

Looking at the economic cost (static versus dynamic) in Figure 4, we see that for the filtering method in the case of the dynamic scenario the cost is going up (despite the runtime going down) making the investment in additional nodes unviable from a cost perspective. This does not compare well with the static scenario where additional nodes result in cost going down. As before, the aggregating application is unaffected by data distribution.

To verify that data distribution on Hadoop clusters does affect UrbanFlow's performance as well as its economic cost in finer granularity, we conduct more experiments on 16-node clusters. Firstly, we start with an $n-$node ($n \in [4, 16]$) cluster, that is all the dataset is distributed among the n nodes. Then, we dynamically add $16 - n$ new nodes to the $n-$node Hadoop cluster. Figure 5 presents UrbanFlow filtering performance and its economic cost on the 16-node cluster. As can be seen, as we increase the number of nodes that have dataset,

TABLE I
SYSTEM METRICS (DYNAMIC VS STATIC)

Metrics	CPU (usr:%)	Disk (writ:B)	Load (avg)	Net (recv:B)	Net (send:B)	IO (writ)
Dynamic	1.52	808005	0.62	1852728	1853332	8.81
Static	2.42	1287723	0.93	884529	884540	13.54
Ratio	0.63	0.63	0.67	2.09	2.10	0.65

both UrbanFlow filtering's running time and economic cost decreases. The questions we will explore now are: (1) why data distribution affects UrbanFlow performance; and (2) how to best neutralize the cost of data distribution in this dynamic scenario both in terms of runtime and economic cost.

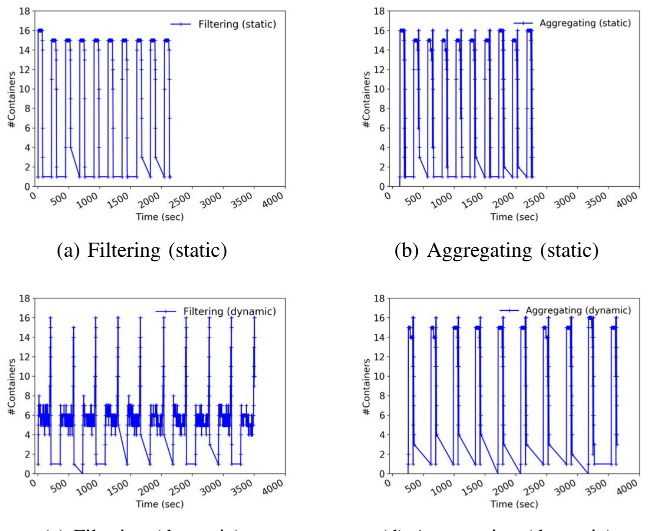
B. Profiling UrbanFlow

To support the hypothesis that we proposed in the previous section, in the following experiments, we launch a cluster with 16 nodes in both static and dynamic modes (we omit the results for a cluster with 8 nodes, since the results are similar). In dynamic mode, we distribute all of the dataset among 4 nodes. In both modes, each node is configured with 8GB memory for YARN, and each Mapper and Reducer container is configured with 8GB memory and 1 CPU, separately. Therefore, the maximum number of available containers in the Hadoop cluster is 16. Sometimes, the number can be below 16 due to YARN scheduler's behavior. For instance, the scheduler may reserve some containers for future jobs. In both modes, we run UrbanFlow for 10 iterations.

To profile UrbanFlow, we use Linux *dstat* tool to generate system resource statistics, which provide us with some general and instantaneous information on CPU, IO, and network utilization, interrupts, system loads, context switches, and other system statistics [26]. For Hadoop metrics, we use YARN's ResourceManager REST API [1] to collect cluster and application metrics, including the number of applications pending, the number of containers reserved, the number of containers currently running for an application, etc. We collect these metrics every second. The profiling processes are lightweight. In addition, we collected the UrbanFlow job counters over the 10 iterations, like file system counters, MapReduce framework, etc. Note that, we profiled 31 system metrics using the *dstat* tool, 18 Hadoop metrics using YARN's ResourceManager REST API, and 31 counters for Filtering job, 51 counters for Aggregating job. After analyzing these metrics in both modes, we only show the ones that differ from each other significantly in following sections.

1) Profiling system resource metrics: Table I shows the average values of the system metrics in the 16-node Hadoop cluster in both modes, as well as the ratio of the system metrics in dynamic mode to the ones in static mode. Looking at the ratios for CPU utilization for userspace processing (CPU usr),

[1]The ResourceManager REST API's allow the user to get information about the cluster - status on the cluster, metrics on the cluster, scheduler information, information about nodes in the cluster, and information about applications on the cluster.



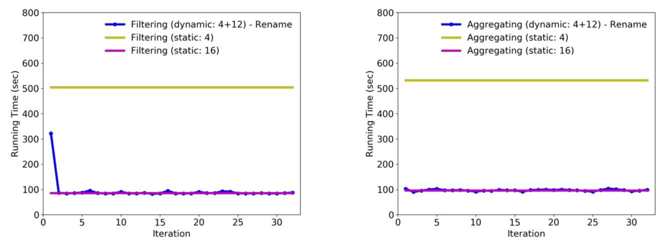
(a) Filtering (static) (b) Aggregating (static)

(c) Filtering (dynamic) (d) Aggregating (dynamic)

Fig. 6. UrbanFlow running containers

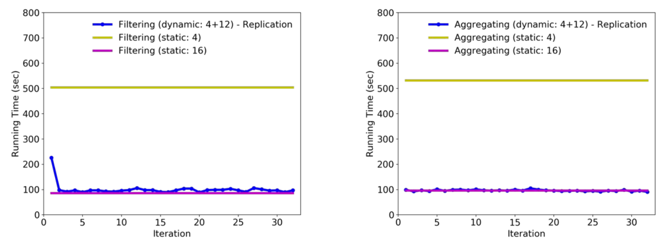
Disk total write bytes (writ), system average load, and the average number of I/O write requests completed (IO writ), we know that *the Hadoop cluster in dynamic mode has less utilization (around* 0.6*) than the static mode*. Notice that, table I shows the metrics treating Filtering jobs and Aggregating jobs as a whole, so the Filtering jobs should have much lower cluster utilization than 0.6 of the static cluster utilization. However, the network usage, in terms of receiving (i.e., Net recv) and sending (i.e., Net send) bytes, is higher in dynamic mode. Since in the dynamic mode, only 4 nodes hold all the dataset, we also observed that the average network sending speed of the 4 nodes is 48.37Mbps, which is $7.17\times$ the one in the static mode. Moreover, by checking all nodes' instantaneous network throughput at any time point, we found the maximum network throughput in the dynamic mode is 3.53Gbps, which is below the configured network speed limit (i.e., 10Gbps). Therefore, the network delay for transferring a data block from one machine to another is around $64 * 8/(1024 * 10) = 0.05$ seconds. Since UrbanFlow needs to launch 256 map tasks to complete the Filtering job, in the worst case (i.e., every block is transferred through the network), the total network delay should be bounded by $0.05 * 256 = 12.8$ seconds. However, in reality, the total delay could be much smaller than it due to the reason that the YARN scheduler prefers to schedule resources taking advantages of data locality, and the number of blocks that need to be transferred should be smaller than 256. Therefore, *the network utilization could be a factor that affects Filtering job's performance, but not a dominant one.*

2) Profiling Hadoop metrics: For metrics obtained from Hadoop cluster via REST API, we only show the results for the number of running containers of UrbanFlow. Figure 6 shows the results over 10 iterations in both static mode and dynamic mode. The experiments start at time 0. As can be seen, Figure 6 (c) is not like the other three. More specifically, *both jobs in the static mode and the Aggregating job in the dynamic mode can always fully utilize the available containers (i.e.,*



(a) Filtering performance (b) Aggregating performance

Fig. 7. UrbanFlow performance (Copy&Rename)



(a) Filtering performance (b) Aggregating performance

Fig. 8. UrbanFlow performance (Replication Factor)

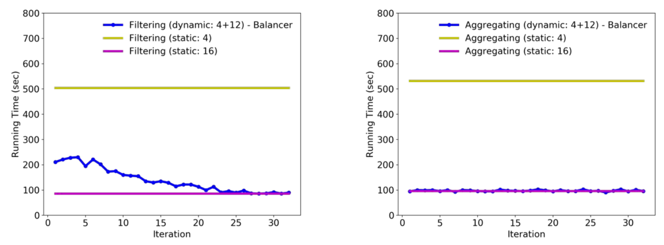
$14-16$ *containers). However, the number is around* $4-7$ *most of the time for the Filtering job in the dynamic mode.* The difference in container allocation for both jobs can explain their running time behavior as we explored in section III-A and the system resources utilization difference in previous section. In this case, the total running time of the UrbanFlow jobs over 10 iterations is 2272 seconds in the static mode, while 3762 seconds in the dynamic mode. The time difference (i.e., 1490 seconds) is mostly caused by the Filtering jobs with different data distribution. Since the container allocation is dynamic, one also needs dynamic models to predict/guarantee the applications' response time. Moreover, we analyzed the job counters over the 10 iterations. Two counters (i.e., data-local map tasks and rack-local map tasks) in the Filtering jobs are significantly different from each other in the two modes. Briefly, the average number of launched rack-local map tasks in the dynamic mode is much higher (i.e., $45:1$) than the one in static mode.

In summary, *UrbanFlow filtering job doesn't fully utilize some of the cluster's resources (e.g., CPU, Disk) in dynamic mode, because data imbalance leads to limited number of allocated containers. The network utilization is higher than the static mode, but it's not a dominant factor that affects UrbanFlow's performance, in the order of several seconds.*

C. A Tale of Three Data Redistribution Methods

To mitigate the overhead caused by data distribution when dynamically adding nodes to the Hadoop cluster, we conduct experiments on three data redistribution methods: (1) copy and rename (*Copy&Rename*), (2) change replication factor (*Replication Factor*), and (3) use Hadoop balancer tool (*Balancer*), as suggested by the Hadoop community [28]. In our experiments, we start with a 4-node cluster, dynamically add 12 nodes, and apply each data redistribution method in turn. Notice that, we run UrbanFlow for 32 iterations with one redistribution for each method, after that the running time of UrbanFlow in all the three cases converges. We then evaluate how they compare. We repeat the experiments five times with small variation ($\leq 4.5\%$). In all three cases, we see that the aggregating (reduce) component of the UrbanFlow application is largely unaffected by the data distribution and shows optimal performance for all cases. Below we discuss the behavior of the filtering (map) component of our application.

1) Copy&Rename: In this method, we first copy the dataset to a new location in HDFS which automatically redistributes

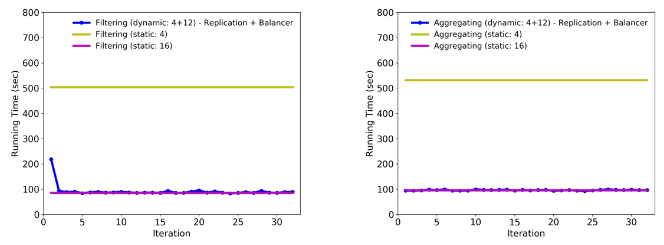


(a) Filtering performance (b) Aggregating performance

Fig. 9. UrbanFlow performance (Balancer)

it to the new number of nodes. We then remove the old copy, and rename the new copy (a quick operation on the order of milliseconds). To prevent confusion, the service needs to be interrupted when removing and renaming the data. However, we can overlap copying the data with the execution of filtering job and remove the old copy once filtering completes. As shown in Figure 7, the increase in filtering job's running time at the first iteration is caused primarily by the cost of performing the data copy operation in HDFS at the same time. Moreover, when copying the data, the cluster needs to store two copies of the data, thus, it takes double storage volume of the dataset for copying files. As shown in the figure, this method converges to the static mode in the second iteration.

2) Replication Factor: HDFS allows users to specify the number of replicas of a file (3 as the default value). If we set a higher replication factor, HDFS will automatically distribute the replicas among the new number of nodes. We leverage this feature in this method: we first turn up the replication factor of the dataset to 6, wait for transfers to stabilize, and then turn the replication back down which leaves us with the replica distributed over the new nodes. In our experiments (see Figure 8), we perform data redistribution



(a) Filtering performance (b) Aggregating performance

Fig. 10. UrbanFlow performance (Combination)

TABLE II
FILTERING COST ANALYSIS (MACHINE TIME)

Method	(1)	(2)	(3)	(2) + (3)	S4	S16
MAX	5152	3616	3680	3504	2240	1520
MEAN	1509.5	1613.5	2222.5	1482.5	2016	1380.5
MEDIAN	1376	1552	2008	1408	2162	1360
SD	666.7	373	779.2	371.9	296.1	54.9
Cost	48304	51632	71120	47440	64512	44176

via replication at the first iteration, which later gets filtering runtime close to optimal. Overall, the execution time increase caused by data redistribution is significantly lower (~30%) than in the *Copy&Rename* method; this is because UrbanFlow can make use of the replicated chunks while it's performing data replication. This method also doesn't interrupt the service, however it results in data distribution that is not well balanced among nodes and the associated slight increase in average execution time after redistribution. Indeed, the execution time increase depends on the resulted data distribution. Since HDFS will randomly remove over-replicated blocks from different nodes when turning down the replication factor, each run may result in different data distribution, and most of the observed execution time increase is greater than 12%. It also requires double storage volume since we set the replication factor to 6 and then turn it back down to 3.

3) Balancer: Administrators can rebalance the data across the data nodes using the Hadoop balancer tool provided by HDFS. It is a lightweight process that rebalances the data during operations. Through experiments, we know that this method results in a dataset that is well balanced among cluster nodes, and since it rebalances the data in place so there is no need for additional storage capacity as in previous two methods. However, it takes much more time to rebalance the dataset. In our experiments (see Figure 9), it takes around 3 hours to complete the rebalancing process. Notice that, we only show the results for the first 32 iterations, which took around 2.4 hours, since the running time becomes stable after that.

4) Combination: We note that these two methods are complementary to each other: method (2) doesn't distribute the dataset very well as method (3), while method (3) needs much longer time than method (2). In this approach, we first redistribute the dataset leveraging replication factor, and then use Hadoop balancer tool to refine the data distribution. Figure 10 presents the results. In the first iteration, it has similar runtime to method (2), and it converges to the static mode in the second iteration. Moreover, the Hadoop balancer just takes 25.4 minutes to redistribute the dataset.

5) Economic cost analysis: Figure 11 shows the cumulative economic cost of Filtering jobs at each iteration in all three cases, with two additional cases in static clusters as baseline. As can be seen, the *Copy&Rename* method is almost consistently more costly than the 16-node static cluster over all the iterations; The *Replication Factor* method has less cumulative economic cost than the *Copy&Rename* method

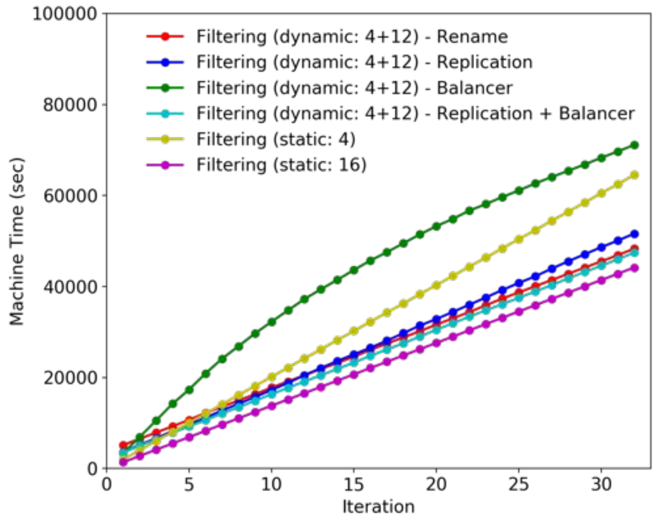


Fig. 11. Filtering cumulative economic cost over 32 iterations

before iteration 13, however, after that, it costs more than the *Copy&Rename* method; The *Balancer* method has the highest cumulative economic cost, even higher than the cost of 4-node static cluster for most of the iterations. Combining methods (2) and (3) gives a better option, since the cumulative economic cost is the smallest. Table II shows the detailed statistics of Filtering economic cost after convergence. *SD* represents standard deviation. Among the three methods, the *Combination* method has the lowest total cost for Filtering jobs, the *Copy&Rename* method has similar but slightly larger cost, while the *Balancer* method has the highest cost.

In summary, all the methods have advantages and disadvantages. The *Copy&Rename* method has optimal runtime in most cases as well as evenly distributed data −− but very high runtime impact during data redistribution and uses double storage. The *Replication Factor* method has smaller runtime impact during data redistribution and most iterations are close to optimal −− however, it ends up with less evenly distributed data and uses double storage. The *Balancer* method has smallest runtime impact during data redistribution as well as evenly distributed data and it does not require extra storage −− but on the other hand, it has slow convergence and high average runtime impact. Finally, combining the methods (2) and (3) has most of advantages overall, but it requires extra storage.

D. Synthetic Workloads

In this section, we conduct an experiment by looking at synthetic workload. The experiment will work off of the assumption that we initially don't redistribute the old data but add nodes only when we get new data, that is then distributed perfectly for the new number of nodes. We design a trace based on UrbanFlow that mixes requests for different data (old and new) for a period of time such that data gets updated daily, and the experiment runs for a week, with new dataset added to the Hadoop cluster each day. We assume that there

TABLE III
PERFORMANCE IMPROVEMENT (%) BY DATA REDISTRIBUTION

Zipf Exponent	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
0	8.37	19.07	26.3	29.36	30.63	32.56
0.5	7.1	15.94	22.03	24.3	24.88	26.37
1.0	5.84	12.75	17.42	18.78	18.55	19.41
1.5	4.68	9.74	12.98	13.42	12.56	12.82
2.0	3.64	7.16	9.13	8.95	7.76	7.64

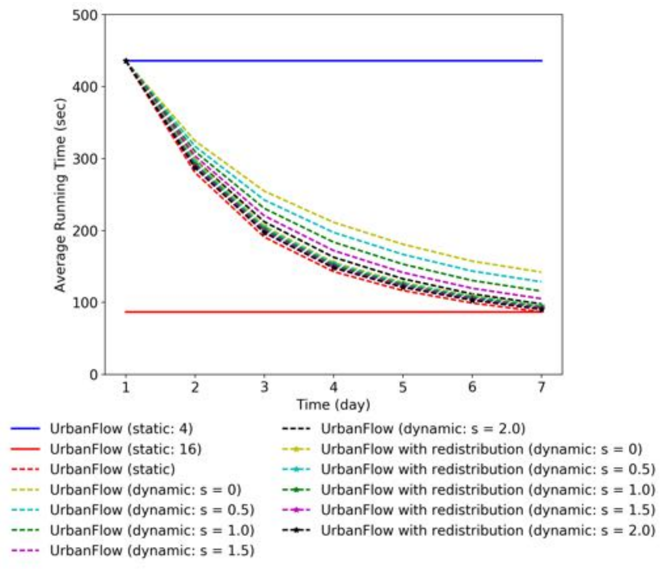


Fig. 12. UrbanFlow Filtering Response Time under Various Workloads

will be requests for a mix of old and new data with the old data gradually phasing out. To generate the request traces, we use a Zipfian distribution, which has been applied to model the Internet traffic [1], [3]. Briefly, the Zipf's law states that given a set of items, the frequency of any item is inversely proportional to its rank in the frequency table. So, the most frequent item will occur approximately twice as often as the second most frequent item, three times as often as the third most frequent item, etc. The probability distribution of an item with rank k out of N items is described by the equation below:

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N}(1/n^s)} \tag{1}$$

Where, k is the rank of an item, N is the total number of items in the set, and s is the value of the exponent characterizing the distribution. When $s = 0$, it means that all the items occurs with equal probability, and a larger s means larger skewness of the probability distribution.

We start with a 4-node cluster, and then we add 2 nodes to the cluster each day with new data coming into the cluster. Figure 12 shows the UrbanFlow filtering job's response time under various workloads, both with and without data redistribution. The workloads are generated by varying the zipf's exponent s, i.e., $s = 0, 0.5, 1.0, 1.5, 2.0$. When $s = 0$, all the data sets are requested equally. As one can see, with a larger value of s, the average response time becomes smaller. With data redistribution, the average response time approaches the static configuration. As we can see, when s is small, the benefits of data redistribution are obvious. More specifically, table III shows the response time improvement in percentage by data redistribution leveraging replication factor. When $s = 0$, the data redistribution improves the response time by $8.37\% - 32.56\%$ on average; when $s = 2.0$, the average response time is improved by $3.64\% - 9.13\%$. Since the aggregating jobs' response time for different traces is almost the same as explained in previous sections, we omit the results for the aggregating job.

The above analysis on workload shows guidelines on whether or not one should perform data redistribution on the data set depending on the QoS requirements of the Hadoop applications. Such workloads can be designed approximating the number of requests in time by using the request workloads from Wikipedia and FIFA'98 [25]. While this is not the focus of our paper, it points to an interesting direction for future work.

IV. RELATED WORK

In order to support a large number of users with varying workloads, and provide a controlled response time to users, the cloud computing platform must dynamically manage deployments of compute resources in such a way that they provide a stable response time. For Hadoop dynamic scaling, Kambatla et al. introduced a signature-based approach [11] to optimize Hadoop provisioning in the cloud. However, it doesn't show a solution to find the number of nodes needed in the cluster. Leverich et al. [13] proposed a strategy to allow the cluster to scale down when the workload is low to improve energy-efficiency. GreenHDFS [12] employs the distribution of the dataset in the cluster, proposed a hybrid multi-zone layout of hot and cold zones, and energy management policies to handle dataset in different zones. However, both of them don't solve the automation of scaling operations. Elastisizer [9] allows users to express cluster sizing problems as queries in a declarative fashion. Maheshwari et al. [17] proposed an auto-scaling algorithm for MapReduce framework simply based on the average cluster utilization. Herodotos Herodotou [8] proposed fine grained Hadoop performance models. Gandhi et al. [7] proposed a model-driven auto-scaling solution to dynamically determine the resources required to successfully complete the Hadoop jobs as per the user-specified SLA under various scenarios. Romer [20] proposed an auto-scaling framework for Hadoop clusters based on the average load with a cooldown period. Li et al. [14] proposed several schemes to automatically scale Hadoop clusters for dynamic geo-processing workload.

Moreover, there are some auto-scaling techniques for cloud applications [2], [5], [6], [10], [16], [18], [22]. Especially, Riteau et al. [19] introduced an auto-scaling algorithm for some web service based on the the number of current requests in the system. CherryPick [2] leverages Bayesian Optimization to automatically identify the optimal or near-optimal cloud configurations with low cost. Lorido-Botran et al. [16] classify auto-scaling techniques based on static threshold-based rules, reinforced learning, queuing theory, control theory, and time-series analysis. Although some excellent work has been proposed to scale Hadoop applications by adding resources to a computation based on need, the overhead of making the application aware of those resources can make this operation too costly, which has not been studied as extensively.

V. Conclusion

Our results show that while adding nodes dynamically to a Hadoop cluster without data redistribution for map components of applications brings improvements in terms of runtime, those improvements are significantly lower —— and costlier —— than can be expected of a cluster with optimal data distribution. To improve both factors we experimented with 3 methods for dynamic data data redistribution that can be applied to better utilize the additional resources. We found that each of the methods presents different trade-offs, of interest to providers and clients with different QoS requirements. The *Copy&Rename* method provides the most optimal response time for most runs however a small number of runs will see an outsize impact; it is thus most suitable for situations where response time QoS is defined as a high percentage of responses within a low time limit. The *Balancer* method on the other hand, has the lowest maximum response time that comes with high cost and is thus suitable for situations when we want to guarantee that response time will never exceed certain threshold. Combining different methods in certain cases can allow us to create a "best of both worlds" situation providing a good balance of different factors depending on specific response time and cost equation.

Acknowledgement

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. The project was funded by the NSF-1443080 award.

References

[1] L. A. Adamic and B. A. Huberman. Zipf's law and the Internet. Glottometrics, 3:143-150, 2002.

[2] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In NSDI, pp. 469-482. 2017.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," In INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 1, pp. 126-134. IEEE, 1999.

[4] Chameleon cloud: A configurable experimental environment for large-scale cloud research. https://www.chameleoncloud.org

[5] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. "Adaptive, Model-driven Autoscaling for Cloud Applications." In ICAC, vol. 14, pp. 57-64. 2014.

[6] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. "Model-driven optimal resource scaling in cloud." Software & Systems Modeling (2017): 1-18.

[7] A. Gandhi, S. Thota, P. Dube, A. Kochut, and L. Zhang. "Autoscaling for hadoop clusters." In Cloud Engineering (IC2E), 2016 IEEE International Conference on, pp. 109-118. IEEE, 2016.

[8] H. Herodotou. "Hadoop performance models." arXiv preprint arXiv:1106.0940 (2011).

[9] H. Herodotou, F. Dong, and S. Babu. "No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics." In Proceedings of the 2nd ACM Symposium on Cloud Computing, p. 18. ACM, 2011.

[10] K. C. Kamal. Performance Tuning of MapReduce Programs. North Carolina State University, 2015.

[11] K. Kambatla, A. Pathak, and H. Pucha. "Towards Optimizing Hadoop Provisioning in the Cloud." HotCloud 9 (2009): 12.

[12] R. T. Kaushik, and M. Bhandarkar. "Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster." In Proceedings of the USENIX annual technical conference, vol. 109, p. 34. 2010.

[13] J. Leverich, and C. Kozyrakis. On the energy (in) efficiency of hadoop clusters. ACM SIGOPS Operating Systems Review, 44(1), pp.61-65. 2010.

[14] Z. Li, C. Yang, K. Liu, F. Hu, and B. Jin. "Automatic scaling hadoop in the cloud for efficient process of big geospatial data." ISPRS International Journal of Geo-Information 5, no. 10 (2016): 173.

[15] Y.Y. Liu, A. Padmanabhan, S. Wang. CyberGIS gateway for enabling data-rich geospatial research and education. Concurr Comput.: Pract Exper, 27 (2) (2015), pp. 395407

[16] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano. "Auto-scaling techniques for elastic applications in cloud environments." Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09-12 (2012).

[17] N. Maheshwari, R. Nanduri, and V. Varma, "Dynamic energy efficient data placement and cluster reconfiguration algorithm for mapreduce framework.", Future Generation Comp. Syst., vol. 28, no. 1, pp. 119127, 2012.

[18] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In Autonomic Computing (ICAC), pages 6982. USENIX, 2013.

[19] P. Riteau, M. Hwang, A. Padmanabhan, Y. Gao, Y. Liu, K. Keahey, and S. Wang. "A cloud computing approach to on-demand and scalable cybergis analytics." In Proceedings of the 5th ACM workshop on Scientific cloud computing, pp. 17-24. ACM, 2014.

[20] T. Röme. "Autoscaling Hadoop Clusters." Master's Thesis, University of Tartu, Tartu, Estonia, 2010.

[21] K. Soltani, A. Soliman, A. Padmanabhan, and S. Wang. "UrbanFlow: Large-scale Framework to Integrate Social Media and Authoritative Landuse Maps." In Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale, p. 2. ACM, 2016.

[22] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics." In NSDI, pp. 363-378. 2016.

[23] T. White. Hadoop: The definitive guide. O'Reilly Media, Inc., 2015.

[24] H. Chouraria, (21 October 2012). "MR2 and YARN Briefly Explained". cloudera.com. Cloudera. Retrieved 11 November 2017.

[25] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada. "A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling." In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 64-73. IEEE Press, 2017.

[26] dstat(1) - Linux man page. Retrieved 9 November 2017 from https://linux.die.net/man/1/dstat

[27] OpenStack. Retrieved 18 November 2017 from http://www.openstack.org.

[28] Hadoop wiki. Retrieved 20 November 2017 from https://wiki.apache.org/hadoop/FAQ