Multi-level Scheduling Algorithm Based on Storm

Jie Wang, Siguang Hang, Jiwei Liu, Weihao Chen, Gang Hou
School of Software, Dalian University of Technology, DL 116620 China
[e-mail: hg.dut@163.com]
*Corresponding author: Gang Hou

Received October 20, 2015; revised January 5, 2016; accepted February 1, 2016; published March 31, 2016

Abstract

Hybrid deployment under current cloud data centers is a combination of online and offline services, which improves the utilization of the cluster resources. However, the performance of the cluster is often affected by the online services in the hybrid deployment environment. To improve the response time of online service (e.g. search engine), an effective scheduling algorithm based on Storm is proposed. At the component level, the algorithm dispatches the component with more influence to the optimal performance node. Inside the component, a reasonable resource allocation strategy is used. By searching the compressed index first and then filtering the complete index, the execution speed of the component is improved with similar accuracy. Experiments show that our algorithm can guarantee search accuracy of 95.94%, while increasing the response speed by 68.03%.

Keywords: Mixed load, long tail delay, online services, Storm

http://dx.doi.org/10.3837/tiis.2016.03.008 **ISSN : 1976-7277**

1. Introduction

Today, Internet enterprise application in the mass cluster system is divided into two types: One is directly oriented to mass users for online service application, such as online services, search engines, e-commerce websites, and social websites. These services are more sensitive to system response time, meeting a milliseconds-level of response degree; the other is an offline mass data batch processing job hidden from the system, such as MapReduce [1], figure calculation, and flow type calculation. Offline jobs involve large amounts of data processing, with no need to respond to requests, and result in a longer running time. To further improve the utilization of resources, it is common to mix online services with batch jobs deployed on the same physical node [2], forming a mixed load cluster. However, the mixture of online services and batch jobs deployed on a physical node will lead the two applications competing against the machine resources (such as cache and I/O bandwidth), and the batch job will interfere with the delay-sensitive online services. Reference [3] noted that batch jobs on different nodes involve different interference to online service components, resulting in a long tail delay for online service (the longest delay for components' implementation) and reducing overall performance.

To ensure execution performance of online service applications in the mixed load cluster, references [4,5,6,7], from the perspective of resource competition, suggested reasonable design of machine resource allocation and scheduling algorithms to prevent offline and online service application of resource sharing and competition (including machine system-level cache, central processing unit (CPU), and memory resources) so as to solve the performance interference problems in the application of online services. Tessellation [8] and Akaros [9] viewed the operating system as changing the system kernel and the process of abstraction to make the service application use the customized system resources to realize optimized performance. In addition, a technique to dynamically manage applications was proposed. The dynamic management technique meets applications' performance requirements at run-time according to the monitored interference metrics, such as the bandwidth reflecting I/O resource contentions [10] and the Last Level Cache (LLC) miss rate reflecting cache contentions [11]. Moreover, references [12] suggested implementation of performance prediction for the heterogeneous load using a collaborative filtering algorithm. According to the predicted results, the service class is applied to the cluster with little disturbance, which is similar to the idea of this paper, but it has a different level of scheduling. Moreover, all of the above-mentioned documents considered the online service application as a whole, from the perspective of the application of coarse-grained resource scheduling or application scheduling. They ignored issues relating to fine-grained latency variability of individual components. However, these components' tail delay dominates the performance of large-scale, parallel services.

From the perspective of online service of delicate granularity, reducing the long tail delay of the component can enhance the overall performance of the online service. Several tail delay reduction techniques have been proposed. Reference [3] mentions that long tail delay can send redundant requests to the online service application thus reducing the long tail delay of the component. There are two feasible schemes at present: (1) copying every request to multiple requests, sending to every component of the online service, and ultimately accepting the response rate of the fastest copy and ask results [13,14] and (2) within a certain time, the unperformed request will resend a new component to execute and accept the new executed results [15]. The above schemes can significantly improve the performance of service when service requests are low. However, when service request pressure is high or the machine is disturbed, the performance of the online service can drop significantly [16]. Another technique is the partially processing request. This kind of technique reduces tail delay by only using a portion of the quickest sub-requests or a synopsis representing the entire input data at a high level of approximation [17]. However, this technique sacrifices result correctness such as query accuracy to reduce service delay.

Many existing techniques have been developed to guarantee the performance of online service in the mixed load cluster. The scheduling operation can also be performed by load balance or even the cluster framework itself within multi-level components by applying the coarse-grained algorithm. Also, some fine-grained techniques that reduce the applications' long tail delay have also been proposed. However, note that our proposed scheduling algorithm is not intended to replace, but rather to complement, the existing scaling and resource provisioning techniques for multi-stage online services. Specifically, the proposed scheduling algorithm is enforced only after the machines have been allocated to the service.

The concrete contributions of this paper are described below.

(1) A multi-level scheduling algorithm based on Storm is proposed. It can improve online applications' performance by reducing the long tail delay. Compared with existing latency reduction techniques, the proposed scheduling algorithm has two different levels of scheduling: coarse-grained component−level scheduling algorithm and fine-grained component internal computing resource allocation algorithm.

(2) The component-level algorithm applied an analytic performance model to predict the latencies of all the components and their impact on the overall service performance, and then formulated the scheduling decisions based on the predicted performance. It will schedule the components that suffer more disturbances to the optimal physical node to perform. Thus, the component-level algorithm makes full use of the resources of the cluster.

(3) Computing the resource allocation algorithm is in the second level of scheduling; it is more fine-grained than the component-level algorithm. Within every search component, it searches the compressed index first and then filters the search index for the complete search to improve execution speed and accuracy of the search results.

The experiment results in a 100-machine cluster demonstrate that compared with the state-of-the-art techniques on reducing tail delay, our approach guarantees search accuracy of 95.94%, while increasing response speed by 68.03%.

2. Structure of the Scheduling Algorithm

Analyzing Google's announcement of the cluster load logs [18] (the Google cluster workload trace), many longer running jobs have relatively stable resource utilization [19], and online service on the same node suffers relatively stable performance interference. Therefore, this paper offers a design for a component-level scheduling algorithm based on real-time performance monitoring by the cluster machine at fixed time intervals (about 5 minutes) to predict the implementation of the performance interference Storm search service component. Based on the prediction results, the next time the degree of interference is larger, the service task will be assigned to machine resources which are relatively idle, thereby reducing the long tail delay for search service components to improve the overall execution performance. The execution performance of the offline jobs of the online service interference can ensure relative stability in accuracy of the performance prediction algorithm within a certain time. These are the source component-level scheduling algorithm design ideas. On the basis of the component-level scheduling algorithms, a computing resource allocation algorithm internal component is proposed in this paper. Through searching the compressed index first, then searching the complete index second, this algorithm allocates limited computing resources reasonably, which can enhance the execution speed of components in a more fine-grained angle and obtain more accurate results.

The overall framework of the scheduling mechanism includes three modules: the real-time monitoring module of machine performance, the performance prediction module of Storm search service components, and the core scheduling algorithm module, as shown in **Fig. 1**.

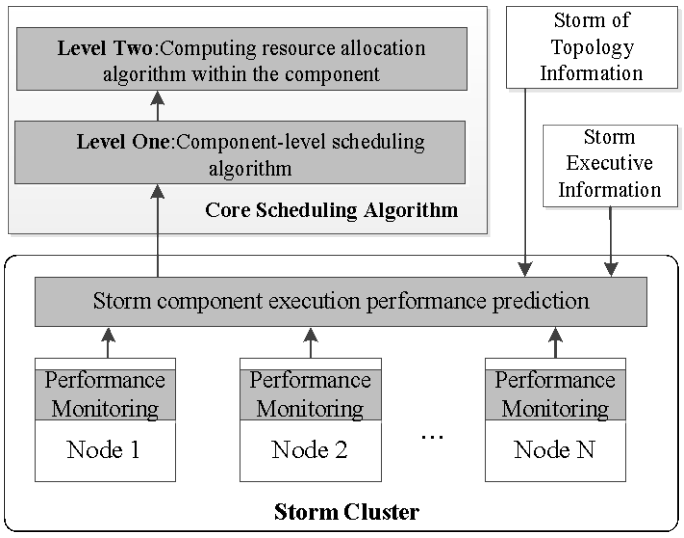

Fig. 1. Overall structure of the scheduling algorithm

The cluster performance monitoring module provides real-time access to two types of information in the cluster; one is status information of the Storm search service (such as request rate of search terms), the other is occupation information of mixed loads for machine resources in the cluster. A physical node will usually run multiple tasks simultaneously, so between machines and between each time period, the load on the utilization of machine resources is not identical. The monitoring module monitors the machine's CPU utilization and I/O bandwidth consumption.

The Storm search performance prediction module executes every time prior to dispatch for the cluster monitoring module to monitor information and, using this information to forecast Storm nodes on different machines, search for a service component of execution time. Meanwhile, according to Storm, search services in the cluster topology distribution also predict the overall response time of the system to perform a search. The forecasting module, based on the forecast of the performance of execution time, arrives at the performance prediction of the matrix, and the matrix will serve as the core scheduling algorithm for input. The key scheduling algorithm is divided into a component-level scheduling algorithm and internal resource allocation algorithm for calculating components. The component-level scheduling algorithms are used for coarse-grained scheduling tasks between cluster nodes. The internal resource allocation algorithm for calculating components is used for more fine-grained resource allocations. Hence, the long tail of search service components delay guarantee high accuracy in search results and improve their overall mixed load applications online in cluster service performance.

2.1 Performance prediction of search component service

To forecast Storm search service execution time for each component, first one needs to clear the factors affecting its execution time. Service on a particular component of the change in execution time should be searched, mainly for two reasons:

- Other loads' resource competition on the same node: If the physical nodes run on many different types of applications, multiple applications share the machine's hardware resources, including CPU, cache, disk, and I/O bandwidth. Competing machine resources between applications will be searched for fluctuations in service performance.
- Different queue time after submission of search words: Search response time includes queuing time and search time. A rate that is higher for search terms will cause a large number of search terms to queue; queuing time, even if the search time does not change, will lengthen the response times.

Therefore, in view of the above two factors, each component of the module includes the following steps to forecast Storm search time:

(1) From the cluster performance, monitor the module to obtain competitive information for the current machine resources (mainly for the use of CPU and I/O bandwidth).

(2) Search service under different search terms sending rate experiments, and perform and record the corresponding search service time and queue time. Use a fitting method based on actual test results of the cluster resource competition search execution time correlation function and send rates and different search terms of queue time-related functions.

(3) Searching time of every component for the next moment and overall response time are predicted by getting the transmission rate information of searching words from the current searching log and by using the fitting functions in step (2).

2.2 Component-level scheduling algorithm

The component-level scheduling algorithm in the text reduces the long tail delay of the searching component and improves the overall executed performance of searching by real-time performance monitoring of the cluster machines. It also improves predicting the execution performance interference on Storm search components at fixed time intervals (about 5 minutes) and, according to forecasts, the service components which experience interference the next moment are performed on machines whose resources are relatively idle. The symbols in the scheduling algorithm are as shown in **Table 1**.

Table 1.Symbols definition of the algorithm

Symbol	Meaning
N	A physical node in the cluster
C_i	The *i*-th component in topology
N_i	*i*-th physical cluster node
N_C	The number of components in the topology
N_N	The number of physical nodes in the cluster
$M_{N_C \cdot N_N}$	Performance matrix
$Arr[N_C]$	Component names are stored to be detected
$Allocation[N_C]$	Various components' distribution in cluster
C_{max}	The component has the longest implementation time
N_{Origin}	Original physical node where C_{max} components
$N_{Destination}$	C_{max} assembly to move the target node
ε	Assembly performance threshold is greater than the movement of the moved resource consumption

Steps in the scheduling algorithm are as follows:

(1) Obtain cluster machine performance monitoring information in every scheduling time interval (resource utilization information of I / O bandwidth of each node *N*).

(2) Construct the performance prediction matrix $M_{N_C \cdot N_N}$ based on performance of the prediction model.

$$M_{N_C \cdot N_N} = \begin{bmatrix} M[1][1] & \cdots & M[1][N_N] \\ \vdots & \ddots & \vdots \\ M[N_C][1] & \cdots & M[N_C][N_N] \end{bmatrix} \quad (1)$$

where any of the elements in the matrix *M[i][j]* indicate the change of execution time that it brings when the topology component$\mathrm{C_i}$ is moving into the cluster physical node N_j. If the value is positive, indicating that the performance of mobile $\mathrm{C_i}$ has been improved, the search execution time is reduced, as is the size of *M[i][j]*. If the value is negative, it indicates that the search time increased the value of *M[i][j]*.

(3) Maintain a component name to be the detected array $\mathrm{Arr[N_C]}$; elements of the array are required to be detected when it moves to other nodes to obtain promotion execution performance. Therefore, initialize the array elements to all the components in the topology collection, namely:

$$Arr[N_C] = \{C_1, C_2, C_3 \dots C_{N_C}\} \quad (2)$$

(4) Traverse $Arr[N_C]$, and get the current search execution longest component C_{max}. Currently, the execution time C_{max} components determine the response time of the entire search service.

(5) Traverse performance prediction matrix $M_{N_C \cdot N_N}$, and get the maximum matrix elements $M[C_{max}][N_j]$ from the matrix. The maximum value represents the C_{max} component after moving to the N_j physical node, and one will get the maximum performance and enhanced time of $M[C_{max}][N_j]$.

(6) Determine whether the value of $M[C_{max}][N_j]$ *i*s higher than the critical value ε of step 5.

When higher than critical ε:

It represents that the movement of the assembly will promote the overall system performance. The implementation of the second movement updates the C_{max} component node assignment information:

$$N_{Origin} = Allocation[C_{max}] \quad (3)$$

$$N_{Destination} = N_j \quad (4)$$

$$Allocation[C_{max}] = N_{Destination} \quad (5)$$

Deleted C_{max} component from $Arr[N_C]$,

$$Arr[C_{max}] = False \quad (6)$$

After moving the assembly, the performance of components on both physical nodes will be affected, so the performance prediction matrix $M_{N_C \cdot N_N}$ should be updated to update the performance prediction matrix operations, described later with examples.

If there is still an undetected topology component in $Arr[N_C]$, skip step 4 to continue; otherwise execute step 7.

If its value is lower than the critical ε, the components of a mobile lift of the time and resource consumption of moving performance is less than the event itself. If movement of the component is not performed, go to Step 7.

(7) The current obtained $Allocation[N_C]$ array is the optimal allocation algorithm. Realize IScheduler scheduling function interface, according to a new component distribution program to implement topology component scheduling.

The steps to update the performance prediction matrix are described below.

C_{max} will be moved from node N_{Origin} to $N_{Destination}$, On the one hand, the resources will be freed of theN_{Origin} node, so that implementation of the performance of other components on the node can be promoted. On the other hand, components will compete for nodes of $N_{Destination}$ for resources, thus reducing implementation performance of other components on the nodes. Therefore, moving a component will affect many other components, and the components must be updated for every movement.

The following illustrates the performance prediction analysis step of matrix updating:

(1) Equation (7) represents the performance of the prediction matrix Storm component within a scheduling period, and $N_C = 5, N_N = 4$, i.e., there are four physical cluster nodes, and the cluster topology has five components. Traversing the matrix, the element $M[1][2]$ is the highest. So $C_{max} = C_2$, $N_{Origin} = N_2$, and $N_{Destination} = N_3$ set the threshold ε = 5, then *M[1][2]* > ε comply with the conditions of moving. So, C_2 assembly is moved from node N_2 to N_3.

$$M_{5*4} = \begin{bmatrix} 0 & -10 & -20 & -3 \\ 19 & 0 & 20 & 15 \\ -4 & -25 & 0 & -2 \\ -6 & -30 & -7 & 0 \\ -5 & 0 & -8 & -4 \end{bmatrix} \tag{7}$$

(2) Equation (7) shows two C_2 and C_5 components before moving; theN_3 node has only one component, C_3, on the mobile node beforeN_2. After the C_2 component moves from N_2 to N_3, reducing the overall degree of interference of N_2 and increasing the interference of node N_3, the other component of a predicted value of execution performance on both nodes must be changed. Thus, the matrix M_{5*4} will update, and the first twovalues and the predictive value of the four must be updated. On the other hand, the mobile module C_2 reduces the interference to C_5 components, while enhancing components of C_3, so the performance predicted value of C_5 and C_3 needs to be changed,

and matrix line 3 and line 5 line need to be updated. At the same time, the performance prediction value C_2 component must be changed after the movement, so the matrix of the second row also needs to be updated.

(3) According to step 2, the updating results of matrix M_{5*4} are shown in formula (8), update 2, column 4, and rows 2, 3, and 5.

$$M_{5*4} = \begin{bmatrix} 0 & -5 & -25 & -3 \\ -8 & -20 & 0 & -2 \\ 2 & -10 & 0 & 4 \\ -6 & -20 & -15 & 0 \\ -10 & 0 & -18 & -8 \end{bmatrix} \tag{8}$$

(4) From formula (8), the matrix element of the updated maximum value of 4 is less than the critical value of 5, so no testing is needed for other components. The final results of the current node moved is the optimal allocation scheme of this scheduling.

2.3 Computing resource allocation algorithms in the internal components

For a search service, speed of response is one of the most important performance indicators. In some cases, obtaining a relatively better search result in a shorter time is more attractive than getting a best result in a longer time. Thus, Google and Microsoft's Bing return results to the user within a certain time and the vast majority of nodes is completed instead of search results not performing end node policy [3] (referred to in this article as partially adopted policies) so as to improve the responsiveness of the system as a whole. The components of the design solution based on this idea, at the expense of some accuracy of search results, is based on the guaranteed speed of search service, but different concrete implementations and some acceptance of the policy. Considering that part of the adopted node is discarded in the policy, because the node is associated with strong performance interference or with search terms on a page number, the execution time is longer and, in the end, it did not perform within the specified response time. Abandoning the nodes, although with a guaranteed response time, reduces the accuracy of search results and is a waste of computing resources.

Implemented components of the resource scheduling program are designed to respond to load changes within frequent data centers or machine resource utilization, relatively saturated scenes, through more granular allocation of computing resources of the component and well-designed search programs making full use of the computing resources of the component, guaranteeing a response within the required time and higher accuracy of related Web pages returned to the user. The algorithm is shown in **Fig. 2**.

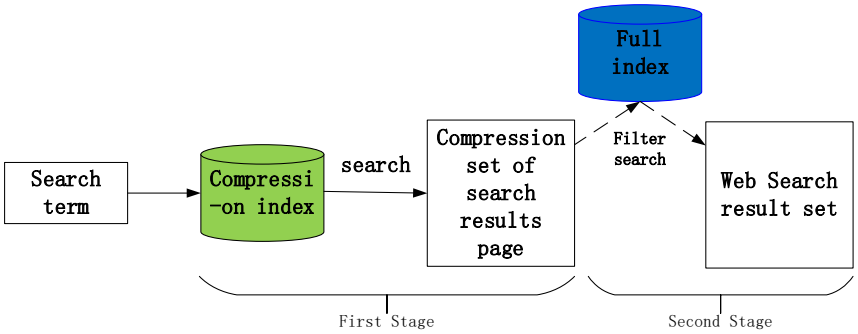

Fig. 2. Computing resource allocation algorithms in the internal components

The first-level search with Lucene indexing sets the compression algorithm for compressed indexes to a faster speed of execution and gets a page ID collection associated with a search term; the second-level filter search is related to collection page ID and subsequent packet filtering based on response time interrupts searching. Thus, the search program, when disturbed, returns fewer search results; less interference will return better search results. Thus, to ensure faster response times in the conditions, one must improve the accuracy of search results.

The algorithm symbols are defined as shown in **Table 2.**

Table 2. Symbol definitions of algorithm description

Symbol	Significance
$IndexAll$	All pages of the complete set of uncompressed indexes
$IndexZip$	Compression index set
$ResponseTime$	Predetermined response time for searching
$StartTime$	Beginning of the search time
$ZipResult[N]$	According to the degree of importance of the search word sorted, compressed Web pages, number of compressed pages related to N
$Original[M]$	M original page information is stored in a compressed page
$CurrentTime$	Search timestamp of currently executed
$Word$	Search term

The following is the concrete implementation algorithm:

Computing resource allocation algorithm stored within a component

1. Establish complete index set *IndexAll* and compression index set *IndexZip* for offline original page.
2. Set search response time *ResponseTime* and recording start time *StartTime*.

1. $ZipResult[N] \leftarrow Search(Word, IndexZip)$;
2. **for** $(i = 0; i < N; i++)$ **do**
3. $Original[M] \leftarrow ZipResult[i]$;

```
4.      FilterSearch(Word, Original[M], IndexAll);
7.   if((CurrentTime − StartTime) > ResponseTime) then
8.      break
9.   end if
10.end for
11.return the result of searching
```

In every search process, first perform searches to search terms on the compression set of indexes and store the compressed information in the array according to the web scores, from high to low (line 3).The higher the score, the higher the association between search terms and compressed web pages, so we think the bigger the web pages' contribution to the accuracy of the search results. Therefore, the loops of 4 to 10 rows of the algorithm, based on the compression degree of importance of web pages, compress web pages in turn, storing the compressed original information of the web page in the array first (line 5).Filter search refers to relevant analysis and grading only of the web page of the completed index set (line 6).Because filter search only calculates a part of the web page, the speed of execution is faster. When the operation of compressed web page filter search is performed in each execution, the execution time is calculated. If the length of the current executed time is up to the response time and terminates the loop, then all filter search results are returned.

Because the sequences of compressed web in the loop are arranged according to their degree of importance, even if a node's interference degree is bigger, it can search more relevant web pages first in the specified search time and ensure the accuracy of final results. In addition, this algorithm can ensure that each node returns some of the web page with which the node has the highest correlation index and search term, and not cause problems such as giving up the search results of the whole node in rejecting strategy. Usually, the abandoned nodes are susceptible to interference, and the execution speed is slower and still does not end the search in setting up the response time (but may be about to end the search).Therefore, from this point of view, this algorithm not only improves the accuracy of the final results, but it also improves the effective utilization of the cluster computing node.

3. The experimental results and analysis

This paper introduces experiments designed to simulate real mixed-load scenarios with cloud data centers, testing and analyzing scheduling algorithms.

3.1 Experiment environments

The experimental environment is described in **Table 3**.Ten physical nodes are in each cluster, and each physical node is created on 11 Xen virtual machines, including 10 for Storm nodes and 1 for Hadoop node. Thus, the cluster has 100 Storm nodes and 10 Hadoop nodes. The cluster nodes are connected to 1 GBPS network bandwidth.

Table 3. Testing environment

Item	Explanation
Operating system of node	SUSE Linux Enterprise Server 11 (x86_64)
Storm node Hadoop node	CPU：1Core，Memory：2GB CPU：2Core，Memory：4GB
Storm edition	0.9.3
Zookeeper edition	3.4.6
Lucene edition	4.10.3
Hadoop edition	1.0.2
JDK edition	1.7.0_65
Python edition	2.6

3.2 Experiment scheme

Two different types of Hadoop job were designed as the interference load of the Storm cluster: CPU resources-sensitive WordCount homework based on Hadoop platform and I/O bandwidth-sensitive Sort operations. The size of the input data set for the Hadoop job varied from 100 MB to 10gb, realizing different degrees of interfere with the actual performance of the Storm node. A Hadoop job was assigned on each physical node as the actual interference load for the cluster. Specific cluster structures are shown in **Fig. 3**. the topological structure of the Storm is constituted by the search term sending part, search component, and result integration. The one which has the function of the core search component is continued interference by the Hadoop offline operation so as to simulate real mixed-load disturbance.

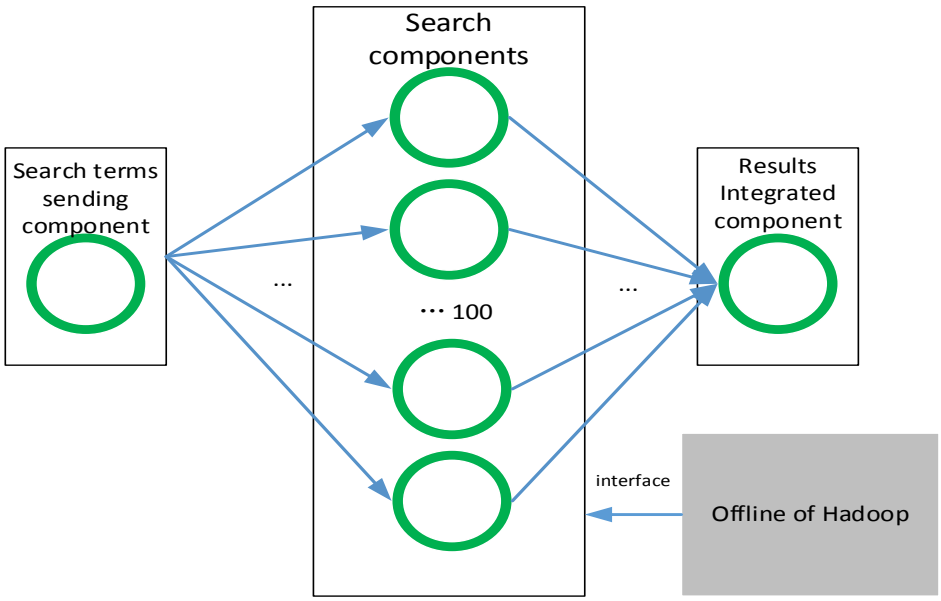

Fig. 3. Experimental cluster setup

In this paper, we designed two experiments. The first set of experiments, under the Hadoop load interference environment, set up a certain gradient stress test plan to search service itself. Based on results of the search engine, search stress mainly comes from growth in the search word request rate, so in this experiment, the response time requirements were not set (i.e., each search component searched out all the corresponding web pages), and the number of search words sent per second of the search term sending component was set at 10, 20, 50, 100, 200, and 500.Under pressure from different search word requests, we tested the performance of search service changes and evaluated the stability of the search service. Meanwhile, we used a state-of-the-art latency reduction technique called request redundancy [13, 14] as compared to an experiment. For each request, multiple replicas were created for parallel execution and only the quickest replica was used. We set the number of replicas at three for every request. The second set of experiments established different search response requirements, comparing the experiment based on the scheduling algorithm designed in this paper with the basic experiment without a scheduling algorithm. Then, according to the introduction part of section 3.4.1, we adopted strategies, setting up different adoption rates to test the search component, and then compared the accuracy of search results and response speed of search service with the algorithm proposed in this paper.

3.3 Analysis of experiment results

The first set of experiments, with no scheduling algorithm, requested a redundancy technique and scheduling algorithm designed in this paper; the paper describes the test under six different word sending rates of the Lucene search service. Among them, the comparison of long tail delays of search services is shown in **Fig. 4**.The diagram shows that the scheduling designed in this paper results in the shortest tail delay of Storm search service in all cases. Compared to the unused scheduling algorithm, under six different search pressures, our scheduling algorithm on average shortened the long tail delay as much as 66.43%. Also, even under greater pressure in searching conditions (such as sending rate of 200 and 500 search terms per second), the scheduling mechanism still showed a good performance advantage. As for the request redundancy technique, when the search term request rate is lower (e.g., 10 or 20), it achieves some latency reduction. However, when the request rate gradually increases to 500, this technique causes longer latencies.

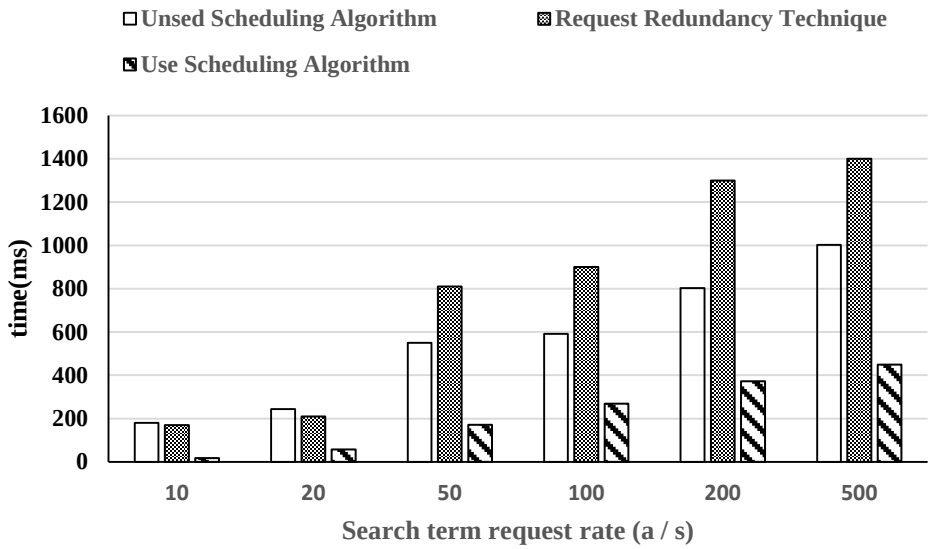

Fig. 4. Long tail search term request rate under different delay

We then analyzed the result of the average search time of search words; the test results are shown in **Fig. 5**.Consistent with the test results of long tail delay, the scheduling algorithm introduced in this paper decreased the response time of search service. Also, under differing search word request pressure, the search response time decreased by an average of 63.55%. Meanwhile, when the request pressure is high, the request redundancy technique case shows the worst performance. In contrast to the test results shown in **Fig. 4** and **Fig. 5**, when the long tail delay is longer, the corresponding search response time is longer. Thus, a long tail delay reflects the performance of the search service.

The second experiments simulated the real search term request rate for 30 minutes to determine the search request to obtain an average response time of 282.27 ms without using the scheduling algorithm introduced in this paper. This experiment set response times of 50%, 70%, 100%, and 150%, respectively, as the setting response time of components in computing resource allocation algorithm of about 150 ms, 200 ms, 300 ms, and 450 ms.

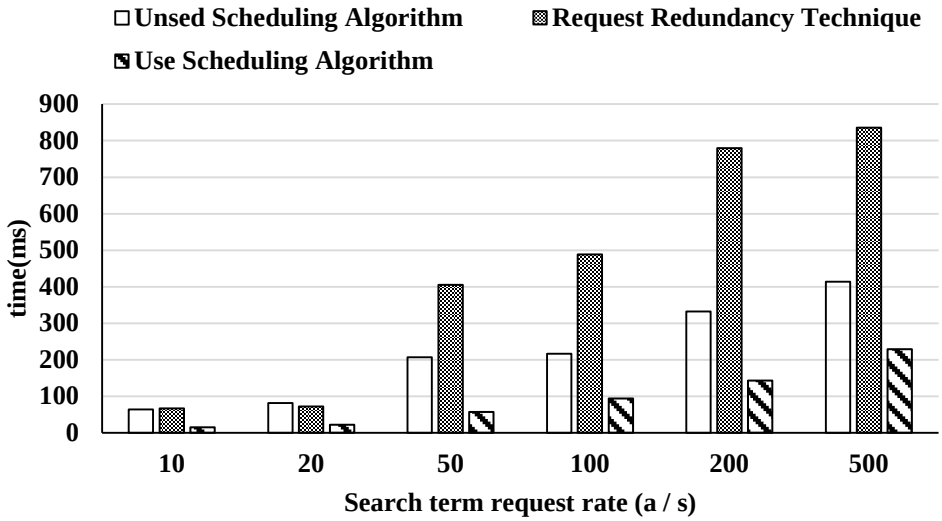

Fig. 5. The average delay request rate under different search terms

In the contrast experiment, 50%, 70%, and 80% were selected as the proportions of the search component, and the contrast experiment was carried out.

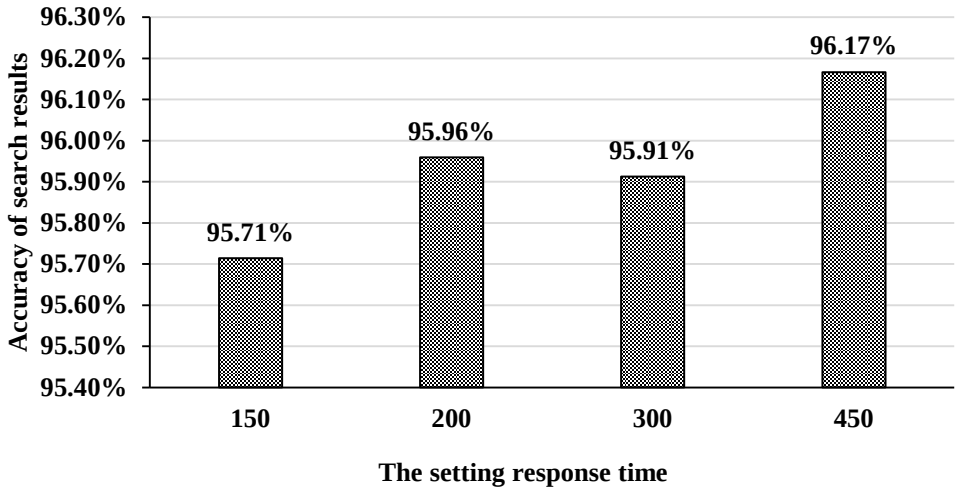

Fig. 6. Nagaodelay and the average response time under computing resource allocation algorithm in the internal components

In this paper, the average search time and the long tail delay of the algorithm in different response time parameters are shown in **Fig. 6**.The results of the default are based on the experimental results. As seen from the graph, the average search response time and the long tail delay of the components in the design of this paper are much smaller than the search service which does not use the algorithm. Moreover, the average response time and the long tail delay were shortened by 68.03% and 62.36%, respectively. This is sufficient to

demonstrate that the algorithm can effectively improve the performance of search services.

In addition, when the response time is 150ms or 200ms, the long tail delay is slightly larger than the set response time. On the one hand, in these two experiments, some components did not completely end the search. On the other hand, this paper introduces an algorithm mechanism for the computing resource allocation algorithm, the compression of web pages each time to start the search, and the implementation of the detection, which reduces the long tail delay phenomenon in the search engine service overall response rate.

Next, the accuracy of the algorithm was analyzed. In the experiment, the average number of Top10 pages in the search results of all search terms was calculated. The results of the search results were Top10 pages. Statistical results are shown in **Fig. 7**.

The average correct rate of search results is 95.94%; that is, the average response time of the algorithm is reduced by 68.03%, and the correct rate of the search results is 95.94%. As seen from the picture, with the increase in response time, the search results in the correct rate of continuous improvement, but the proportion is not high, indicating the importance of the algorithm to sort the search mechanism for higher relevance of the web search and return so as to improve the accuracy of the results.

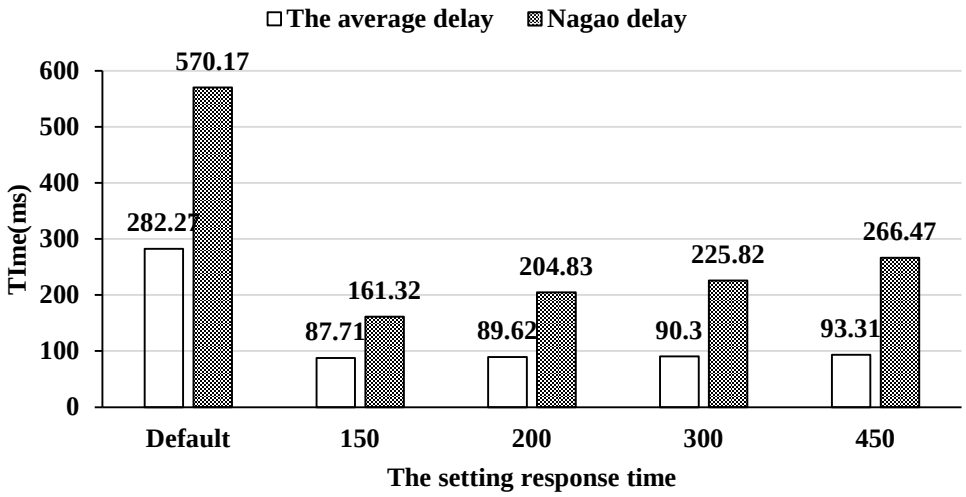

Fig. 7. Accuracy of the scheduling algorithm

In addition, in this experiment, the 300msresponse time of search accuracy was slightly lower than the 200ms response time of the accuracy. The following reasons can be considered: In the test of the 300msresponse time, some nodes experienced more serious interference with their performance, and the time of searching-program execution was longer, resulting in a smaller cycle searching ratio which is expanded by part of the searching components' compressed pages. Thereby, the accuracy of the results is lost.

The comparison of experimental results of the partially adopted policy is shown in **Table**

4. The table shows that when the adoption rate is 80%, the average response time (249.01ms) is close to a basic response time of the experiment (282.27ms). The accuracy of the searching results and the adopted ratio are in a linear proportional relationship. Moreover, when the average response time is 98.48ms (which is close to our designed response time of the algorithm), the accuracy of the result is only 68.75%, which is far lower than the 95.94% accuracy. Thus, our computing resource scheduling algorithm within the components, whose design is introduced in this paper, has more advantages in search response speed and results accuracy.

Table 4. Comparison of test results

Part of the adoption of proportion	Average response time（ms）	The accuracy of results
50%	65.20	51.43%
70%	98.48	68.75%
80%	249.01	82.68%

4. Conclusion

The purpose of this research is to solve the problem, the long tail delay, caused by offline batch jobs disturbing the performance of online service applications. In search engines for online service applications based on the Storm platform, overall execution performance is improved by reducing the long tail delay of the searching components at a more fine-grained component level. Two levels of the scheduling algorithm have been designed: component-level scheduling algorithm and internal-resource allocation algorithm of the components. The component-level scheduling algorithm makes full use of the resources of the cluster nodes; the internal-resource allocation algorithm of the components further effectively utilizes every component's internal computing resources. By testing scheduling algorithms under different searching pressures and comparing with current technologies, one can conclude that the proposed scheduling algorithm, in six different pressure experiments, on average improved the execution performance by 63.55%. Under the requirements of the setting response time, it also offers up to 95.94% search result accuracy. Therefore, the scheduling algorithm designed for this paper effectively reduced the long tail delay caused by the online service of the search engine in the mixed overloaded cluster and improved performance of search service execution.

Acknowledgement

This paper was supported by National Nature Science Foundation of China (No. 61472100, No. 61402073) and Fundamental Research Funds for the Central Universities, China (No.DUT14QY32). Thanks to Dr. Rui han for the help and guidance.

References

[1] J. Dean, S. Ghemawat. "MapReduce: simplified data processing on large clusters".*Communications of the ACM*, vol. 51, no.1, pp. 107-113, 2008. Article (CrossRefLink)

[2] C. Kozyrakis. "Resource efficient computing for warehouse-scale datacenters". In *Proc. Of the Conference on Design, Automation and Test in Europe*, pp. 1351-1356, March 18-22, 2013. Article (CrossRefLink)

[3] J. Dean J, L. A. Barroso. "The tail at scale". *Communications of the ACM*, vol. 56, no. 2, pp. 74-90, 2013. Article (CrossRefLink)

[4] H. Kasture, D. Sanchez. "Ubik: efficient cache sharing with strict qos for latency-critical workloads". *Association for Computing Machinery*, vol. 42, pp.729-742, 2014. Article(CrossRefLink)

[5] S. Govindan, J. Liu, A. Kansal and A. Sivasubramaniam. "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines". In *Proc. of the 2nd ACM Symposium on Cloud Computing*, pp. 22-35, October 26-28, 2011. Article (CrossRefLink)

[6] H. Yang, A. Breslow, J. Mars and L. Tang. "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers". In *Proc. of the 40th Annual International Symposium on Computer Architecture*, pp. 607-618, June 23-27, 2013. Article (CrossRefLink)

[7] Zhang X, Tune E, Hagmann R, R. Jnagal and V. Gokhale." CPI 2: CPU performance isolation for shared compute clusters". In *Proc. of the 8th ACM European Conference on Computer Systems*. pp. 379-391, April 15-17, 2013. Article (CrossRefLink)

[8] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird., M. Moretó, D. Chou ... and J. D. Kubiatowicz. "Tessellation: Refactoring the OS around explicit resource containers with continuous adaptation". In *Proc. of the 50th Annual Design Automation Conference*, pp. 1-10, May 29-June 7, 2013. Article (CrossRefLink)

[9] B. Rhoden, K. Klues, D. Zhu and E. Brewer. "Improving per-node efficiency in the datacenter with new OS abstractions". In *Proc. of the 2nd ACM Symposium on Cloud Computing*, pp. 25-32, October 26-28, 2011. Article(CrossRefLink)

[10] D. Xu, C. Wu, P. C. Yew. "On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling" In *Proc. of the 19th international conference on Parallel architectures and compilation techniques*, pp.237-248, 2010. Article (CrossRefLink)

[11] J. Ahn, C. Kim, J. Han, Y. R. Cho and J. Huh. "Dynamic virtual machine scheduling in clouds for architectural shared resources". In *Proc. of the 4th USENIX conference on Hot Topics in Cloud Computing*, pp. 1-5, June, 2012. Article (CrossRefLink)

[12] C. Delimitrou, C. Kozyrakis. "Paragon: QoS-aware scheduling for heterogeneous datacenters". *Acm Sigarch Computer Architecture News*, vol. 48, no. 4, pp.77-88, April, 2013. Article(CrossRefLink)

[13] A. Vulimiri, O. Michel, P. Godfrey and S. Shenker. "More is less: reducing latency via redundancy". In *Proc. of the 11th ACM Workshop on Hot Topics in Networks*, pp. 13-18, October 29-30, 2012. Article(CrossRefLink)

[14] G. Ananthanarayanan, A. Ghodsi, S. Shenker and I. Stoica. “Effective straggler mitigation: attack of the clones”. *Proc Nsdi*, vol. 21, no. 10, pp.185-198, April, 2013. Article (CrossRefLink)

[15] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin and C. Yan. “Speeding up distributed request-response workflows”. In *Proc. of the SIGCOMM 2013 and Best Papers of the Co-Located Workshops*, pp. 219-230, August 12-16, 2013. Article (CrossRefLink)

[16] C. Stewart, A. Chakrabarti and R. Griffith. “Zoolander: Efficiently meeting very strict, low-latency SLOs”. In *Proc. of the 10th International Conference on Autonomic Computing*, pp. 265-277, June, 2013. Article (CreossRefLink)

[17] R. Han, J. Wang, F. Ge, et al. “SARP: producing approximate results with small correctness losses for cloud interactive services”. In *Proc. of the 12th ACM International Conference on Computing Frontiers*. ACM, 2015. Article (CrossRefLink)

[18] J.Wilkes and C.Reiss. “Details of the ClusterData-2011-1 trace”. Article (CrossRefLink)

[19] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz and M. A. Kozuch. “Heterogeneity and dynamicity of clouds at scale: Google trace analysis” In *Proc. of the Third ACM Symposium on Cloud Computing*. p. 7, October 14-17, 2012. Article (CrossRefLink)

Jie Wang born in 1979. He obtained PhD of computer architecture at Harbin Institute of Technology, China in 2009. He entered School of Software Technology, Dalian University of Technology as a lecture until now. He interests parallel computing, network security and trusted software. Dr. Wang is a committee member of CCF YOCSEF (Dalian).

Siguang Huang born in 1994. Graduate student. His research interests include Cloud Computing and Distributed System.

Jiwei Liu born in 1989. He is a graduate student in School of Software Technology, Dalian University of Technology. He interests in parallel computing.

Weihao Chen, born in 1991. He is a graduate student in School of Software Technology, Dalian University of Technology. He interests in parallel computing.

Gang Hou, born in 1982. Lecturer and PhD Candidate. His research interests include model checking, trusted computing and embedded system.