A Fine-Grained Performance Bottleneck Analysis Method for HDFS

Yi Liu, Yunchun Li, Honggang Zhou, Jingyi Zhang, Hailong Yang, Wei Li

▶ To cite this version:

Yi Liu, Yunchun Li, Honggang Zhou, Jingyi Zhang, Hailong Yang, et al.. A Fine-Grained Performance Bottleneck Analysis Method for HDFS. 15th IFIP International Conference on Network and Parallel Computing (NPC), Nov 2018, Muroran, Japan. pp.159-163, 10.1007/978-3-030-05677-3_17 . hal-02279548

HAL Id: hal-02279548
https://inria.hal.science/hal-02279548

Submitted on 5 Sep 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

A Fine-grained Performance Bottleneck Analysis Method for HDFS

Yi Liu, Yunchun Li, Honggang Zhou, Jingyi Zhang†, Hailong Yang, and Wei Li

School of Computer Science and Engineering,
School of Instrumentation Science and Opto-electronics Engineering†,
Beihang University, Beijing 100191, China
{luice, lych, zhg, 16171093, hailong.yang, liw}.buaa.edu.cn

Abstract. The performance issue of HDFS has always been a great concern due to its widely adoption in both production and research environments. However, a fine-grained performance analysis tool is missing to effectively identify the bottlenecks as well as to provide useful guidance for performance optimization. In this paper, we propose a fine-grained performance bottleneck analysis tool, which extends HTrace with fine-grained instrumentation points that are missing in Hadoop official distribution. In addition, we propose an effective trace merging method that improves the understandability of our analysis. We analyze the performance of HDFS under different kinds of workloads and get undiscovered insights.

Keywords: HDFS, Instrumentation, Bottleneck Analysis, Performance Optimization

1 Introduction

Distributed file systems are widely used in various computing domains such as supercomputing and big data analytics. However, diagnosing performance issues of distributed file systems is still a challenging task, because the performance bottleneck of a distributed file system may come from various components of the system, and even interaction between different components. Therefore, effective performance analysis tools for distributed file systems such as Hadoop are of vital importance. Currently, many researches focus on end-to-end performance analysis frameworks, which capture the information flow of each request of the distributed file system and then obtain the performance information of each component of the system and the interaction between the components such as Dapper [5], Magpie, Stardust [6], Xtrace [1], HTrace [2], etc.

Among the above performance analysis tools, HTrace has been merged into Hadoop release to provide useful performance data. However, the default HTrace instrumentation within Hadoop has the following limitations for fine-grained performance analysis. Firstly, the default Hadoop provides very limited instrumentation points without detailed information captured. For example, the major components of HDFS [4] such as *Namenode*, *Datanode* and their interactions are

not instrumented. For example, we can not conclude whether *Namenode* bookkeeping is the bottleneck because Hadoop's official implementation haven't instrumented *Namenode*. Secondly, the default instrumentation in Hadoop cannot obtain the detailed parameter information for the function calls instrumented. For better analyzing the performance of a distributed file system, not only the time series of each function call but also the size of bytes processed by each function need to be known in order to identify the potential performance bottlenecks. Lastly, instrumentation information provided in default Hadoop is difficult to retrieve and visualize. For example, in just a few minutes, hundreds of megabytes of trace files are generated, making it hard to locate and diagnose the performance issues.

Therefore, this paper focuses on the performance analysis of HDFS by extending HTrace to provide fine-grained instrumentation. In addition to solve the trace explosion problem, we propose a trace compression method that merges the traces of repeated function calls and only maintains the representative statistics during instrumentation. Finally, through experiments on representative big data workloads, we obtain some useful insights.

2 Methodology

2.1 Fine-grained Instrumentation

The instrumentation of Hadoop's official distribution mainly instrument client sensed delay or *Datanode* sensed delay. HDFS contains more complex interaction beyond *Datanode*s and the client node. What's more, we can not distinguish network delay from the local file I/O delay. Due to this reason, we instrument some new performance-related blocks. They mainly reside in *Namenode*, *Datanode*s. Our purpose is to get fine-grained *Namenode* performance, *Datanode* local I/O performance, *Datanode* network performance, *Datanode* and *Datanode* data exchange performance. Except for simply obtaining function call duration, our instrumentation also encodes important function arguments into traces such as data size processed, block id and filename so we can monitor data process rate, I/O error occurrence. One of the biggest challenges of our instrumentation is that Java has many polymorphous functions. In the case, we will instrument every function and merge them in the trace processing procedure.

2.2 Trace Compression

The running of HDFS will generate a huge volume of traces. In our experiments, after several minutes of Spark execution, a trace larger than 1 GB will generate. Traditional HDFS performance analysis tools neglect this fact and rely on human labor to find the bottleneck in a large amount of data.

We present an effective method for compressing traces. We observe that before compressing, there are many repeated function call. For example, the *receiveBlock* function usually contains hundreds of *receivePacket* functions. We

merge repeated function call receivePacket in this circumstance and only extract several representative statistics from these merged function calls. The number of call trees will reduce by more than 90% after trace compression. Formally, we do a breadth-first traversal from bottom to top inside a call tree and merge the subtrees with the same structure. After compression inside every call tree, we compress these trees with the same structure.

3 Evaluation

3.1 Experiment Setup

Our experiments are conducted with a cluster with seven nodes with one master node (which is *Namenode* in HDFS), five slave nodes (which are *Datanode* in HDFS) and one client Node. The master node and slave nodes are equipped with Xeon E5-5620, 16GB memory. To achieve higher throughput, we use Intel Xeon Phi (Knights Landing) for workload generating. The many-core and high volume of memory enable Phi to start many HDFS clients simultaneously. The implementation is shown in Figure 1. The trace is generated into local files that are collected and stored in database. The *Workload Generator* component generates HDFS I/O requests. The *Recover Call Tree* component reads from user configuration to decide the function calls to keep. The *Trace Compress* component traverses call trees and performs compression. Finally, the *Bottleneck Visualization* component displays the compressed call tree.

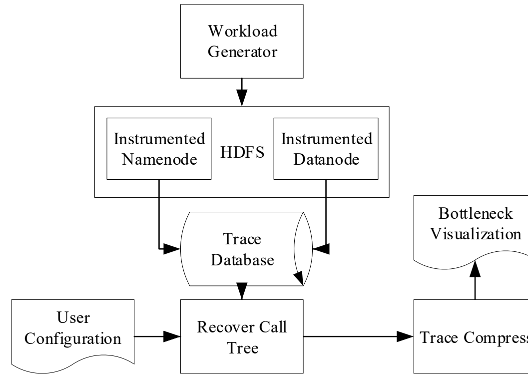

Fig. 1. The implementation overview of our HDFS performance analysis tool.

3.2 Performance Bottleneck Analysis

Across Workloads - We choose the tiny sized workload input from *Hibench*. For machine learning workloads, data will be iterated for many times generating large traces thus we use sampling (sample rate is 0.05) to reduce trace size. For *Wordcount* workload, the largest delay is caused by *FileSystem#createFileSystem* which spends total 90.21s. The second largest delay is caused by *DFSOutputStream#close* which spends total 10.54s. Local I/O plays an ignorable role here. The delay of *Datanode* flushing buffer into local file system is too small to measure. And also we can conclude that using faster storage medium won't speed up application greatly. We can see the bottleneck is in the client node. The process for initiating *FileSystem* object has a large potential to optimize. We have a similar conclusion for *Sort*, *Terasort*, *Pagerank*, *LogisticRegression* and *Nweight* workloads. *Bayes* workload is different from the above workloads. The largest delay is caused by *BlockSender#*sendBlock. Reading from local file system causes 3.10s delay and reading from remote *Datanode* causes 0.49s delay.

Impact of File Size - We use *Wordcount* workload to explore the impact of file size on HDFS performance. We use tiny, small, large, huge sized workloads which contains 32000, 320000000, 3200000000, 32000000000 respectively. Due to the trace size explosion, we use a sample rate of 1, 0.01, 0.001, 0.0001 respectively. With the increase of data size, the impact of *FileSystem#createFileSystem* is becoming weaker. In tiny sized workload, this operation causes total 91.92s delay compared with application time 28s(we add up delay from different *Datanodes*). In small-sized workload, it takes 1.55s compared with application time 32s. And in larger sized workloads, it hasn't been sampled. So in small-sized workloads, the file system creation process is an import bottleneck.

Impact of Request Frequency - In [3], the authors directly model real request patterns from the AliCloud on IOPS, Inter-arrival time, session size and read request size. However, Alicloud is a very large cluster contains tens of thousands of nodes. For our small cluster, we multiply IOPS with different factors α but retains the distribution the model. With request frequency increasing, we can explore which part of HDFS facing the request pressure as shown in Table 1.

The delay of request mainly caused by *sendBlock* operation. However, the average delay of this operation is decreasing. Although *FileSystem#createFileSystem* plays an important role in request delay, its duration has little to do with request frequency. We can find out that the delay of *sendBlock* first increase but then decrease with request frequency increasing. In small frequency, HDFS is in cold start state thus the delay is relatively large. And when request frequency is very large, the resource contention is more severe. The *BlockSender#sendPacket*, *FSNamesystem#getBlockLocations* (*Namenode* searching for block locations for a given file) operation has the same conclusion. Contrary to common sense, the bottleneck under frequent request is neither in *Namenode* nor in *Datanode*. Thus optimization for the concurrent request in client node is more important.

Table 1. The average delay (second) of some major functions under different request frequencies.

Function	$\alpha = 0.001$	$\alpha = 0.005$	$\alpha = 0.01$	$\alpha = 0.05$	$\alpha = 0.1$	$\alpha = 0.5$
FileSystem# create-FileSystem	0.1567	0.1842	0.1827	0.2060	0.1887	0.1992
sendBlock	0.0054	0.0026	0.0020	0.0012	0.0009	0.0015
BlockSender# send-Packet	0.0005	0.0004	0.0002	0.0001	0.0001	0.0002
FSNamesystem# getBlockLocations	0.0088	0.0006	0.0005	0.0003	0.0007	0.0014
total	0.0423	0.0439	0.0426	0.0648	0.0801	0.0674

4 Conclusion

In this paper, we propose an extension to HTrace in order to support fine-grained performance bottleneck analysis for HDFS. In addition, we propose a trace compression method to merge the repeated function calls for efficient performance analysis. We've also done a series of experiments to explore the bottleneck under different workloads and get useful insights.

Acknowledgment

The authors would like to thank all anonymous reviewers for their insightful comments and suggestions. This work is supported by National Key Research and Development Program of China (Grant No. 2016YFB1000304) and National Natural Science Foundation of China (Grant No. 61502019). Hailong Yang is the corresponding author.

References

1. Fonseca, R., Porter, G., Katz, R.H., Shenker, S., Stoica, I.: X-trace: A pervasive network tracing framework. In: Proceedings of the 4th USENIX conference on Networked systems design & implementation. pp. 20–20. USENIX Association (2007)
2. HTrace, A.: htrace. https://htrace.incubator.apache.org/ (2015)
3. Ren, Z., Shi, W., Wan, J., Cao, F., Lin, J.: Realistic and scalable benchmarking cloud file systems: Practices and lessons from alicloud. IEEE Transactions on Parallel & Distributed Systems **PP**(99), 1–1 (2017)
4. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on. pp. 1–10. Ieee (2010)
5. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Technical report, Google, Inc (2010)
6. Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., Ganger, G.R.: Stardust: tracking activity in a distributed storage system. In: ACM SIGMETRICS Performance Evaluation Review. vol. 34, pp. 3–14. ACM (2006)