Asian Journal of Information Technology 11 (5): 181-185, 2012
ISSN: 1682-3915

An Intelligent Cloud System Adopting File Pre-Fetching

M.R. Sumalatha, C. Selvakumar, B. Jagadeesh,
V. Bhoopesh Kumar and R. Baluprasad
Department of Information Technology, MIT Campus,
Anna University, Chennai, Tamil Nadu, India

Abstract: Cloud basically deals with huge volumes of data and each cloud user stores and accesses gigabytes of data. Cloud applications that require very fast data access are emerging quite frequently. Hence, it is of utmost importance to provide a high performance with reliability and efficiency. The Hadoop Distributed File System is one of the most widely used distributed file systems. Such cloud systems must provide efficient caching mechanisms in order to improve the speed of accessing the data. However, due to inefficient access mechanisms of the Hadoop, the access latency is too high that it reduces the throughput of the system. An efficient pre-fetching technique in Hadoop improves the overall performance of the cloud system.

Key words: Cloud, Hadoop Distributed File System, file pre-fetching, caching, efficiency

INTRODUCTION

Cloud computing is now being utilized in every field of computer science. Data intensive applications and computation intensive applications are dependent on cloud. Cloud has growing needs in the field of Web Services and Web Hosting. Recently, Internet service file systems are extensively developed for data management in large scale Internet services and cloud computing platforms (Tantisiriroj *et al*., 2008). Individual organizations also have cloud systems to insure that their works are done at the highest possible speeds. Many of the applications that execute on the cloud require more number of files for their computation and processing. These systems incur a major overhead of having to process all those file requests (Soundararajan *et al*., 2008). The latency involved in accessing and transferring the files exceed the computation time to a considerable level.

Hadoop is a framework that supports data intensive distributed applications under a free license and hence, it has become one of the popular distributed file systems. For instance, Yahoo manages its 25 petabytes of data with the help of the Hadoop System at its lower level (Shvachko *et al*., 2010). Due to the access mechanisms of HDFS, the access latency is extremely serious and is the worst when a large number of small files are to be accessed.

Pre-fetching with the help of relationship among files is considered an efficient method to alleviate access latency (Shriver *et al*., 1999). This provides the mechanisms for caching the required files before hand that is even before it is requested. Currently, the HDFS does not provide any such mechanisms. This methodology exploits the fact that the users or processes often follow an access pattern using which a correlation between the files is established. This relationship can be used in the later stages for file pre-fetching.

In this study, an algorithm is proposed that can effectively establish the relationship among the files and using the relationship structure, pre-fetch the files as when they are needed. The algorithm also ensures dynamic adaptability of the pre-fetching activity and the pre-fetch speed, based on the speed with which the files are accessed, thereby ensuring that the pre-fetching does not exceed the limit needed and that the cache space is also used effectively.

LITERATURE REVIEW

The Hadoop is the most widely and easily available distributed file system. An improvement in its efficiency will be one of the most recognized ones. Currently, proposals have been made for implementing the pre-fetching techniques in the Hadoop (Dong *et al*., 2010). But no algorithms have been proposed so far to implement the pre-fetching in cloud.

The main goal of the cloud service providers is to increase the performance of the cloud for the user and at the same time increase the profits for their organization. This is achieved by a strategy called the optimal service pricing of the cloud (Kantere *et al*., 2011). The pricing solution employs a novel method that estimates the correlations of the cache services in a time-efficient manner. So, caching plays an important role in providing the quality of service to the client and increasing the profitability for the cloud service provider. On adopting

Corresponding Author: M.R. Sumalatha, Department of Information Technology, MIT Campus, Anna University, Chennai, Tamil Nadu, India

the pre-fetching strategy, it is equally important to concentrate on the other parameters of the cloud system like performance, load balancing, fault tolerance, etc. (Wu *et al*., 2010).

PROPOSED WORK

The main aim of the research is to reduce the latency involved in the file access by introducing the concept of file pre-fetching. This is achieved by using a correlation detection module that can be used to establish the relationship between the files and a pre-fetching module that exploits the correlation module to pre-fetch the files. The basic structure of the Hadoop System (HDFS) which is depicted in Fig. 1 involves:

- Name node which serves as the master for the Cloud System to which all the requests from the client are directed. It has the meta-data about the files
- Data nodes which are the data centers
- User nodes/client which issue the requests

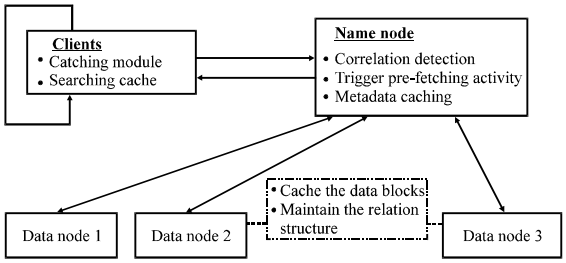

Fig. 1: Hadoop with the related modules

The proposed pre-fetching strategy takes into considerations the network load, the load on the system and the throughput of the system (Jia *et al*., 2010) while pre-fetching is triggered. The caching of files also happens depending on these parameters. Another important parameter to be taken into consideration is the pre-fetch speed also called the pre-fetch degree. The degree to which the pre-fetch activity has been triggered for a file should be within the limits of the speed of access of files by the process that is currently using those files. The algorithm adjusts the pre-fetch speed based on the file access speed by the process.

IMPLEMENTATION OF SYSTEM

In this study, the algorithm to adopt the file pre-fetching strategy in the Hadoop System is discussed. The two most important modules that are used include the file correlation detection module and the file pre-fetching modules, along with the processes involved in it (Dong *et al*., 2010). The interaction among the different modules is given in Fig. 2. The name node, being the master node of the Hadoop, accepts the request for file access from the clients. It takes up the responsibility of detecting the correlation between the files. It maintains a

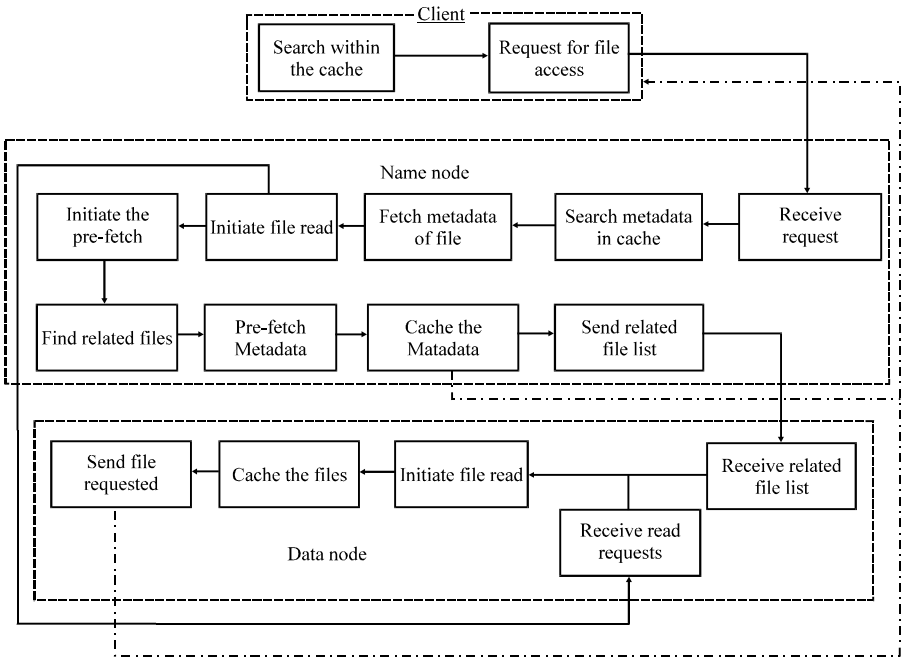

Fig. 2: Interaction among the components of Hadoop

tree for every directory that the user holds which actually represents the relationship existing between the files of that directory.

The client needs an efficient caching mechanism to cache both the metadata of the file as well as the file itself. The data nodes are also provided with the caching mechanisms to cache the data files that are being fetched and pre-fetched. Also, it has to maintain the relevant data structures to ensure the file correlation is maintained. The steps in the algorithm are briefed as follows:

- The client before requesting the name node for the file, checks its cache if it has the metadata about the file it needs. If it has then it can directly fetch the file from the appropriate data node. Otherwise, the request is sent to the name node
- The name node will call the file relation module when the file is used for the first time by any process/user
- The name node upon receiving the request will start fetching the file from the data node, based on the metadata of that file (Shvachko *et al*., 2010)
- Simultaneously a pre-fetch activity is triggered for that file to retrieve the list of related files which make use of the file relationships (If the file has no relationships established during that access then no pre-fetch activity will be triggered)
- The pre-fetching module fetches the metadata of files to be pre-fetched. They are then returned and stored in the client if the load is low; otherwise stored internally
- The data nodes as per the instructions of the name node will fetch the files and cache them in its memory or in the client based on the load on the network
- The name node also adjusts the pre-fetch speed based on the file access speed of the process currently using the files

A description of the core modules that were implemented is as follows:

File correlation detection module: The file relationship structure is the most important component in triggering pre-fetch, storing the sequence of file accesses made by a process. For the purpose of easy understanding, the relationship structure is established in every directory accessed, relating the files of that directory alone and is stored in a separate file within the same directory. A tree is maintained to establish the relation between the files. Access count is one important parameter that influences the extent of the file relationship. The more the access count a file, the more is the probability of accessing the file next in the sequence. Each node in the tree will have the following fields:

- Name of the file
- Frequency of access
- Its successor file

A hash table is also maintained that contains the information about the files that are newly opened. This hash table ensures that the tree is not over congested by adding more and more nodes which will make the traversal in the tree extremely difficult. Hence, the new file accesses are first hashed in the table and after the number of accesses reaches a threshold, the entries are transferred to the tree. This is an optional structure, simplifying the tree. Construct the tree for the directory as follows:

```
Build Relationship (T)
Input  : Process Accessing the files
Output : Relationship Structure

1. for each new incoming request for file f
2.   if f is first accessed from the directory
3.       add f as root;
4.       increment access count;
5.   else if f present in tree as root
6.       increment access count of f;
7.   else if predecessor (p) of f is present
8.       if f not present
9.           add f as successor of p;
             increment the access count;
10.      else
11.          increment access count;
12. else
13.     search hash list;
14.     if f found
15.       increment file access count;
16.       add f to tree if threshold access count
            is reached;
17.  else
18.       insert f into hash list
```

Pre-fetch activity: After the files are related using the tree as discussed, the next step is to exploit this relationship in pre-fetching the files. The file that is currently accessed is first located in the tree of its directory. The path along the decreasing order of the access frequency is then followed to find the files for pre-fetch. This path represents the files having the highest probability of being accessed in the sequence.

The files are pre-fetched from the data nodes by sending the list of related files obtained above. The name node takes care of retrieving the metadata of all the related files and caching them either in the name node itself or in the client depending on the network load. This metadata serves the important purpose of locating the data node where the actual file is present. The pre-fetched files and metadata are stored in a software cache. Due to the

limitations of the space of the cache, a reference strategy is introduced. In cases where the file size is small (say 1 kb as used in implementation), it is fetched from the data node and its contents are cached. In cases of large files, only the metadata, representing the location of the file is cached. The client can then use this information to fetch the required file directly from the data node, without a request being issued through the name node thus reducing the overhead involved in requesting through the name node. This ensures that there is an effective use of the cache.

The name node also monitors the speed with which the files are being accessed. After a set of few pre-fetches if the file access speed is detected to be slow then depending on the interval between two file accesses the next pre-fetching interval is adjusted. If the limit exceeds a certain value (1 min as assumed in the implementation) then the process is assumed to be in sleep or is dead. In that case, the pre-fetching activity serves no purpose and is completely stopped:

```
Prefetch_Module
Input: Relationship File
Output: Loaded Cache

1. find the directory of the file being accessed (f)
2. Load the relation file of f
3. if metadata of f present in cache
     directly access the datanode
4. else send request to the same node to fetch file
5. search for related files
6. check load on network and Namenode
7. if network load is low
     prefetch metadata of related files
     cache metadata in the client
     cache the file in the client
     send list of related files to datanode
8. else if network load on namenode is high
     prefetch metadata of related files
     cache metadata in namenode.
     if process active
        continue caching
        change prefetch speed
     else if process is slow
        stop prefetch
```

EXPERIMENTAL EVALUATION

The pervious algorithm is implemented in a cluster of 4 PCs. All the PCs are installed with CentOs 5.8.19. One of the four PCs is configured as the name node. It has i3 Intel CPU 2.10 GHz, 4 GB memory and 500 GB disk. The other three nodes act as the data nodes. Each of them has Pentium Dual-Core CPU 2.10 GHz, 3 GB memory and 320 GB disk. Hadoop needs Java for working. Java Version 1.7.0 is installed and over which Hadoop 0.20.203.0 is installed. Files, amounting to 1 GB in size are uploaded to the Hadoop Filesystem, a majority of them being small files.

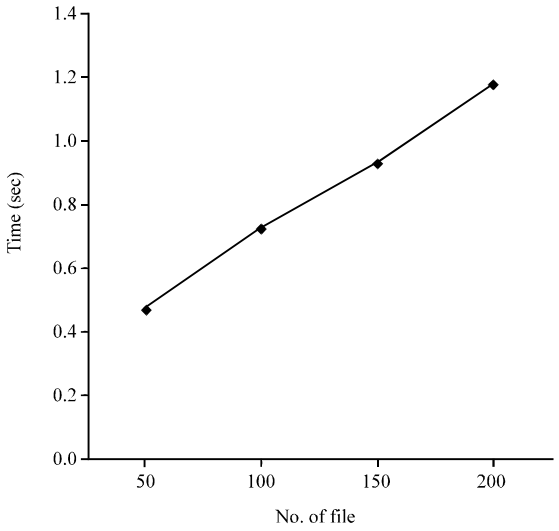

Fig. 3: Time taken (sec) for establishing relation structure

The algorithm is designed keeping in mind that there is some definite access pattern for the files. This is actually true in most cases of the software processes that are used to run some applications. This is explored in the algorithm for establishing the relationship among the files and pre-fetching. A process is taken as the input for the algorithm. The process is simulated to use varying number of files so as to analyze the algorithm effectively. In order to analyze pre-fetching, first the correlation is established among the files being accessed by the process and the same is stored as a file in the directory. The process may use files from different directories. The module is designed to form the appropriate structures for the files from every directory and store the relation in their respective directory. The time taken to establish the same is noted for different number of files as in Fig. 3. The process which access a set of 150 files are first made to run and the file-correlation is established for different number of clients.

After the establishment of the relation structures, the same set of 150 files is downloaded in the presence and absence of the pre-fetching techniques and the download times are found to follow the graph in the Fig. 4. The pre-fetching module makes use of the file that represents the relationship among the files in every directory accessed by the process. The name node will fetch the list of related files for the given file and will retrieve the metadata for the same, caching them either in its cache or in the client based on the network conditions. Simultaneously, it also tries to pre-fetch the data from the data nodes by sending the list of related files to it. The client, on accessing the files will try to find the file in the cache. If either the metadata of the file or the file content

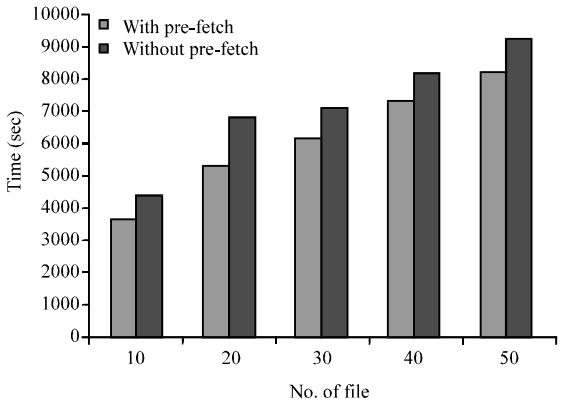

Fig. 4: Download time (msec) vs. No. of file requests

is present in the cache then there arises no need for the client to send the request to the name node. The file content is used as such whereas the metadata of the file is used to fetch the file from the data node directly without the intervention of the name node. This greatly reduces the access latency involved in file accesses.

It is obvious from the graph that with the introduction of pre-fetching, the time taken to download the same set of files has been reduced considerably. The difference found to be nearly about 1 sec at the minimum. Since, large scale applications will involve a lot of such file requests, there will be very good increase in the performance. So, the time taken for the establishment of relationships can be ignored. It can be noted that as the number of clients increase, the download time is also found to increase. This is due to the fact that there is just a single namenode for all the clients which has to perform all processing related to the file pre-fetching, incurring an overhead and thus a lowered response. But still, the end result is found to reduce overall latency involved in file accesses, proving the efficiency of the technique. Therefore, the proposed algorithm is found to increase the performance of the Hadoop Distributed File System by reducing the latency involved in file retrieval.

CONCLUSION

Hadoop, though is widely used is suffering from the access latency involved in reading large number of files. In this study, the pre-fetching mechanism and the algorithms that are discussed are found to be very effective in alleviating the access latency while at the same time, monitoring the cloud parameters to ensure that the pre-fetching activity does not affect the cloud performance.

RECOMMENDATIONS

As a future research, more sophisticated policies are to be experimented, taking into considerations the presence of replica for further improving the efficiency of the HDFS pre-fetching.

REFERENCES

Dong, B., X. Zhong, Q. Zheng, L. Jian, J. Liu, J. Qiu and Y. Li, 2010. Correlation based file prefetching approach for hadoop. Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science, November 30-December 3, 2010, Indianapolis, IN., pp: 41-48.

Jia, B., T.W. Wlodarczyk and C. Rong, 2010. Performance considerations of data acquisition in hadoop system. Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science, November 30-December 3, 2010, Indianapolis, IN., pp: 545-549.

Kantere, V., D. Dash, G. Francois, S. Kyriakopoulou and A. Ailamaki, 2011. Optimal service pricing for a cloud cache. IEEE Trans. Knowledge Data Eng., 23: 1345-1358.

Shriver, E., C. Small and K.A. Smith, 1999. Why does file system prefetching work? Proceedings of the USENIX Annual Technical Conference, Monterey, California, USA., June 6-11, 1999, USENIX Association Press, Monterey, CA., USA., pp: 71-84.

Shvachko, K., H. Kuang, S. Radia and R. Chansler, 2010. The hadoop distributed file system. Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, May 3-7, 2010, Washington, DC., USA., pp: 1-10.

Soundararajan, G., M. Mihailescu and C. Amza, 2008. Context-aware prefetching at the storage server. Proceedings of the USENIX 2008 Annual Technical Conference, June 2008, Berkeley, CA., USA., pp: 377-390.

Tantisiriroj, W., S. Patil and G. Gibson, 2008. Dataintensive file systems for internet services: A rose by any other name. Technical Report, CMU-PDL-08-114, Carnegie Mellon University.

Wu, J., L. Ping, X. Ge, Y. Wang and J. Fu, 2010. Cloud storage as the infrastructure of cloud computing. Proceedings of the International Conference on Intelligent Computing and Cognitive Informatics, June 22-23, 2010, Geneva, Switzerland, pp: 380-383.