Journal of Data Acquisition and Processing
ISSN: 1004-9037
https://sjcjycl.cn/
DOI: 10.5281/zenodo.98549851

CONTEXT AWARE ADAPTIVE SMALL FILE MANAGEMENT FOR HADOOP

Prof. Shwetha K S
Ph. D Research Scholar, Department of Computer Science & Engg., East Point College of Engineering and Technology, Bengaluru, Karnataka, INDIA

Dr. Chandramouli H
Professor, Department of Computer Science & Engg., East Point College of Engineering and Technology, Bengaluru, Karnataka, INDIA

Abstract— Small file processing is a performance bottleneck in Hadoop big data platform. It creates huge storage overhead at Hadoop Namenode and exhausts computational resources by spawning multiple map tasks. Current solutions for small file problem can be categorized to merging small files, reusing java virtual machine instance, caching are not adaptive to user and application context. In most solutions, merging is based only on size and do not consider user access or application contexts, content characteristics and their semantic relation. As the result, processing and query latency increases. This work proposes a user and application context adaptive small file management solution to this problem. As part of solution, adaptive block categorization is proposed to categorize blocks based on context and merged. By this way processing speed is increased. Caching is done based on multi criteria optimization specific to context access pattern to reduce the latency. The proposed solution is able to provide 10% higher speed up and 15% lower latency compared to existing works.

INTRODUCTION

Hadoop is a popular open source big data platform. Based on map reduce paradigm, the platform is designed to process large volume of data. Hadoop distributed file system (HDFS) stores the files. A map task is spawned to process a block. By default each file in HDFS is mapped to a block and passed to map task. Each map task runs on a java virtual machine (JVM) instance. Block to file mapping is kept at the Namenode of Hadoop. Maximum performance in terms of processing and minimal storage overhead at Namenode is achieved when the size of file in HDFS is equal to the block size. The earlier big data applications did not have any problem with this constraint as the file size was higher and almost occupying the block size. But in case of Internet of things (IoT) applications, file size is very small (less than 2KB)[1-2]. Copying these files directly to HDFS and using it for big data processing creates huge store overhead at Namenode and computational crunch due to multiple JVM instances [3]. The current solutions for small file problem can be categorized as merge based, caching based and optimization based. Merge based solutions involve pretreatment of small files to create a large file fitting to block size and uploading the large file to HDFS. Caching based solutions caches the files location to merged block to speed up the query execution. Optimization based solution involves cluster structure reorganization and JVM instance minimization to reduce the computational overhead in spawning map tasks for each small file. Some of the work combined multiple techniques in each category into a hybrid method and used it to solve the small file problem. The existing solutions for small file problem have three important issues (i) non

adaptive merging, (ii) lack of personalization and (iii) lack of support for streaming data. The merging and caching solutions are not adaptive to user and application contexts. As the result, they have higher latency and processing delay for specific applications. Caching solutions do not consider temporal user access patterns and use generic strategies like least recently used (LRU). As the result, their query latency is higher. Most the merging solutions do not support streaming data and does merging only based on size. This works considers two important issues of non adaptive merging and lack of personalization and proposes a novel user and application context aware small file management technique. Small files are categorized into context using machine learning and files are merged based on context. Users are clustered based on access profile and caching is made adaptive to user group and multi objective optimization parameters. The objective of the solution is to reduce the processing delay and query latency. Following are the novel contributions of this work

(i) A novel technique to learn the user and application context for the small files and use the context information to merge files.

(ii) A multi objective optimization based caching technique adaptive to context access patterns.

The paper is organized as follows. Section II presents the survey on existing solutions to solve the small file problem in Hadoop. Section III presents the proposed user and application context aware small file management technique. The results of the proposed solution and comparison to existing works are presented in Section IV. Concluding remarks and the future work scope are presented in Section V.

SURVEY

Ahad et al [4] solved small file problem using a dynamic merging strategy. Next fit allocation policy is used to fit the small file to the most suitable block. Once the block is fully occupied, it is moved a large file to HDFS. Merging was done only based on size without consideration for file contents and their semantic relation. Siddiqui et al [5] replaced default Hadoop Archives (HAR) with a cache based block management scheme for solving the small file problem. A large file of block size is created with logical chain of small files and moved to HDFS. Read/Write on block was made efficient using block manager. Files were merged only based on size without consideration for contents and their semantic relation. Zhai et al [6] solved the small file problem using index based archive file management. Small files are merged to large file to the size of the block. A order preserving hash based indexing system is built to locate the small files efficiently. Though the solution was able to solve the small file problem, access efficiency becomes lower when volume of files increases. Streaming data is also not supported in this work. Cai et al [7] used correlation information between the files to merge the small files to create large file. The large file is kept in HDFS. Authors found that merging files based on correlation reduced the access latency. This work considered correlation only based on meta data and did not use content characteristics. Choi et al [8] solved the small file problem by optimizing java file buffering and JVM reuse. Small files are merged to large file in buffer and the buffer is passed to map task. JVM reuse reduced the JVM bootstrap overhead. Merging did not consider content characteristics and their semantic relation. JVM reuse can create fast memory buildup and cause system crash. Peng et al [9] solved the small file problem by combining merging and caching strategies. Correlation between small files is learnt using collaborative filtering based on the access pattern of files. Files with higher correlation is

merged to a large file and moved to HDFS. Small files to block mapping is kept in caches and this information is cache is used to speed up the access time. The overhead is higher in this scheme for merging for large dynamics in access patterns. Merging was not based on content characteristics and semantic relation between contents in the files. Niazi et al [10] proposed a hybrid storage strategy to solve the small file problem. Default Hadoop archive system is used for large files. To manage small files, data blocks are combined to create large blocks logically. The approach is not scalable as it increases storage overhead at Namenodes for higher volume of files. Jing et al [11] used sentence similarity between the files to learn file correlation. Files with higher correlation are merged to large files. File access was speeded up using pre-fetching. Compute file correlation based on sentence similarity pair wise is cumbersome and not scalable for large number of files. Sharma et al [12] extended the default Hadoop archive system with dual merge technique to solve the small file problem. Two level compaction is proposed to merge small files to large files with higher storage efficiency. Author speeded up file access using two level hash function. Merging was based only size of file and did not consider content characteristics and their semantic relation. Wang et al [13] proposed a integrated solution combining merging and caching to solve small file problem. Files were merged based on size. Access to file was speeded up using pre-fetching and caching. Though the scheme was able to achieve higher storage utilization, content merging did not consider content characteristics as the result, processing delay increased. Ali et al [14] merged small file based type and size using an enhanced best fit merging algorithm. Though the scheme increased storage utilization, the file access time was higher. Prasanna et al [15] created a large file fitting Hadoop block size by compressing many small files. Though storage utilization is increased, computational overhead for compression/decompression is higher in this approach. Huang et al [16] solved the small file problem for medical images using a two level model. At first level images are grouped based on structural similarity and second level based clinical examination similarity. Grouped images are saved as one file in HDFS. File pre-fetching was done to reduce the access time but the file hit ratio was lower. Renner et al [17] proposed a appendable file scheme to solve the Hadoop small file problem. The scheme was implemented as an extension to Hadoop archive file system. Small files are appended to the most suitable block using first fit algorithm. File access was speeded up using Red black tree based indexing. Merging was based only on file size. Liu et al [18] merged small files based on content similarity. A feature vector based on term frequency is constructed for files and similarity between the feature vectors is computed using cosine similarity. Files with higher similarity are merged. Constructing a global feature vector space is difficult without knowledge of distribution of terms in files. Lyu et al [19] solved small file problem using an optimized merging strategy. Optimization was based on maximization of storage utilization of blocks. Authors used caching and pre-fetching to speed up the access. The merging was based only on size without considering content characteristics and their semantic similarity. Similar optimized merging strategy was proposed by Mu et al [20]. Optimized merging strategy was realized with appending based merging over default Hadoop archive system in this work. Merging was based only on file size and access time was higher in this approach. Wang et al [21] merged small files based on user access pattern. But the scheme is not adaptive to user access pattern over a long duration. He et al [22] merged small file to maximize the storage utilization of data blocks using best fit. Merging did not

consider content characteristics and their semantic relations. Fu et al [23] solved the small file problem using a flat storage file system. In this scheme both files and their meta data are merged in same block. This reduces the storage overhead at Namenode. But author did not consider the effect on access time due to flat storage file system. Multi level caches was used in Tao et al [24] and Bok et al [25] to speed the access when small files are merged to large blocks. But merging was still based on size and this increased the processing time for applications requiring co-location of data.

From the survey, most of existing approaches merged small files only based on size and type. Content characteristics and their semantic relation was not considered in any of the works. Without consideration of it, the processing time is very high for applications requiring co-location of related data. Also in most caching schemes, caching was based on least recently without considering the context of the files. Caching on global context can cause poor utilization of applications like clustering which works on multiple contexts at same time. To solve this access time discrepancy context based caching must be employed. This work considers this two problem of content characteristic based merging and context based caching while solving the small file issue in Hadoop and propose a solution for it.

CONTEXT AWARE SMALL FILE MANAGEMENT

The architecture of the proposed user and application Context Aware Small File Management (CASFM) is given in Figure 1. The proposed solution has three functional components: Context management, Metadata management and Cache management. Context is the summary information of the file. It is predicted and association to the

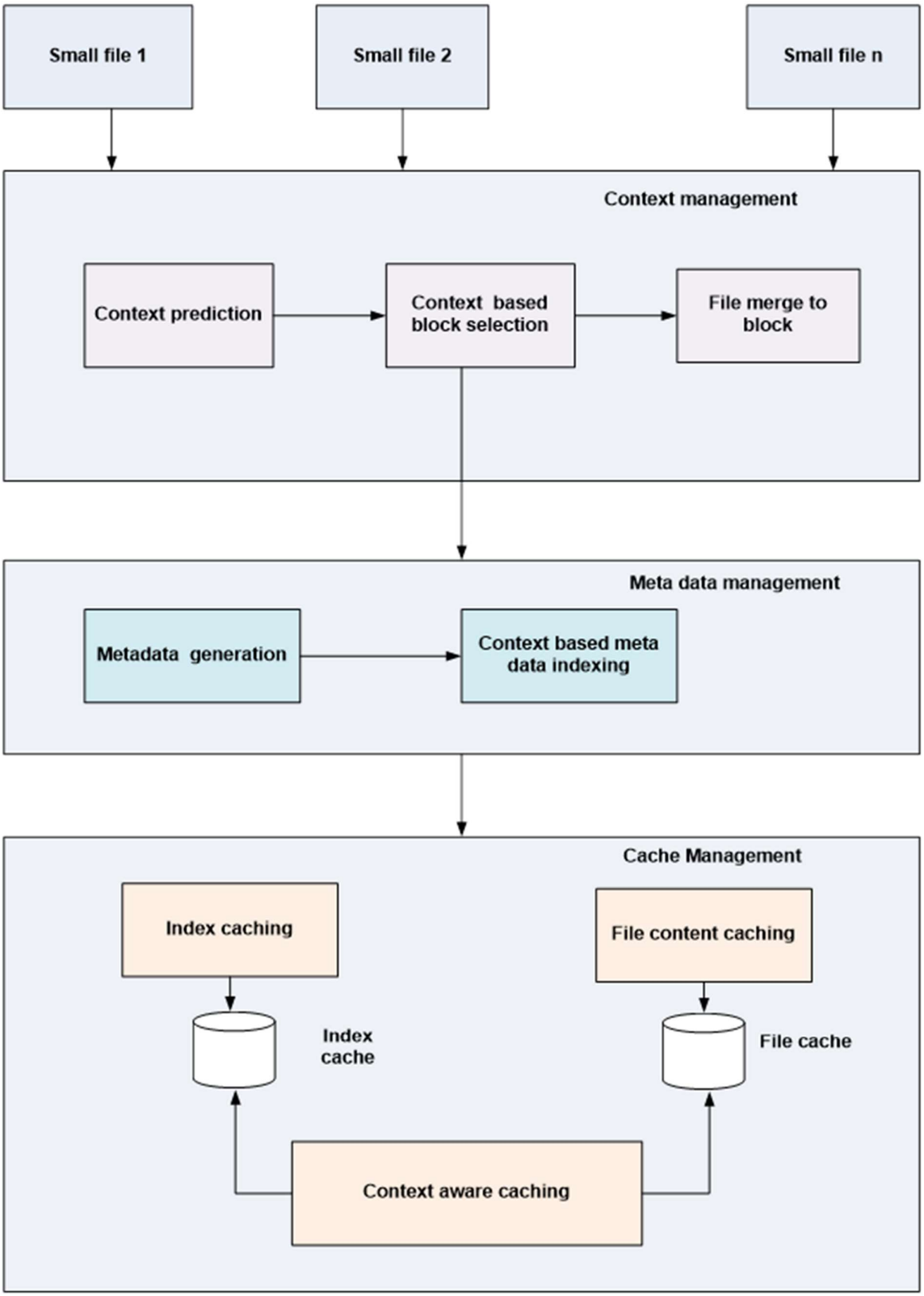


Figure 1 Proposed CSFM architecture showing components of context, meta data and cache management

most suitable blocks is done by context management module. Context is predicted for file based on content characteristics and their semantic relation. Contexts are grouped and aggregation label is created. Blocks are associated with aggregation label and files are appended to the blocks. Metadata is the information about the placement of small files within the block. This is needed for retrieval of files either by name or keyword. Metadata management module manages the metadata information of the small files. Placement of frequently used small file content and metadata indexes in cache speed up the access of small files. Cache management module manages the cache based on context information. The details of each of the functional modules are given below.

Context management

Context is learnt in adaptive manner from small files in this work. Content characteristics and semantic relation are extracted from the small file and a vocabulary is built for contexts. Both the context construction and context association of small files are done on fly. Context is learnt based on the concept modeling concept.. This model estimate the probability distribution of words for a concept (p(w|θ)) and probability of distribution of context for a small file (p(θ|s)). These two probability distributions measures the correlation between word w and concept θ and that between the concept θ and small file s. Both p(w|θ) and p(θ|s)) can be inferred by maximizing the log likelihood function of the observed small file to word matrix. This can be learnt using a batch of small file T as

$$L(T) = \log \left(\prod_{s \epsilon T} \prod_{w \epsilon W} p(w,t)^{n(w,s)}\right)$$

$$= \prod_{s \epsilon T} \prod_{w \epsilon W} n(w,s) log p(w,s)$$

$$= \prod_{s \epsilon T} \prod_{w \epsilon W} n(w,s) \log \left(\sum_{\theta \epsilon \emptyset} p(w|\theta)\, p(\theta|s)\right)$$

Small files up to size of T are collected and salient words are collected from it creating a file to word matrix. The salient words are the nouns and noun phrases from the text skipping the most common stop words from English. Nouns and noun phrases are grouped based on their semantic similarity and group of nouns/noun phrases are created. Context is modeled as a group of nouns/noun phrases The group is referred as context. Every time a new batch of small files arrives, the noun/noun phrases extracted from it are matched to existing contexts. When they are semantically similar to existing context, the noun/noun phrases are merged to context and when they are different, a new context is created. For each context, a block is created. Small files classified to a context are appended to the corresponding block. When block is full, a new block is created. A context index is created with a unique id for context, the noun/noun phrases in the context and blocks belonging to that context.

Metadata management

Meta data is information about the location of the small file in the block. This is needed for retrieval of the file for any queries by the user/application. Meta data has fields as listed in **Table 1.**

Table 1Metadata fields

Fields	Size (bytes)

File name	128
Block number	16
Offset	16
File size	16

Different from existing works of keeping the metadata in a index file, in this work meta data is organized in terms of contexts as shown in Figure 2.

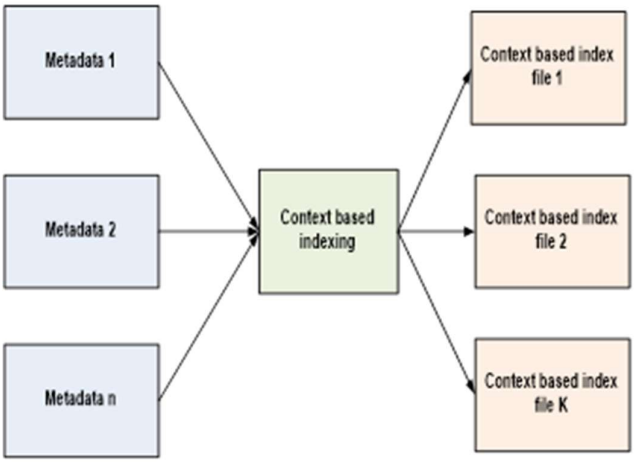


Figure 2 Metadata indexing

For each context an index file is created and the metadata of file categorized to a context is kept in its corresponding index file. By this way, index file is generated for each context. Locality sensitive hashing(LSH) is used for indexing the metadata. LSH is a method for approximate neighbor search in high dimensional space. It maps the high dimensional data to lower dimensional representation using random hash function such a way that points closer in higher dimensional space maps to same low dimensional space with higher probability. LSH hashes the item repeatedly several times, so that similar items are more likely hashed to same bucket than dissimilar items as shown in Figure 1. Thus to find the items in a database, which is similar to a query, LSH maps to most relevant bucket and number of buckets are also less. Due to this lookup becomes faster in LSH compared to hashing based lookup.

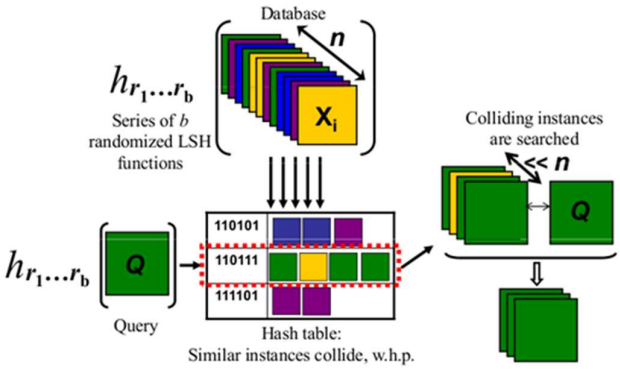


Figure 3 LSH lookup

The idea of LSH is to construct hash functions g: $R^d \rightarrow U$ such that for any two points p, q

if $||p - q|| \leq r$, then $Pr[g(p) = g(q)]$ is high

if $||p - q|| > cr$, then $Pr[g(p) = g(q)]$ is small

This is achieved with family H of functions

$g(p) =< h_1(p), h_2(p), \ldots . h_k(p) >$

For all data point pϵP, p is hashed to buckets $g_1(p), g_2(p), \ldots g_b(p)$

For an input query q, the points are retrieved from the buckets $g_1(q), g_2(q)$,… until all points from b buckets are retrieved.

The effectiveness of LSH comes from use of multiple hash functions instead of single hash. Multiple hash reduces the number of buckets needed for mapping the items in the database

The metadata mapping to a context is indexed using LSH to generate the buckets. The buckets corresponding to the context are written to a index file.

When user/application queries for the small file, they can query the file by name or keywords. When the user queries by file name, parallel search is launched on each of index file using LSH. The bucket matching the filename is returned. From the bucket linear search is done with filename as key to return the metadata. When the query is done with keyword, linear search is done over the context file to find the matching context at first step and search is launched the index file corresponding to that context using LSH. The bucket matching the filename is returned. From the bucket linear search is done with filename as key to return the metadata.

Cache management

Cache is used in this work to speed up the access of small files. Two kinds of cache are used in this work: File cache (FC) and Index cache(IC). FC is used for pre-fetching small files to speed up the access of small file. IC is used for pre-fetching index files to speed the processing of fetching small files. The proposed cache management solution splits the FC cache into K partitions with each belonging to a context. The size to be allocated for each partition is decided based on multi criteria optimization. In the size allocated, the small files to be cached are decided based on the access patterns of the user group/application. The cache size to be allocated for a context is depended on following factors

(i) Access delay for the context (p1)

(ii) Hit ratio for the context (p2)

(iii) Speed up of application (p3)

The access delay must be minimized; hit ratio and speed up must be maximized. Minimizing access delay very much for one context may reduced the speed up of total application by affecting other contexts and it can also reduce the hit ratio of other contexts. Thus these three factors are conflicting goals. A fitness function is designed to accommodate this three conflicting goals as

$$f = w_1 \frac{1}{p1} + w_2\, p2 + w_3 p3$$

The weights w_1, w_2, w_3 are the importance factors given to the parameters. Their values are assigned depending on the applications in such a way that

$$w_1 + w_2 + w_3 = 1$$

The cache size to be allocated for each context must be optimized to maximize the fitness function (f). This work uses a hybrid meta heuristics algorithm combining particle swarm optimization (PSO) with firefly algorithm.

PSO is a swarm intelligence algorithm (Kennedy et al 1995) simulating the social behavior of swarm of organisms. This method is popular for solving optimization problems due it its simplicity, flexibility and versatility. Organisms move randomly with different velocities and use these velocities to update their individual position. Each candidate solution is a 'particle'. Each particle tries to attain its best velocity based on its own local best (p_best) value and its neighbor's global best (g_best). Each particle's next position depends on the current position, current velocity, distance from current position to p_best, distance from current position to g_best. The movement of particle in its search space depends on its velocity. For a particle X, it current position X_i and current velocity V_i is updated as

$$X_i(t+1) = X_i(t) + V_i(t+1)$$

$$V_i(t+1) = wV_i(t) + c_1 r_1\big(p_{besti}(t) - X_i(t)\big) + c_2 r_2\big(g_{besti}(t) - X_i(t)\big)$$

In the above equations, t is the iterative value. c_1 and c_2 are acceleration coefficients, . r_1 and r_2 are random numbers, w is the inertia weight. The iteration is repeated till termination condition is met.

Firefly is swarm intelligence algorithm (Yang et al 2008) based on the behavior of fireflies in naturally occurring environment. Fireflies exhibit unique light flashes for various purposes like mating, warning about potential danger etc. Fireflies operation is guided by two parameters: light intensity and level of attractiveness. Light intensity (I) is inversely proportional to the distance between the emitting and observing firefly (r). It is given as

$$I = \frac{1}{r^2}$$

Level of attractiveness is proportional to the light intensity. It is calculated as

$$\beta(r) = \beta_0\, e^{-\gamma r^2}$$

β_0 is the attractiveness at r=0 and γ is the light absorption coefficient. Euclidean distance formula is used for calculating r.

The movement of firefly (FX_i) governed by attraction from another firefly (FX_j) is calculated as

$$FX_i = FX_i + \beta_0 e^{-\gamma r^2}(FX_j - FX_i) + \alpha\epsilon_i$$

In the above equation α is the randomization parameter and ϵ_i is a random number.

Firefly algorithm has strong exploitation ability while PSO has great diversification ability. This by combining optimization algorithms with complement properties of strong exploitation and diversification, a near optimal solution can be obtained with a faster convergence rate. For this reason, PSO was hybridized with Firefly algorithm in this work. Once in fixed time interval, the process of determination of optimal cache size for each context is started. An M array of size K is created with each element in array is value between 0 and 1. The sum of all values in the array must be 1. The initial values of the array are filled with random numbers..

Firefly algorithm starts with the M arrays and finds the initial solution. PSO takes this initial solution and finds the optimal solution. The optimal solution is array of K size with each of value between 0 and 1. The size of the cache to be allocated is calculated by multiplying the value for the context in the array by the total size of the cache.

Once the size of the cache for each context is found, the FC is filled with entries using least recently used algorithm (LRU). For the files decided to be cached in the FC size, the corresponding metadata information is kept in the IC to speed up the access.

RESULTS

The performance of the proposed solution is tested against experimental setup consisting of 6 nodes with 1 Namenode and 5 Data node. The configuration of each node is as follows

Table 2 Node configuration

Parameter	Value
CPU	2 core with 2.13 GHZ
RAM	8 GB
Disk	500 GB
OS	Ubuntu
Hadoop version	2.9.1
Number of replicas	3
HDFS block size	512 MB

Small text dataset of 100000,200000,300000 and 400000 files with file size from 1KB to 10 MB is used for experimentation. The performance of the proposed solution is tested against Hadoop perfect file (HPF) [6], Hadoop archive system (HAR) and Map file system.

The access performance of 100 different files spread across 5 different contexts is measured and the results are given in Figure 4.

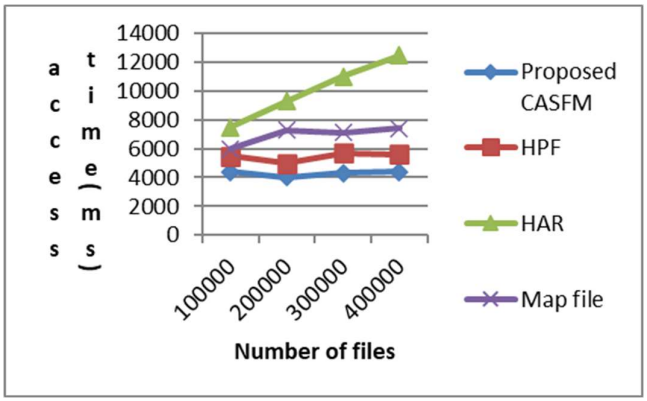


Figure 4 Comparison of access time

The access time in proposed solution is 27% lower compared to HPF, 62% lower compared to Map file and 135% lower compared to HAR. The access time has reduced in the proposed solution due to use of context based caching and context based merging in proposed solution. As result of context based caching, when file is retrieved, it related files by context in same block are also cached. This reduced the file access time for those relevant files.

The application speedup was measured for K means clustering with K=4 and the results are given in Figure 5.

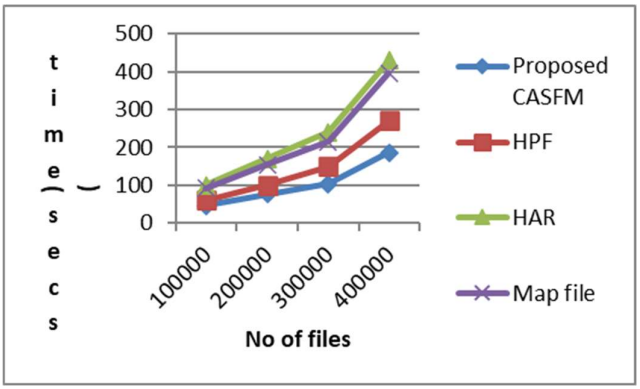


Figure 5 Comparison of application speedup

The application speedup in the proposed solution is atleast 39% higher compared to existing works. The application speedup for clustering has increased due to co-location of highly related files by context in nearby locations in the proposed solution. Due to this collocation, the distance computation operations in clustering have lower time complexity.

The cache hit ratio is measured during the clustering operation and the results are given in Figure 6.

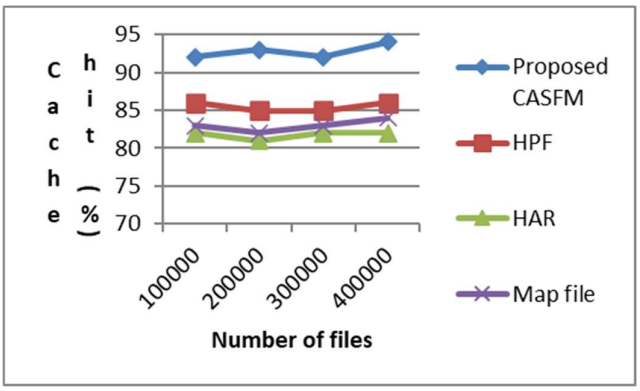
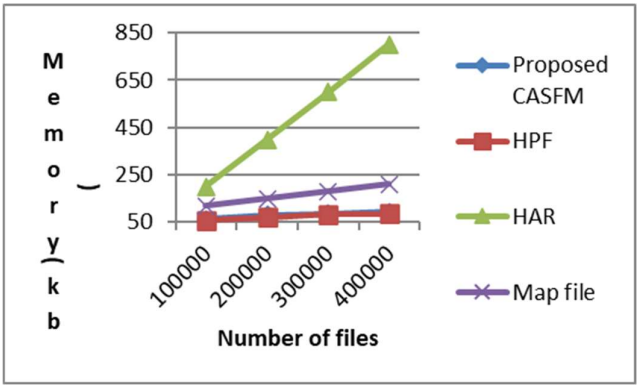


Figure 6 Comparison of cache hit ratio

The cache hit ratio in proposed solution is 7% higher compared to existing works. Organizing the cache based on context and size allocation using optimization function maximizing hit ratio has increased the cache hit ratio in the proposed solution.

The Namenode's memory consumption is measured and the result is given in Figure 7.



The Namenode memory consumption in proposed solution is slightly higher compared to HPF but it is significantly lower compared to HAR. The higher consumption compared to HPF is due to organization of blocks by contexts in proposed solution. This increases the number of blocks.

CONCLUSION

A context aware small file management scheme was proposed in this work. Context information was learnt on fly from small files and files were merged based on context. In addition, cache size were allocated to context based on multi criteria optimization. The proposed solution has increased the application speed up by atleast 39% and reduced the access delay by 27% compared to existing works. Extending the solution for streaming data is in the scope of the future work.

REFERENCES

- Small size problem in Hadoop: http://blog.cloudera.com/blog/2009/02/the-small-files-problem/
- Solving Small size problem in Hadoop https://pastiaro.wordpress.com/2013/06/05/solving-the-small-files-problem-in-apache-hadoop-appending-and-merging-in-hdfs/
- Bo Dong , Qinghua Zheng, Feng Tian , Kuo-Ming Chao , Rui Ma, Rachid Anane.(2012), An optimized approach for storing and accessing small files on cloud storage, Journal of Network and Computer Applications, 35 (2012) 1847-1862, Elsevier
- Ahad, Mohd & Biswas, Ranjit. (2018). Dynamic Merging based Small File Storage (DM-SFS) Architecture for Efficiently Storing Small Size Files in Hadoop. Procedia Computer Science. 132. 1626-1635. 10.1016/j.procs.2018.05.128.
- Siddiqui, Isma & Qureshi, Nawab Muhammad Faseeh & Chowdhry, Bhawani & Uqaili, Mohammad. (2020). Pseudo-Cache-Based IoT Small Files Management Framework in HDFS Cluster. Wireless Personal Communications. 113. 10.1007/s11277-020-07312-3.
- Zhai, Yanlong & Tchaye-Kondi, Jude & Lin, Kwei-Jay & Zhu, Liehuang & Tao, Wenjun & Du, Xiaojiang & Guizani, Mohsen. (2021). Hadoop Perfect File: A fast and memory-efficient metadata access archive file to face small files problem in HDFS. Journal of Parallel and Distributed Computing. 156. 10.1016/j.jpdc.2021.05.011.
- Cai, Xun & Chen, Cai & Liang, Yi. (2018). An optimization strategy of massive small files storage based on HDFS. 10.2991/jiaet-18.2018.40.

- Choi, C., Choi, C., Choi, J. et al. Improved performance optimization for massive small files in cloud computing environment. Ann Oper Res 265, 305–317 (2018)
- Peng, Jian-feng & Wei, Wen-guo & Zhao, Hui-min & Dai, Qing-yun & Xie, Gui-yuan & Cai, Jun & He, Ke-jing. (2018). Hadoop Massive Small File Merging Technology Based on Visiting Hot-Spot and Associated File Optimization: 9th International Conference, BICS 2018, Xi'an, China, July 7-8, 2018, Proceedings. 10.1007/978-3-030-00563-4_50.
- S. Niazi, M. Ronström, S. Haridi, and J. Dowling, 'Size Matters : Improving the Performance of Small Files in Hadoop', presented at the Middleware'18. ACM, Rennes, France, 2018, p. 14.
- Jing, Weipeng & Tong, Danyu & Chen, GuangSheng & Zhao, Chuanyu & Zhu, LiangKuan. (2018). An optimized method of HDFS for massive small files storage. Computer Science and Information Systems. 15. 21-21. 10.2298/CSIS171015021J.
- V. S. Sharma, A. Afthanorhan, N. C. Barwar, S. Singh and H. Malik, "A Dynamic Repository Approach for Small File Management With Fast Access Time on Hadoop Cluster: Hash Based Extended Hadoop Archive," in IEEE Access, vol. 10, pp. 36856-36867, 2022
- K. Wang, Y. Yang, X. Qiu and Z. Gao, "MOSM: An approach for efficient storing massive small files on Hadoop," 2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA), Beijing, China, 2017, pp. 397-401
- Ali, N. M. Mirza and M. K. Ishak, "Enhanced best fit algorithm for merging small files," Computer Systems Science and Engineering, vol. 46, no.1, pp. 913–928, 2023.
- L. Prasanna. Kumar, "Optimization Scheme for Storing and Accessing Huge Number of Small Files on HADOOP Distributed File System". International Journal on Recent and Innovation Trends in Computing and Communication, vol. 4, no. 2, Feb. 2016, pp. 315-9
- Xin Huang, Wenlong Yi, Jiwei Wang, Zhijian Xu, "Hadoop-Based Medical Image Storage and Access Method for Examination Series", Mathematical Problems in Engineering, vol. 2021, Article ID 5525009, 10 pages, 2021.
- Thomas Renner, Johannes Müller, Lauritz Thamsen, and Odej Kao. 2017. Addressing Hadoop's Small File Problem With an Appendable Archive File Format. In Proceedings of the Computing Frontiers Conference (CF'17). Association for Computing Machinery, New York, NY, USA, 367–372.
- Liu, Jun. (2019). Storage-Optimization Method for Massive Small Files of Agricultural Resources Based on Hadoop. Journal of Advanced Computational Intelligence and Intelligent Informatics. 23. 634-640. 10.20965/jaciii.2019.p0634.
- Y. Lyu, X. Fan, and K. Liu, "An optimized strategy for small files storing and accessing in HDFS," in Proc. IEEE Int. Conf. CSE, IEEE Int. Conf. EUC, Jul. 2017, pp. 611_614.
- Q. Mu,Y. Jia, and B. Luo, ``The optimization scheme research of small files storage based on HDFS," in Proc. 8th Int. Symp. Comput. Intell. Design, Dec. 2015, pp. 431_434.
- T. Wang, S. Yao, Z. Xu, L. Xiong, X. Gu, and X. Yang, ``An effective strategy for improving small _le problem in distributed file system," in Proc. 2nd Int. Conf. Inf. Sci. Control Eng., Apr. 2015, pp. 122_126
- H. He, Z. Du, W. Zhang, and A. Chen, ``Optimization strategy of Hadoop small _le storage for big data in healthcare," J. Supercomput., vol. 72, no. 10, pp. 3696_3707, Aug. 2016

- S. Fu, L. He, C. Huang, X. Liao, and K. Li, ``Performance optimization for managing massive numbers of small files in distributed file systems," IEEE Trans. Parallel Distrib. Syst., vol. 26, no. 12, pp. 3433_3448, Dec. 2015
- W. Tao, Y. Zhai, and J. Tchaye-Kondi, “LHF: A new archive based approach to accelerate massive small _les access performance in HDFS”, in Proc. 5th IEEE Int. Conf. Big Data Service Appl., Apr. 2019, pp. 40_48.
- K. Bok, H. Oh, J. Lim, Y. Pae, H. Choi, B. Lee, and J. Yoo, ``An efficient distributed caching for accessing small files in HDFS," Cluster Comput.,vol. 20, no. 4, pp. 3579_3592, Dec. 2017.