Replica Parallelism to Utilize the Granularity of Data

1st Author
1st author's affiliation
1st line of address
2nd line of address
Telephone number, incl. country code
1st author's E-mail address

2nd Author
2nd author's affiliation
1st line of address
2nd line of address
Telephone number, incl. country code
2nd E-mail

3rd Author
3rd author's affiliation
1st line of address
2nd line of address
Telephone number, incl. country code
3rd E-mail

ABSTRACT

As the volume of relational data is increased significantly, big data technologies have been noticed for recent years. Hadoop File System (HDFS) [14] is a basis of several big data systems and enables large data sets to be stored across the big data environment which is composed of many computers. HDFS divides large data into several blocks and each block is distributed and stored in a computer. To support reliability of data, HDFS replicates the data block. Generally, HDFS provides high-throughput when a client accesses to data. However, the architecture of HDFS is mainly designed to process data whose pattern is large and sequential. A data input whose pattern is small and random is not appropriate for applying HDFS. The data can result in several weak points of HDFS in terms of performance. HBase [2], one of Hadoop eco-systems, is a distributed data store which can process random, small read/write data efficiently. HBase utilizes HDFS structure but the block size of HBase is smaller than one of HDFS and a file of HBase is composed by blocks which is arranged by index structure. As many softwares which are related with big data science have emerged, many researches improving the system performance also have emerged steadily.

Because a read workload is dominant in most big data workloads, read performance in the big data systems is crucial. To improve the read performance of HBase, we propose a replica parallelism algorithm utilizing the fact that block size of HDFS and HBase are different. When a read I/O is requested, HDFS generally selects a best node which is closest to client and I/O tend to converge on the specific node. We suggest distributing I/O jobs to replicas of HDFS based on sub-region of file to reduce a seek time and utilize the full bandwidth of all replicas. We analyzed the performance results of original Hbase using YCSB benchmark known as a well-known benchmark for distributed data store. In additional, we predict the performance of the replica parallelism comparing with original Hbase. According to predicted result of experiment, the replica parallelism can improve the read performance up to 1.5x ~ 2.5x comparing with the conventional Hbase. Additionally, highly random read requests can intensify the effectiveness of the replica parallelism.

DOI: http://dx.doi.org/10.1145/12345.67890

CCS Concepts

• Information systems → Database management system engines→Parallel and distributed DBMSs

Keywords

Big data platform; Distributed store; HBase; Storage technology; Hadoop

1. INTRODUCTION

As the extremely large and complex data emerges, traditional relational database management systems are inadequate for processing the big data in terms of performance and scalability. For this reason, big data science which includes the analyzing meaningful results from data and system engineering with big data platform has been noticed by computer science researchers in recent years. Hadoop supports a basic framework to process the big data by providing distributed file system (HDFS) that can split the large data set and scatter the small units of the data to cluster environment. Hadoop also provides a data processing model to handle the data efficiently, which is called MapReduce. Emergence of Hadoop stimulates the various research about big data science.

Especially, many researchers focus on improving overall performance of the big data platform. Because the existing HDFS and MapReduce are only designed for batch processing, optimization for various data type is relatively limited. Moreover, although the overall performance of big data processing is significantly influenced by I/O performance related with physical storage, lots of research subjects for improving storage I/O performance have been remained.

Many Hadoop eco-systems or new big data paradigms have emerged to solve the above problem. For example, Spark [3] and Tajo [7] provides new big data query engines to overcome the limits of existing data processing model. Originally, MapReduce [8] generates additional storage I/O through lots of computational iterations. The new query engines tend to utilize a main memory to mitigate the overhead of I/O and reduce the overhead from the computational iterations by generating optimized query plans. In addition, as conventional HDFS is not appropriate to cope with data requests whose size are relatively small, HDFS needs eco-systems to cover with above problem. HBase is a distributed data store which enables to handle the small and random I/O effectively providing the blocks whose sizes are smaller than one of HDFS and a block index structure to search the data block efficiently.

Hadoop is also about to focus on applying storage technologies to the conventional HDFS architecture. Heterogeneous HDFS supports utilizing different type of storage to boost the overall performance. [1] It is possible for the hottest data to be stored in the storage which has low read/write latency such as Solid States Drives (SSDs). Similarly, the research about constructing the cache layer composed of Hard Disk Drives (HDD) and SSDs has been progressed [15]. Though SSDs have higher performance than HDD, price of SSDs is much expensive and capacity of SSDs is smaller than HDD. Therefore, deploying data blocks appropriately is important to achieve the optimal performance per price.

In this paper, we target HBase architecture which is generally utilized with various big data engines. We focus on I/O type of HBase and propose the replica parallelism to improve the random read performance of HBase.

The paper is organized as follows. In Section 2, we review previous researches about big data techniques. In Section 3, we propose a replica parallelism and explain its algorithm of implementations. In Section 4, we evaluate the performance of proposed structure with YCSB benchmarks. Finally, we conclude in Section 5.

2. RELATED WORK

2.1 Hadoop Distributed File System (HDFS)

Hadoop is an open source software that provides a basic framework to store and process the large data set through cluster environment. Hadoop is consisted of two parts, MapReduce and Hadoop Distributed File System (HDFS). MapReduce is a data processing model which handles the large data set with parallel and distributed algorithm. HDFS is a data storage model which enables to store the large data files. We focus on the original HDFS structure, because it is linked closely to storage I/O performance.

Figure 1 illustrates the overall architecture of HDFS. HDFS is composed of two types of nodes. One is a NameNode. The other is a DataNode. In the general HDFS environment, a single master cluster is in charge of NameNode and several slave clusters serve DataNode. Hadoop divides the large data file by the unit of block whose default size is 64MB and scatters the blocks to DataNodes. NameNode stores overall file system metadata, location of the blocks in cluster environment and additional information of blocks. NameNode manages the information in main memory to improve the response performance. Journaling is responsible for metadata reliability, information of block location is also received from DataNodes periodically. When a client requests read or write I/O, the client accesses to NameNode firstly and NameNode returns the information about data blocks.

DataNode stores the data block and transfer the block information to NameNode periodically. To retain data scalability and reliability, HDFS makes several copies of a block, which is called replication. HDFS supports three block replicas basically. The block replicas are distributed to the different DataNodes considering the locality of data.

When a client requests write I/O, HDFS splits the data file by the unit of block and replicates the block to three blocks. NameNode determines the location of each blocks. Generally, two of replicated blocks is stored in two DataNodes in same rack and one is stored in other DataNode in other rack. Although NameNode manages all meta information about blocks, NameNode is unrelated with actual data access and the client only accesses directly to DataNodes to process the practical write operation.

When a client requests read I/O, the client accesses to NameNode and NameNode returns the best block location whose block is closest to the client in terms of network distance among three replicas. After the client receives the block location, the client accesses to the DataNode which has the data block.

HDFS architecture is originally implemented for batch processing whose I/O pattern is sequential and large chunks of data. For this reason, the default block size of HDFS is 64MB and 128MB block is also used in real workload to optimize the performance.

However, the feature of HDFS that block size of HDFS is large enough to generate additional overheads in spite of small size input is the reason that cannot manage the small and random type of data effectively. If a client requests the small size data whose size is under the block size of HDFS, HDFS can face a crucial performance issue.

2.2 Heterogeneous HDFS

Because conventional HDFS recognizes different types of storages as only disk type, HDFS cannot fully utilize the storage which has better performance than HDD, such as SSDs and RAM disk. However, current released version of HDFS supports an archival storage API. HDFS also stores information about storage type. With the storage type information, HDFS can store a data blocks to desired storages with various placement policies. For instance, using ONE_SSD policy, we can store one data block in SSD and two data blocks in HDD among three data replica blocks. Figure 2 illustrates the above example.

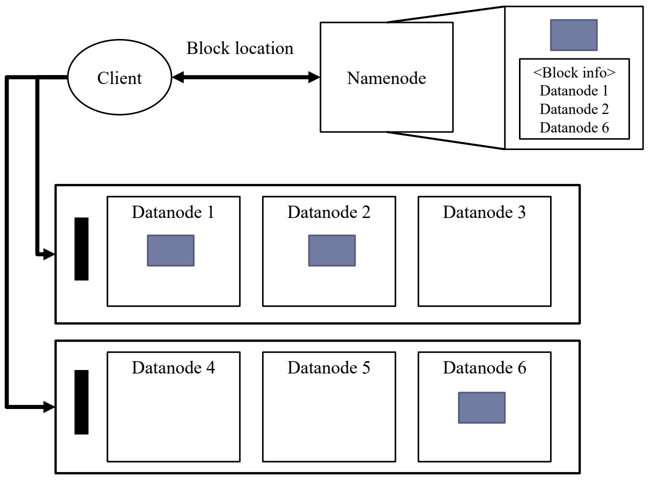

Figure 1. Overview of HDFS

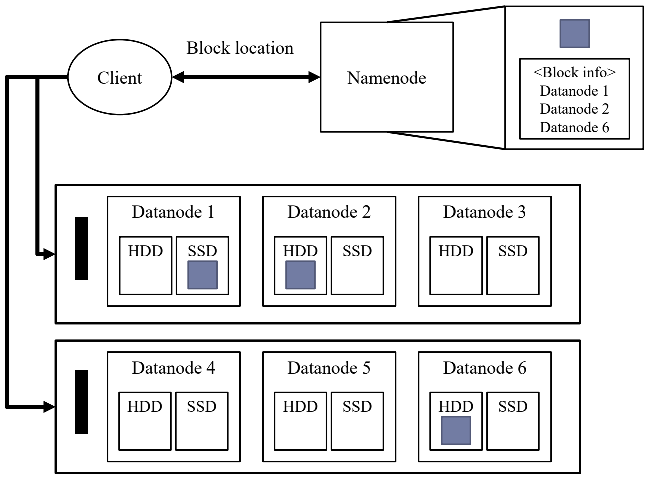

Figure 2. Heterogeneous HDFS with ONE_SSD policy

2.3 HBase

HBase is an open-source software that provides a distributed data store which can store data in formation of key-value. HBase is also Hadoop eco-system which is based on Hadoop architecture and utilize HDFS structure.

Figure 3 illustrates the overall architecture of HBase. HBase is consisted of three components, Zookeeper [5], Hmaster, Regionserver. Zookeeper is a software that checks the status of the clusters and manages metadata of HBase files.

Hmaster also manages tables of HBase and is generally placed on a master node with NameNode. When a client of HBase requests put or get operations, Hmaster cooperates with Zookeeper in returning the information about location of required data. In addition, Hmaster also manages load balancing of HBase structure. If data stored in HBase is skewed seriously, Hmaster determines to rearrange the location of data.

For data scalability, HBase splits HBase table to several regions by key range. Regionserver serves to manage data items in a specific region. Because Hbase is one of HDFS clients, regionservers are generally placed with DataNodes. A regionserver has several data structures to optimize the storage I/O. The optimization also can handle random and small storage I/O. A regionserver is consisted of BlockCache, Memstore, Hfile, Write Ahead Log (WAL) structures.

HBase utilizes the memory to boost overall performance. BlockCache is one of the examples. BlockCache is used as read cache. Because the hottest data which is accessed by clients frequently has high possibility in being stored in BlockCache, the client checks BlockCache firstly to fetch the data. Basically HBase tends to set a data item that is recently accessed to low latency storage such as memory. Memstore is a write buffer of HBase to utilize sequential pattern I/O. The default size of Memstore is 128MB. When HBase receives the write operations, HBase never writes data to the storage directly. Firstly, data is stored in memstore, once memstore is full, data is flushed to the storage. The process enables HBase to handle the small data effectively.

Because data can be stored in the memory, Write Ahead Log (WAL) handler writes log data which indicate the information of data in storage to prevent data loss. If the data in the memtable is vanished due to system failure, the data can be recovered using WAL. As the WAL structure is stored on HDFS, it can be an important factor which affects the system performance.

Hfile is a storage format of HBase. Figure 4 illustrates the Hfile structure. Key-value pairs are arranged by the key order in a data block of Hfile. Block size of Hfile is 64KB and blocks of Hfile are arranged based on index structure which is constructed similar to B-tree index structure to support small and random data requests. As the input data is increased, the number of Hfile is also increased. To improve read efficiency, HBase provides compaction operation which merges several Hfiles into one Hfile. There are two type of compactions, minor compaction and major compaction. Minor compaction is merging several files into one file to improve the read performance by reducing overhead from opening files. Major compaction is merging all Hfiles into the large Hfile to support delete operation and improve the read performance. While compaction is performed to improve the read performance, it can generate serious write overhead.

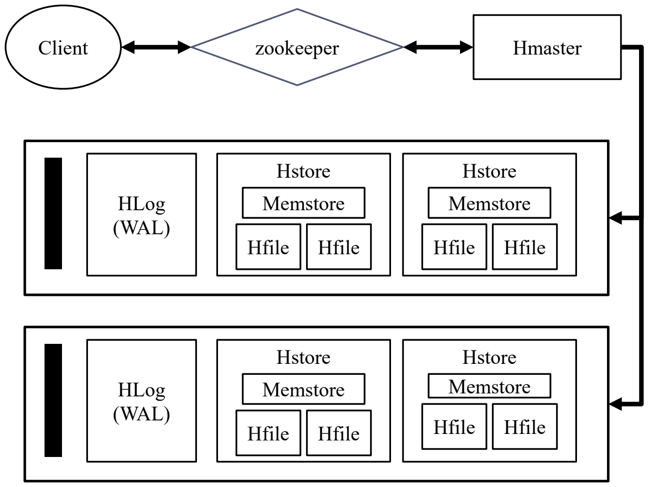

Figure 3. Overview of HBase

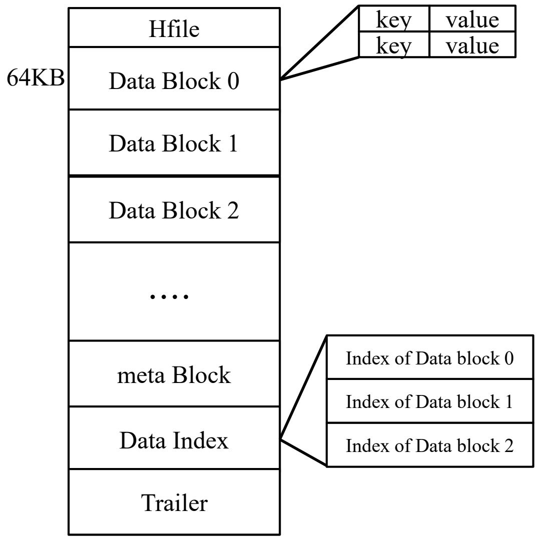

Figure 4. Hfile Structure

When HBase receives a write request, a client receives location information of data from Hmaster. The client finds the appropriate regionserver, write the WAL and then write the data into the memstore. The data is finally flushed to the storage as the format of Hfile. Hfile and WAL structure transfer write requests to HDFS.

When HBase receives a read request, a client checks BlockCache firstly. If the data is not stored in BlockCache, the client checks memstore, Hfile in order. If the data is stored in Hfile, read requests are transferred to HDFS.

Originally HBase uses HDFS to store the data though, HDFS cannot knows Hfile information because HBase merely uses function of HDFS in imported libraries. In aspect of HDFS, Hfile is only binary data not to be recognized.

3. Replica Parallelism

3.1 Overview

HBase is operated on HDFS basically and Hfile is stored on HDFS. However, HBase isn't concerned about detailed HDFS structure or operations of distributed file system. HBase only utilizes HDFS through HDFS APIs. Although HBase is designed for random and small data input and block size of HBase is much smaller than one of HDFS, HBase cannot communicate with HDFS about difference of granularity of block for optimizing storage performance.

While HDFS makes multiple copies of data block, we cannot utilize bandwidth of replicas fully because NameNode generally tends to return one block location among three replicas which is closest to client. It can lead to skewing I/O requests in one DataNode and we cannot utilize the rest of replicated node's storage resources. Additionally, in case that DataNodes are composed of HDD, skewing I/O requests in one storage results in generating unnecessary seek operations which influence the seek time of HDD and overall access time. We propose a replica parallelism algorithm to distribute the read requests in one HDFS block to three replicas with referencing Hfile structure. Generally, parallelism on different data blocks is a natural situation in Hadoop environment. However, distributing random I/Os in a HDFS block to three replicas is a novel algorithm. When multiple threads request random read I/Os, we design the replica parallelism to distribute the I/Os across replicas utilizing full bandwidth of storages in all replicas.

Figure 5 illustrates the difference of block granularity between HBase and HDFS. Generally, HDFS block size is 1000x~2000x bigger than HBase's block size. It explains that possibility of generating lots of random I/Os in one HDFS block is feasible. It is true that the replica parallelism also generates additional network cost. However, performance of specific workloads such as Facebook message [10] which have highly random and small data input is more influenced by seek time than network cost. In additional, network cost will be reduced with the environment which has high bandwidth network like 10 Gigabit ethernet.

When multiple read requests in a HDFS block, first we divide the Hfile by the unit of HDFS block and then each unit is also divided into sub-regions. If a specific key of read request is required to process, our function returns the sub-region where the key is stored. HDFS assigns the sub-regions to each replica and the request for the key accesses to assigned replica in order to distribute the read quests. Figure 6 explains an example of the replica parallelism. Replicated HDFS blocks are stored in DataNode 1, DataNode 3 and DataNode 6. Read requests for keys which is stored in Data Block0 are processed in DataNode 1. In the same manner, read requests for keys which is stored in Data Block 1, Data Block 2 are processed in DataNode 3 and DataNode 6, respectively.

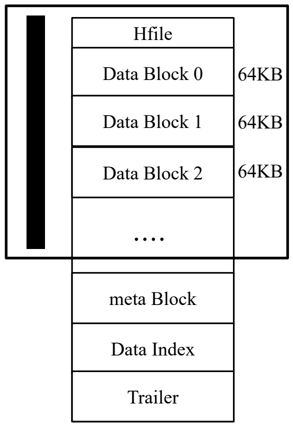

Figure 5. Granularity of HBase and HDFS

3.2 Design

We first design the schema of interface between HDFS and HBase to transfer information of Hfile. Figure 7 explain overall algorithm of replica parallelism. HBase fetches the block size of HDFS and split the Hfile structure by the unit of HDFS block. Next, the unit is also divided into several sub-regions by key range. When HBase receives the read requests, implemented function determines sub-region in which desired key is stored. Transferring the sub-region information to HDFS enables Hbase to scatter the read I/Os across the three replicas The reason for splitting block by key range is to reduce the block seek time to narrow the key range.

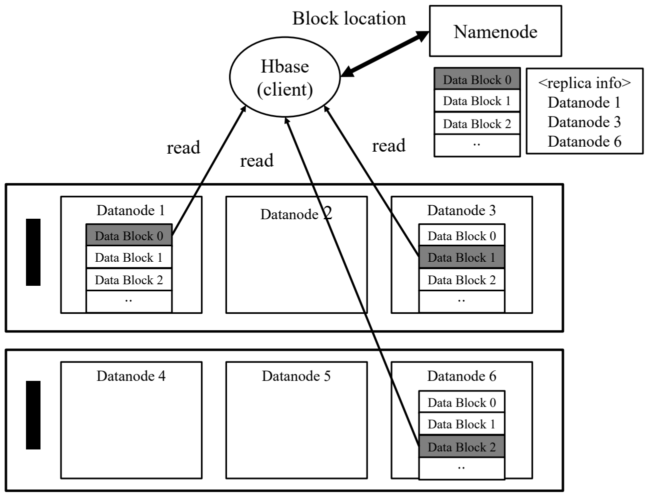

Figure 6. Replica parallelism

Algorithm 1, Algorithm 2, Algorithm explains the design of replica parallelism concretely. Algorithm 1 shows Hbase's original function which seeks a key requested from client. getDataBlockIndexReader function in Line 1 load the information about index block in Hfile. With this information, Hbase call loadDataBlockWithScanInfo function to return the block which has the key which a client requests. Next, in Line 3, loadBlockAndSeekToKey function try to find the exact key entry in a Hbase block.

Algorithm 1 HFileReader.seekTo(Cell key, boolean rewind)

```
    Description
    Seek a desired key in a Hfile

    Input
    Cell key : key to search
    Boolean rewind : variable used in search oepration

    Output
    status after key search

    /* Call the indexReader of Hfile */
1   indexReader = getDataBlockIndexReader();

    /* Through binary search in index structure, return the
    block which has key */
2   blockWithScanInfo =
    indexReader.loadDataBlockWithScanInfo();

    /* Search the key in a Hfile block */
3   retrun loadBlockAndSeekToKey()
4   End
```

Algorithm 2 explains about loadDataBlockWithScanInfo function, which includes the fundamental design parts. The function receives the key information to find, block information, called currentBlock, which is stored in memory and so on. First, the function loads the rootLevelIndex information to memory (Line 1)

Hbase checks the block whether it has the desired key during the binary search operation. Our proposed idea calculates the HDFS block size (Line 3), and identify the sub-region where the key is located. Then RepNum meaning the replica id is returned by CalculateKeyRange function. The function determines which DataNode should return the block corresponding to the key.

Algorithm 2 loadDataBlockWithScanInfo(Cell Key, HfileBlock currentBlock, boolean pread, …)

```
    Description
    Find a specific Hfile block which has the desired key using
    index block.

    Input
    Cell key : key to search
    HfileBlock currentBlock : a current block
    Boolean pread : a variable which presents data pattern

    Output
    status after key search

    /* Search and cache the block using binary search with
    Index blocks */
1   NextIndexedKey = rootLevelIndex;
2   While( Binary Search) {
3   /* Proposed desgin part*/
    HdfsBlockSize = GetHdfsBlockSize();
    RepNum = CalculateKeyRnage(NextIndexedKey);
4   /* Read the block from HDFS and cache it*/
    block = readBlock(RepNum);
5   if ( block has the desired key )
      break;
6   /* line 9-10 related with binary search operation*/
    Index = locateNonRootIndexEntry();
    NextIndexedKey = getNonRootIndexedKey(Index);

    }
7   End
```

Algorithm 3 explains Modfied_getBestNodeDNAddrPair function in HDFS code. When Hbase calls the interface of HDFS, the function is called finally. The function determines and returns the best replica suited to a client request. In Line 1, getLocation function fetches DataNode information from Namenode and returns a list of DataNode replicas, and list is ordered by network distance order generally. RepNum calculated in Hbase is transferred to the Modified_getBestNodeDNAddrPair as one of parameter. According to the RepNum, the function returns one DataNode information in the node list. Using this fundamental algorithm, the replica parallelism distributes the seek operation jobs across the replicas.

Algorithm 3 Modified_getBestNodeDNAddrPair(LocatedBlock block, Collection<DatanodeInfo> ignoredNodes, int RepNum)

```
    Description
    HDFS function to fetch a HDFS block from best replica

    Input
    /* related with HDFS block location*/
    LocateBlock block
    Collection<DatanodeInfo> ignoredNodes
    Int RepNum : variable to represent a ID of replica

    Output
    DataNode Information
    /* getLocation function returns a list of DataNode replicas
    in network distance order*/
1   DatanodeInfo[] nodes = block.getLocation();
    StorageType[] storageTypes = block.getStorageTypes()

    /* RepNum is a value which is calculated based on Hbase
    key*/
2   if ( !deadNodes.containsKey(node[RepNum-1])
    && !ignoredNodes.contains(nodes[RepNum-1]))){
        chosenNode = node[RepNum-1];
        break;
        }

7   End
```

Considering network distance, distributing read requests equally is too naïve approach and it can generate additional performance issue. When determining the amount of read requests for replica is processed, considering the locality of data is necessary to optimize the performance. Because fetching the data from other replica occurs additional network costs, the amount of data requests for each replica is inversely proportional to network distance from the client.

In addition, current released version of HDFS supports an archival storage API, data blocks can be stored in storages that have different storage types and blocks can provide information with storage type. For this reason, the storage type can be a factor which determines the amount of requests. SSDs have much lower read latency than HDD because seek time of SSDs is almost equal to zero. In addition, bandwidth of SSDs is also higher than one of HDD. When a read request is processed, putting weight on data block in SSDs results in better performance.

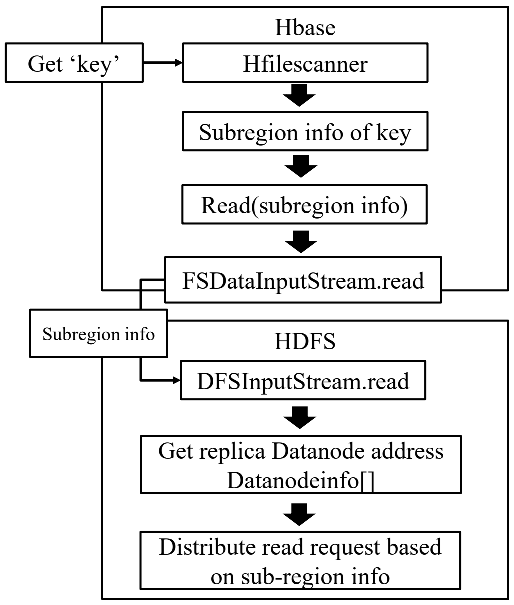

Figure 7. Algorithm of replica parallelism

4. Performance Prediction

We construct HDFS environment with 4 clusters which is composed of one NameNode and three data nodes. Table 1 explains the hardware environment of each cluster. We use Intel i7-3770 quad core CPU 3.4GHz, 16GB memory. Each cluster has two 1TB HDD and one 120GB SSD. Table 2 explains the software settings in our experiments. Each cluster runs on CentOS 7 with ext4 file system. Version of Hadoop is 2.7.1 which is recently released. We use Zookeeper-3.3.6 and HBase-1.1.4. We modify the code of HBase and HDFS.

To calculate the original performance of HBase. We use Yahoo! Cloud serving Benchmark (YCSB) which is well-known big data benchmark program to be used in comparing relative performance of NoSQL database management systems.

YCSB provides various workloads. We set the record count as a hundred thousand with 4 thread execution in experiment setting. In additional, Table 3 explains the type of workloads which are performed to evaluate the performance. Workload a is an update-intensive workload that has a mix of 50% read and 50% write. Workload b is a read-intensive workload that has a 95% read and 5% write mix. Workload c is a read-only workload. Workload d is a read latest workload. In this workload, new records are inserted, and the most recently inserted records are the most popular. Workload e is a short ranges workload that short ranges of records are queried.

Because we implemented only primitive version which is not perfectly performed, we only have to predict experiment results based on mathematical calculation and compare with the performance of original Hbase. There are two major points which can affect Hbase performance regarding data block selection: Network Cost, Storage Cost (Latency and Seek Time)

We experiment several cases using fio which is one of File System Benchmark Tool to predict the performance of replica parallelism. We assume that the default block size of HDFS is 128Mb and the default block size of Hbase is 64KB.

Total execution time will be calculated by sum of Disk Access time and Network time. First Figure 8 shows simple test results regarding Hard Disk Drive performance with fio. There are two cases in graph. One is that a test is established with random read operations for 128MB total size file with 64K block size matching original Hbase. The other is that a test is established with random read operations for 42MB total size file with 64K block size matching replica parallelism which has 3 replicas. As reduction of seek range is occurred, the later one is 3.5x faster. Because replica parallelism may generate additional network cost, we consider calculating the cost. As we calculate approximatively, 64KB block transfer latency is calculated that 64KB / 105MB/s = 0.6 msec. The number of seek operations in one replica is calculated that 42MB/64KB, so total additional network cost is calculated that 0.6 msec x 656.25 = 393.75 msec.

In this way, replica parallelism is predicted to perform 1.5 ~ 2.5x faster than original Hbase. If workload is read intensive, the performance gap will increase. Figure 9 and 10 shows the original Hbase performance with YCSB test and predicted replica parallelism performance. While we basically think about only Storage Cost and Network Cost, there are many performance factors such as locking mechanism for retaining concurrency and CPU computing cost, etc.

Table 1. Hardware environment

CPU	Intel i7-3770 quad-core CPU 3.4GHz
Memory	16GB memory
Disk	2TB HDD and OCZ Vector 120GB SSD
Network	1 Gigabit network (105MB/s)

Table 2. Software environment

OS	CentOS 7
File system	Ext4
Hadoop	Hadoop-2.7.1
Zookeeper	Zookeeper-3.3.6
HBase	HBase-1.1.4

Table 3. Type of workloads

Workload Name	**I/O Property**
Workload a	Update heavy workload
Workload b	Read mostly workload
Workload c	Read only workload
Workload d	Read latest workload
Workload e	Short ranges workload

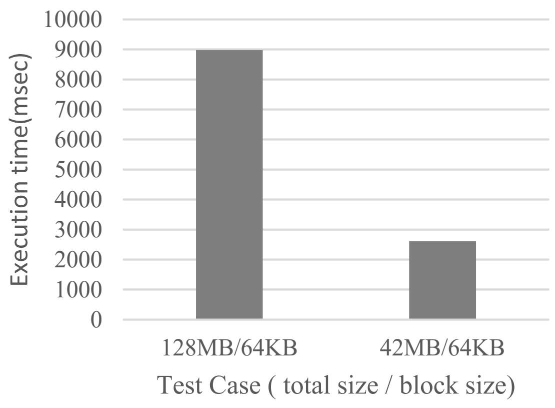

Figure 8. fio experiment

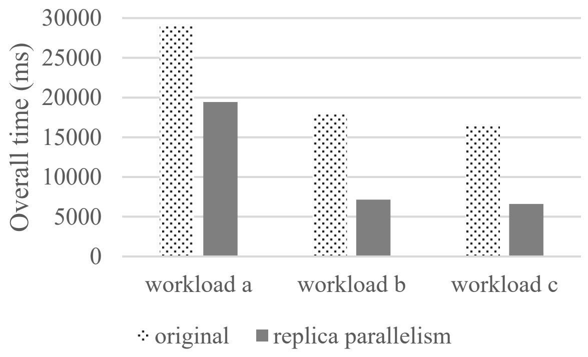

Figure 9. Experiment Prediction for Workload a, b and c

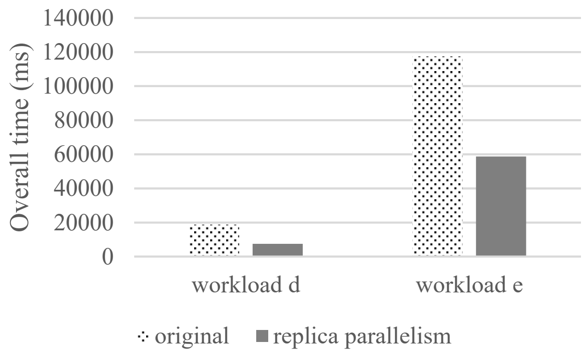

Figure 10 Experiment Prediction for Workload d and e

5. Conclusion

As large scale of data emerges, big data platforms handling the data have attracted attention of many researchers in recent years. Since performance is a critical point in big data system engineering, many researches have emerged to optimize the overall system performance with modifying the system architecture. Storage I/O takes an important part in performance. Therefore, optimization and improvement of storage I/O is one of the main subjects to be considered critically.

We propose a replica parallelism algorithm to utilize the different granularity of blocks between HBase and HDFS. A key idea is that lots of read requests in a HDFS block are distributed to several block replicas. In addition, storage type and network distance weight the amount of read requests distributed to get optimal performance of each replica. Researches about HBase have ignored practical storage structures relatively in case of handling extremely random and small data. Communication between HBase and HDFS about mutual block structures is meaningful to improve performance of storage I/O.

In future we research the detailed storage technologies. We try to improve the replica parallelism utilizing the characteristics of HDD completely. Evaluating the degree of read I/O's randomness in a HDFS block can be an another solution to get the optimal performance.

6. REFERENCES

[1] Apache Software Foundation. Apache Hadoop Documentation. https://hadoop.apache.org/docs/r2.7.2/

[2] Apache Software Foundation. Apache HBase Documentation. https://hbase.apache.org

[3] Apache Software Foundation. Apache Spark Documentation. https://spark.apache.org

[4] Apache Software Foundation. Apache Tajo Documentation. https://tajo.apache.org

[5] Apache Software Foundation. Apache Zookeeper Documentation. https://zookeeper.apache.org

[6] Ahmad, Muhammad Yousuf, and Bettina Kemme. 2015. "Compaction management in distributed key-value datastores." *Proceedings of the VLDB Endowment* 8.8 (2015): 850-861.

[7] Choi, H., Son, J., Yang, H., Ryu, H., Lim, B., Kim, S., and Chung, Y. D. 2013. Tajo: A distributed data warehouse system on large clusters. In Data Engineering (ICDE), 2013 IEEE 29th International Conference on (pp. 1320-1323). IEEE.

[8] Dean, Jeffrey, and Sanjay Ghemawat. 2010. "MapReduce: a flexible data processing tool." Communications of the ACM 53.1 (2010): 72-77.

[9] Dean, Jeffrey, and Sanjay Ghemawat. 2008. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.

[10] "Harter, T., Borthakur, D., Dong, S., Aiyer, A., Tang, L., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2014. Analysis of hdfs under hbase: A facebook messages case study. In Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14) (pp. 199-212).

[11] Moon, Sangwhan, Jaehwan Lee, and Yang Suk Kee. 2014. "Introducing ssds to the hadoop mapreduce framework." Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on. IEEE, 2014.

[12] Roh, H., Park, S., Kim, S., Shin, M., and Lee, S. W. 2011. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives.Proceedings of the VLDB Endowment, 5(4), 286-297.

[13] Sathiamoorthy, M., Asteris, M., Papailiopoulos, D., Dimakis, A. G., Vadali, R., Chen, S., and Borthakur, D. 2013. Xoring elephants: Novel erasure codes for big data. In Proceedings of the VLDB Endowment (Vol. 6, No. 5, pp. 325-336). VLDB Endowment.

[14] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. 2010. The hadoop distributed file system. In Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on (pp. 1-10). IEEE.

[15] Zhao, Dongfang, and Ioan Raicu. 2013. "HyCache: A user-level caching middleware for distributed file systems." *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013.