Block Placement in Distributed File Systems based on Block Access Frequency

Jianwei Liao, Zhigang Cai, Francois Trahay, Xiaoning Peng

***Abstract*—This paper proposes a new data placement policy to allocate data blocks across storage servers of distribute/parallel file systems, for yielding even block access workload distribution. To this end, we first analyze the history of block access sequence of a specific application, and then introduce a `k-partition` algorithm to divide data blocks into multiple groups, by referring their access frequency. After that, each group has almost same access workloads, we can thus distribute these block groups onto storage servers of distributed file system, to achieve the goal of uniformly assigning data blocks when running the application. In summary, this newly proposed data placement policy can yield not only an even data distribution, but also the block data access balance. The experimental results show that the proposed scheme can greatly reduce I/O time and better improve utilization of storage servers when running the database-relevant applications, compared with the commonly used block data placement strategy, i.e. the round-robin placement policy.**

***Index Terms*—Distributed File Systems, `k-partition`, Block Access Balance, Data Re-distribution, I/O Performance.**

I. Introduction

Parallel/distributed file systems have been widely adopted as the back-end storage for Big Data applications in high performance computing [1]. As a result, the aggregate transfer rate can be much higher, and overall storage capacity are supposed to be significantly enhanced [2]. In fact, big files are supposed to be divided into many blocks, which are striped over multiple storage servers, to enable these servers processing file data in parallel [24].

The data layout policy of a distributed/parallel file system, which organizes the physical data layout across storage servers, is a key factor in determining parallel I/O performance [5]. Therefore, preferable data layout policies can result in data locality and workload balance among storage servers, to primarily boost system performance. In general, the distributed/parallel file systems take advantage of the round-robin placement policy [3], or the uniform-at-random algorithm [4], to allocate data blocks across the storage servers, for yielding an even distribution of data amount.

On the other side, different from conventional applications, one of the main features of big data applications, such as database applications, is about the tight coupling relationship between data and computation, as computation tasks can be conducted only when the required data are available [3]. In other words, not only task assignment, but also data placement deeply affect the execution of big data applications. In fact, several research work [30], [31] analyzed how file system performance can be impacted by many factors of workload such as the distribution manners of files, I/O request sizes, and I/O access characteristics. As a conclusion, it is argued that a major issue in this context is to identify where to allocate data blocks and block replicas for these applications, to reach the targets of optimizing disk utilization, reducing network congestion, and improving data throughput [34].

Therefore, a number of advanced block (file) data placement schemes have been proposed for distributed file systems, by considering various of factors. Specifically, H. Song et al. [6] introduced a cost model to calculate the overall I/O cost of any given application, to guide choosing an appropriate layout policy for the application[1]. Also, they have presented a server-side adaptive data layout strategy in parallel file systems, for data-intensive applications with non-uniform data access patterns [7]. L. Gu et al. [3] and L. Wang et al. [5] have respectively proposed their data placement policies, for minimizing I/O overhead, and thus speeding up the execution of applications. D. Yuan et al [11] have explored data access patterns of scientific cloud workflows and then introduced a clustering data placement strategy in distributed file systems, for the purpose of reducing the I/O time of scientific applications. A novel scheme to initially place data blocks across storage nodes of the Hadoop distribution file system (HDFS) has been presented in [12], for balancing data processing while running MapReduce applications. Similar to [35], S. Yazd et al. [13] have presented a flexible block data placement strategy on storage servers in Hadoop-based datacenters, named as Mirrored Data Block Replication, to better cut down the energy utilization of storage servers.

However, these existing schemes may result in a waste of resources on storage nodes, and fail to speed up the execution of applications. The big data workloads associated to the different files, and even to the different parts of the same file might be totally diverse, which lead to varied levels of access frequency to data blocks. As a consequence, a storage server with frequently requested block data may stay busy, but other storage nodes having less frequently accessed block data blocks may keep idle, though both kinds of servers have the same number of data blocks. Eventually, the execution of applications depends on the completion time of the requested I/O services, provided by the busiest storage server.

To address this issue, we propose a novel data placement policy on the basis of access frequency to data blocks, for the purpose of achieving an even access distribution. Firstly, we collect access statistics on all data blocks when running the application. After that, data blocks are evenly divided into

[1]Note that we intentionally describe some of the most related work here to further motivate our proposal, but a more general survey of related work will be presented in Section II.

multiple groups, by referring their access frequency and access time. At last, different block groups can be allocated onto separated storage nodes. As a result, not merely the block load balance, but also the block access balance across the storage nodes can be ensured. To put it from another angle the storage servers can be well utilized and provisioned, to maximize the performance of big data applications, in the case of running them with multiple cycles. In summary, this paper makes the following two contributions:

- We have performed a detailed theoretical analysis on the history of block accesses belonging to a specific application. Then, we argue that distributing data blocks onto storage servers of distributed file system for yielding data load balance and access load balance, is a NP-complete problem. Regarding this issue, we have also introduced principles to guarantee achieving approximately optimal solutions to our target problem or similar ones.
- We have proposed the `k-partition` algorithm to sub-optimally classify data blocks into `k` groups. Consequently, these divided groups will be placed to `k` storage servers of distributed file system, to balance block access workloads, as well as speed up I/O processing. As a consequence, the execution time of application can be noticeably cut down, and the overall performance of file system can be also enhanced.

The rest of the paper is organized as follows: Section II discusses background knowledge and related work on data placement in distributed/parallel file systems. The design and implementation details of the newly proposed scheme of block data distribution are described in Section IV. Section V evaluates the implemented system, and then presents experimental results. At last, we conclude the paper in Section VI.

II. Background and Related Work

Many efforts have been contributed to decreases in time resulted by I/O operations, through providing flexible data (replica) placement solutions, when executing the target applications.

1) Data placement in conventional distributed/parallel file systems. J. Xie et al. [12] have introduced a novel way to initially allocate data blocks across storage nodes of the Hadoop distribution file system (HDFS), for the purpose of balancing data processing load, according to the computing capacity of each node. Furthermore, they have proposed data reorganization and redistribution algorithms, to adaptively regulate data layout on the basis of data access patterns by considering dynamic data insertions and deletions. H. Herodotou et al. [26] have introduced a self-tuning system for tuning big data analytic within the Hadoop stack. To reduce the imbalance data placement of write-read workflow in the storage, they have proposed to use a round-robin policy for chunk placement instead of the default random policy.

Co-Hadoop is an extension to the Hadoop distributed file system [15]. It can help the applications to control data placement at the file-system level [16]. To this end, the authors introduced a new file level property called locator, and then modified the default data placement policy of Hadoop to make use of this locator property for neatly placing data.

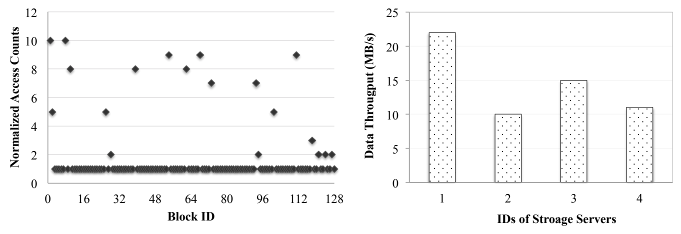
H. Song et al. [7] have proposed a strategy that adopts different stripe sizes for different file servers in PVFS [24], according to the data access characteristics on each individual server. Then, the busy file servers can fully utilize bandwidth to hold more data, and the file servers having limited request service rate can manage less data. S. Weil et al. [9] proposed CRUSH in Ceph, that is a scalable, pseudorandom data distribution function designed for distributed object-based storage systems. It efficiently maps data objects to storage devices without relying on a central directory.

2) Adaptive data placement policies. J. Wang et al. [14] recently argued that keeping physical data distribution and maintaining data blocks should be separated out from the metadata management and conducted by each storage node autonomously, for the purpose of decreasing the memory cost and maintenance cost on the master node. M. Bhadkamkar et al. [10] described BORG, a self-optimizing layer in the storage stack, which is able to automatically regulate disk data layout for adapting to the disk access patterns of workloads. The basic idea of BORG is to optimize both read and write traffic dynamically through making reads and writes more sequential. L. Wang et al. [5] have proposed a parallel file system for reducing the I/O time required by remote sensing applications. This file system can adaptively offer application-aware data layout policies, to benefit different data access patterns of remote sensing applications from the server side.

H. Hsiao et al. [34] have designed an attractive load rebalancing scheme for distributed file systems in Clouds. Because compute nodes may be dynamically upgraded, replaced, and added in the cloud system to bring about data load imbalance, their proposal is able to balance the loads of nodes and decrease the demanded movement cost by using the information on physical network locality and node heterogeneity. Besides, in order to eliminate the imbalance of parallel writes on distributed file systems, D. Huang et al. [27] proposed *Opass*, that employs a heatmap for monitoring the I/O state of storage servers, and uses a novel heuristic policy to choose an optimal storage node for serving write requests.

3) Application-specific data (file) placement. Targeting at different application contexts, many researchers have presented their specific schemes for purposely allocating file data. D. Yuan et al [11] have explored data access patterns of scientific cloud workflows and then introduced a clustering data placement strategy. In other words, this strategy is able to automatically distribute application data among storage nodes based on data dependencies. Consequently, it can minimize the overhead caused by data movement during the execution of workflows, as well as reduce the time spent waiting for the required data, since relevant data are managed locally.

S. Agarwal et al. [19] have presented an automated data placement mechanism, named Volley, for geo-distributed cloud services, by taking several factors into account, including WAN bandwidth cost, data center capacity limits, and data inter-dependencies. Similarly, B. Yu and J. Pan [33] have



(a) Access frequency on data blocks (b) Data throughput over storage servers

Fig. 1. Imbalanced data access workloads in parallel execution of a database application.

proposed a data placement policy to enhance the co-location of associated data and the localized data serving under the premise of ensuring the balance among storage nodes.

J. Wolf et al. [18] investigated how to conduct a disk placement of Video-on-Demand (VoD) file copies on the servers and the amount of load capacity assigned to each file copy, for minimizing the communication cost while ensuring quality of service of VoD services. In addition, H. Jin et al. [25] have proposed a joint optimization scheme that simultaneously optimizes virtual machine (VM) placement and network flow routing to maximize energy savings on the host machines.

None of aforementioned approaches, however, focuses on both data load balance and access load balance when placing file data to storage servers of distributed/parallel file system.

III. Motivations and Theoretical Analysis

This section first discusses our motivation to properly allocate data blocks onto storage servers. After that, we perform a detailed analysis on the issue of evenly distributing data blocks, and then demonstrate this problem is NP-hard.

A. Motivating Factors

To illustrate the significance of even data placement on storage servers of distributed/parallel file system, we launch a job running on a 12-node cluster to perform analysis on a small scale of TPC-C [21]. The file data are divided into multiple blocks (block size is 64KB), and these blocks are striped with the round-robin manner onto 4 storage servers in the PVFS file system [24].

There are varied levels of access frequency to the blocks, the access counts to the first 128 data blocks of a file are shown in Figure 1(a). As presented, the access frequency to data blocks is far from balanced. In other words, a small part of blocks contain the most of desirable data, though a major part of blocks are accessed only once. We argue that this situation causes an imbalanced access workload distribution over the storage nodes during parallel execution. Figure 1(b) further demonstrates the storage servers do have different data throughput, and this imbalance could seriously degrade the execution performance in many sub-dataset analyses. That is to say, the application's I/O time (even its execution time) is determined by the storage server having the heaviest I/O access workloads.

In conclusion, properly distributing data blocks onto the storage servers of distributed file system is critical to not only balancing I/O access workloads, but also speeding up the execution of applications.

B. Theoretical Analysis

This section conducts a theoretical analysis on preferably allocating data blocks onto storage servers of distributed/parallel file system, as expected.

To put it from another angle for stating this problem: there is a set of n positive numbers, i.e. $a_1, a_2, ..., a_n$, which represents block access frequency to n data blocks of $B_1, B_2, ..., B_n$. On the other side, assuming we have k storage servers in total, labeled as $S_1, S_2, ..., S_k$. The target is to partition the n positive numbers into k subsets, in which the sum of the elements of subgroups are equal or at least nearly equal, and the number of elements in each subgroup is almost the same. Thus, it indicates we can perfectly distribute n data blocks onto k storage servers of distributed file system, by also considering the access frequency to the blocks.

In order to express this division problem with a mathematics manner we first define a dummy variable of x_{ij}, to indicate whether the data block of B_i is distributed onto the storage severs of S_j. In the case of the value of x_{ij} is 1, it means the data block of B_i is allocate on S_j; otherwise, the data block is not allocated on the server of S_j. Next, we define a variable of A_j to represent the total access count with respect to the storage server of S_j:

$$A_j = \sum_{i=1}^{n} a_i * x_{ij} \tag{1}$$

The average accessed number of all storage servers can be addressed as the following equation:

$$\overline{A} = \sum_{j=1}^{k} A_j / k \tag{2}$$

Equation 3 defines the total number of blocks mapped onto the storage server of S_j.

$$D_j = \sum_{i=1}^{n} x_{ij} \tag{3}$$

Consequently, the average block number among all storage servers can be defined as:

$$\overline{D} = \sum_{j=1}^{k} D_j / k. \tag{4}$$

At last, we define a variable of N for showing the sum of access counts to all data blocks.

$$N = \sum_{i=1}^{n} a_i = \sum_{j=1}^{k} A_j, \tag{5}$$

We choose variance to measure the average degree of a set of data, we can transfer the problem to a standard optimization problem:

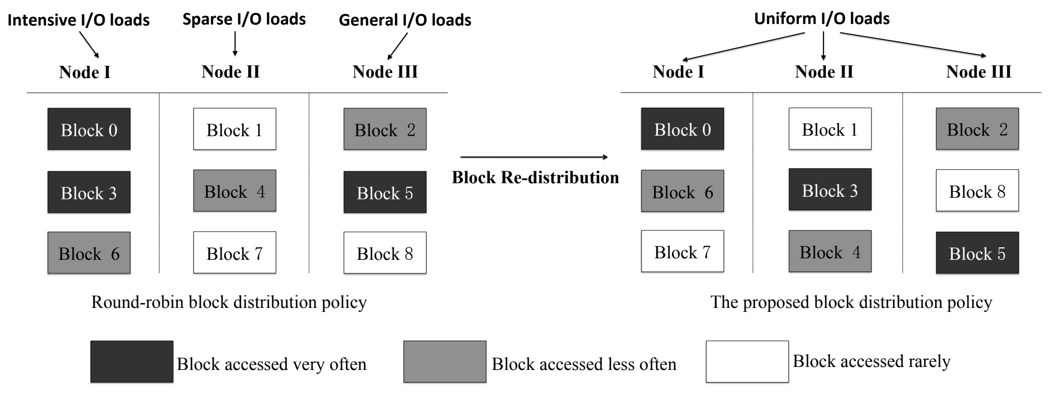


Fig. 2. Comparison overview of the round-robin block placement policy and the proposed placement policy.

$$\begin{aligned} &Min \quad \lambda\sum_{j=1}^{k}(A_j-\overline{A})^2+(1-\lambda)\sum_{j=1}^{k}(D_j-\overline{D})^2 \\ &St. \quad \sum_{j=1}^{k}x_{ij}=1 \quad (i=1,2,...,n \; and \; x_{ij}=0,1) \end{aligned} \tag{6}$$

where $0 \leq \lambda \leq 1$, to represent the weighing coefficient of balancing access counts and data amount.

According to the general rule, we let λ equal to `0.5`, and note that both $\overline{A}$ and $\overline{D}$ are constants with a given block access sequence. As a consequence, Equation 6 can be further simplified as Equation 7.

$$\begin{aligned} &Min \quad \sum_{j=1}^{k}A_j^2+\sum_{j=1}^{k}D_j^2 \\ &St. \quad \sum_{j=1}^{k}x_{ij}=1 \quad (i=1,2,...,n \; and \; x_{ij}=0,1) \end{aligned} \tag{7}$$

In the model presented with Equation 7, the first part of $\sum_{j=1}^{k}A_j^2$ in the objective function means the goal of yielding average accessed count on all storage servers. That is to say, this part may reach the minimum to N^2/k, when all A_js are (closely) equal to $\overline{A}$. For the second part of $\sum_{j=1}^{k}D_j^2$, it implies all data blocks are supposed to be evenly distributed onto all storage servers. In the case of all D_js are (nearly) equal to $\overline{D}$, this part can reach the minimum value to n^2/k. It is worth to say that we may not achieve the ideal optimal value, because the variable x_{ij} is a `0/1` dummy variable, instead of a continuous variable.

Through aforementioned analysis, we can identify the target problem is a `0-1` quadratic programming problem, which is a typical NP complete hard problem [17]. Therefore, we can understand that the above defined problem is also hard to solve. In order to minimize the time overhead caused by solving this problem, we have proposed a partition algorithm to classify data blocks into many groups, which have almost the same sum of access counts to the in-group blocks. Section IV will depict the details about the proposed partition algorithm.

IV. Design Philosophy and Implementation

Apart from presenting the system architecture, the algorithm used to classify data blocks into multiple groups, and its implementation specifications with the PVFS file system are elaborately described in this section.

A. System Architecture

Figure 2 demonstrates a comparison overview of the round-robin placement policy (left side) and the proposed placement policy (right side). Both of them are able to yield an even data amount distribution across storage servers of distributed/parallel file system, but the newly proposed one can achieve the goal of balancing block accesses over the storage servers.

As shown, the commonly used round-robin manner distributes data blocks evenly to the storage servers of distributed file system, regardless of the block access frequency. Consequently, the storage servers having many frequently accessed data blocks may become the bottleneck of the file system, even though some storage servers may stay idle. To overcome this limitation, we have proposed a new scheme to distribute data blocks, by taking the information on block access frequency into account.

In addition to yielding an average block allocation in quantity, we intend to achieve the goal of balanced block access workloads, associating with these storage servers. The basic idea is to divide all data blocks into multiple groups, with accordance to the number of storage servers. More importantly, the number of total accesses to the blocks in a group should be approximately equal to sums of numbers in other groups.

B. *k*-partition Grouping Algorithm

As discussed in Section III-B, the problem of grouping blocks in average is NP-hard in the ordinary sense [28]. It is not possible to have any polynomial-time solution for such problem [29]. In our application contexts, the selected

Algorithm 1: The Equal k-$partition$ Algorithm

```
   Input: $S$ is the list, and $k$ (# of parts);
   Output: $k$ equal parts of $a$;
1  if $k \leq 1$ then
2      return $S$;
3  if $k \geq len(S)$ then
4      return [s] for $s$ in $S$;
5  parts_between = [];
6  for $i$ in range(k) do
7      parts_between.append($(i + 1) * len(S)/k$);
8  a = $sum$(n) / k;
9  adjust_cnt = 0;
10 #loop over possible partition decisions
11 while true do
12     #partition the list into $k$ subsets
13     parts = [];
14     index = 0;
15     for div in parts_between do
16         parts.append(a[index:div]);
17         index = div;
18     parts.append(a[index:]);
19     worst_diff = 0;
20     worst_parts_idx = -1;
21     for p in parts do
22         diff = a - sum(p);
23         if abs(diff) >abs(worst_diff) then
24             worst_diff = diff;
25             worst_parts_idx = parts.index(p);
26     while adjust_cnt <COUNTS or worst_diff <DIFFS
        do
27         adjust_part(worst_parts_idx, worst_diff,
            parts_between, parts);
28         adjust_cnt += 1;
29     return parts;
```

approximation solution for this problem is based on the first-fit decreasing algorithm, which is often used to generate approximate solutions to similar NP-hard problems.

Regarding a sequence of block access counts belonging to a specific application, the modified first-fit decreasing algorithm is able to evenly divide the numbers (i.e. access counts) in the sequence into multiple (e.g. k) partitions. It first sorts the numbers in a descending order. Next, it distributes the first k numbers (the largest k numbers) across the k subgroups. After that, it iterates over the remaining $n-k$ numbers, and add each number into the subgroup that currently has the lowest sum. Finally, a group of n integer numbers (i.e. n data blocks having varied access counts) can be divided fairly into k subgroups, and the max of the sums of k subgroups is minimal.

Algorithm 1 specifically demonstrates this newly proposed grouping method with a Python style. In fact, this algorithm arranges integers in the list of S that represents access counts to a number of data blocks, into k partitions. This approach defines partition boundaries that divide the integer list of S in roughly equal numbers of elements, and then repeatedly searches for better partition decisions. To put it from another angle, we first create a list of indexes to a partition list of $partition\_between$, by using the index on the left of the partition, for representing the start index of the partition. Then it can roughly partition the list of S into equal groups of $len(S)/k$, but note that the last group may be a different size. Then, for the purpose of determining whether the current partitions are acceptable or not, the algorithm computes the sum differences of partitions, as shown in Lines $21 - 25$. At last, the process of partition can be ended when the iterative search reaches the predefined times (i.e. *COUNTS*) or the maximum differences among partitions is less than predefined threshold (i.e. *DIFFS*).

The criteria to end the algorithm are important for yielding a good result with complex data, and changing such predefined thresholds is a good way to experiment with getting improved results. Occasionally, we have to adjust the current partitions, when the ending criteria are not met, as presented in Lines $26 - 28$ of Algorithm 1.

The basic idea to adjust the partitioning of the worst partition that has the maximum sum, is to move it closer to the ideal size. Algorithm 2 illustrates the details of partitioning adjustment. To be specific, the overall goal is to take the worst partition and adjust its size to make its sum closer to the ideal. In general, if the worst partition is too big, we want to shrink the worst partition by moving one of its ends into the smaller of the two neighboring parts. On the other side, if the worst partition is too small, it is better to grow the partition by expanding the partition towards the larger of the two neighboring parts.

C. Implementation

We have implemented the proposed block data placement algorithm in PVFS2, that is a high performance distributed file system designed for high-bandwidth parallel access to large data files [24]. The current physical distribution mechanism used by PVFS2 is a simple, round-robin striping scheme. The function of distributing data is described with three parameters:

- `base`: the index of the starting I/O node, with `0` being the first node in the file system.
- `pcount`: the number of I/O servers on which data will be stored.
- `ssize`: strip size, the size of the contiguous chunks stored on I/O servers

In fact, the physical block placement is determined once the file is first created. Thus, it is possible to specify the aforementioned three parameters when using the function of `pvfs_open()` to create the file.

We have modified the implementation of `pvfs_open()`, to allow flexible block data distribution by a new request scheduler module with the given placement specifications.

Algorithm 2: Adjust Partition (`adjust_part(...)`)

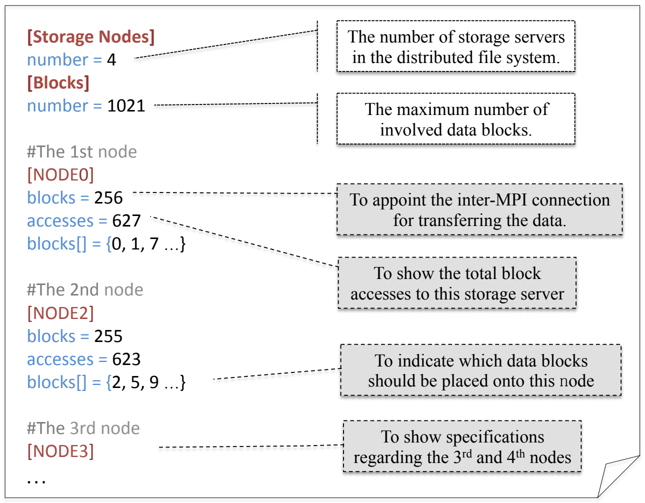
```
Input: $S$ is the list, and $k$ (# of parts);
Output: $k$ equal parts of $a$;
if worst_parts_idx == 0 then
    if worst_diff < 0 then
        parts_between[0] -= 1;
    else
        parts_between[0] += 1;
    if worst_parts_idx == len(parts)-1 then
        if worst_diff < 0 then
            parts_between[-1] += 1;
        else
            parts_between[-1] -= 1;
    else
        left_bound = worst_parts_idx -1;
        right_bound = worst_parts_idx;
        if wrost_diff < 0 then
            if sum(parts[a-1])
              > sum(parts[worst_parts_idx+1]) then
                parts_between[right_bound] -= 1;
            else
                parts_between[left_bound] += 1;
        else
            if sum(parts[a-1])
              > sum(parts[worst_parts_idx+1]) then
                parts_between[left_bound] -= 1;
            else
                parts_between[right_bound] += 1;
```

As discussed, the algorithm of grouping data blocks can split the blocks into several groups, by referring the history of access frequency to them. So that we are able to finally obtain a block distribution specification, regarding the file accessed by the application. Figure 3 shows an example specification of block distribution, when placing `1021` data blocks to `4` storage nodes in the case of running the target application. As seen, `256` blocks having totally `627` access requests are supposed to be mapped to *NODE0*, and then the detailed information on these blocks is illustrated in the array of *blocks*.

Given that a file accessed by the application links against the modified PVFS distributed file system, the block distribution settings are loaded with the relevant application, and the specified settings will guide the process of block distribution when the application tries to open the file.

V. Experiments and Evaluation

This section describes the experimental methodology for evaluating the file system with the newly proposed block placement policy, and then reports the experimental results. First, we describe the experimental setup, including the experimental platform and the selected benchmarks. Next, the experimental results and relevant discussions are presented, to show the feasibility and applicability of our newly proposed mechanism. At last, we make a brief summary about the findings obtained from the evaluation experiments.



```
[Storage Nodes]
number = 4
[Blocks]
number = 1021

#The 1st node
[NODE0]
blocks = 256
accesses = 627
blocks[] = {0, 1, 7 ...}

#The 2nd node
[NODE2]
blocks = 255
accesses = 623
blocks[] = {2, 5, 9 ...}

#The 3rd node
[NODE3]
...
```

The number of storage servers in the distributed file system.
The maximum number of involved data blocks.
To appoint the inter-MPI connection for transferring the data.
To show the total block accesses to this storage server
To indicate which data blocks should be placed onto this node
To show specifications regarding the 3rd and 4th nodes

Fig. 3. A simplified example of distributing `1021` blocks of the file onto `4` storage nodes, generated by running the k-partition algorithm.

TABLE I
Machine Specifications of Storage Servers and Clients

	Storage Servers	Clients
CPU	2xIntel E5410 2.33GHz	Intel E5800 3.20GHz
Memory	1x4GB 1066MHz/DDR3	4GB DDR3-SDRAM
Disk	6x114GB 7200rpm SATA	500GB 7200rpm SATA
Network	Intel 82598EB, 10GbE	1000Mb
OS	Ubuntu 13.10	Debian 6.0.4

A. Experimental Setup

1) Platform Specifications: We employed one cluster and one local area network (LAN) to conduct evaluation experiments. One active metadata server and five storage servers of the distributed file system are deployed on a 5-node cluster (the metadata server is deployed with a storage server on the same machine). All client file systems are installed on `12` machines of a LAN. Specifically, `12` client file systems are installed on the machines of the LAN that is connected with the server cluster by a 1 GigE Ethernet. Table I presents the specifications of machines on the server cluster as well as on the LAN. All clients are equipped with MPICH2-1.4.1.

2) Benchmarks: Since one of the targeted contexts to use the proposed mechanism is the database-related application, we selected the following two widely used, representative database-related benchmarks for conducting evaluation experiments:

- *TPC-C*, which is a typical online transaction processing (OLTP) benchmark, issued by the Transaction Processing Performance Council (TPC) [21]. It consists of simple

TABLE II
READ/WRITE WORKLOAD ANALYSIS OF TPC-C AND TPC-E

Metrics	**TPC-C**	**TPC-E**
Number of Tables	9	33
Number of RW Transactions	3	6
Number of RO Transactions	2	6
Transaction Mix(RO)	8.0%	76.9%
Transaction Mix(RW)	92.0%	23.1%
Read Ratio	62.44%	90.44%
Average Read Size (KB)	8.0	8.0
Write Ratio	37.56%	9.56%
Average Write Size (KB)	9.26	8.62

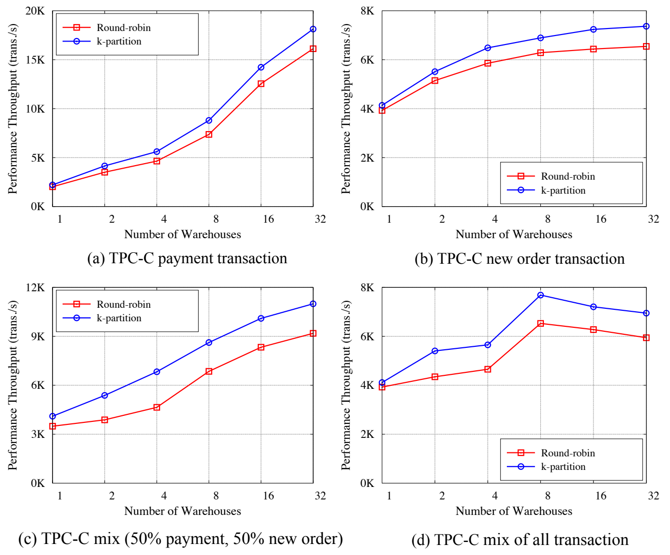


Fig. 4. Transaction throughput of *TPC-C* with varied number of warehouses

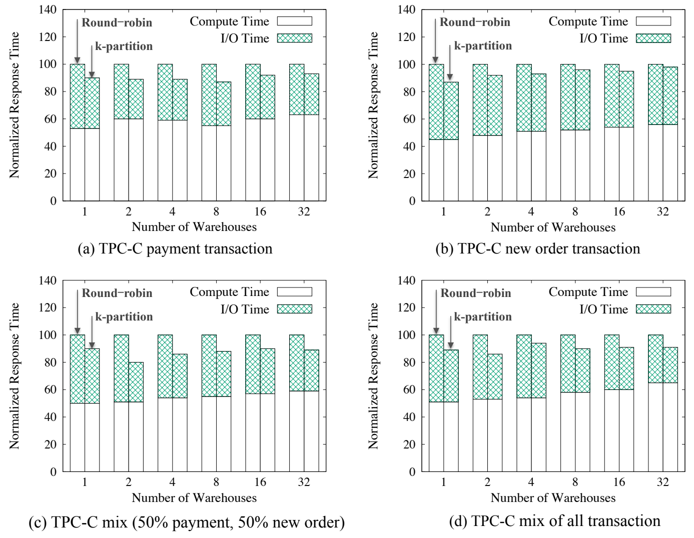


Fig. 5. Normalized time of *TPC-C* with varied number of warehouses

short-running transactions with frequent updates and less frequent index scans. Specifically, *TPC-C* reflects an online transaction processing (OLTP) database for an order-entry environment [23].

- *TPC-E*, which is the most recently standardized OLTP benchmark by TPC, as well. It includes transactions for real-time business intelligence combined with client-side requests [22] Namely, *TPC-E* models a financial brokerage house, and acts like a typical OLTP and an online analytic processing benchmark [23].

Table II summarizes the high-level specification comparisons of the mentioned two database benchmarks. As shown, one remarkable distinction of *TPC-E* from *TPC-C* is the majority of the transaction (mix) workloads are Read-Only (RO), whereas *TPC-C* has 92.0% Read-Write (RW) transaction (mix) workloads. In addition, there are different types and quantities of read/write requests targeting the tables in the benchmarks. For example, over 90% of the write operations are related to only 8 tables, among 33 tables in *TPC-E*.

B. Parameter Settings

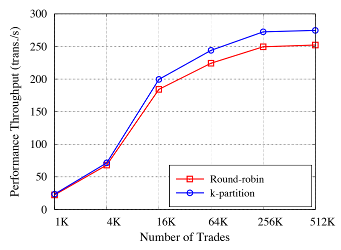
We have evaluated disk-based query processing frameworks by running the selected *TPC-C* and *TPC-E* benchmarks, which are update-intensive and read-intensive OLTP/OLAP workloads respectively. We run these two benchmarks, by employing MySQL (version 5.6.10) as the back-end database [20]. All 12 compute nodes perform the same job script, to execute the selected benchmarks. We set each client process repeating 10 times of transaction submissions, for yielding average experimental statistics.

To emulate varied levels of sharing data between client processes running on the compute nodes, we have adopted a similar evaluation method to the one introduced by C. Yan and A. Cheung [32]. In the evaluation experiments of running two selected benchmarks, we changed the data set size for each transaction access, to yield varying level of data access frequency.

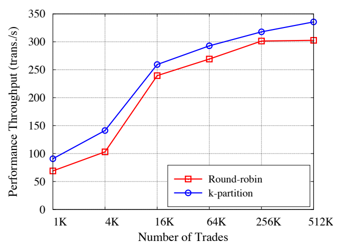
C. Results and Discussions

1) TPC-C Evaluation: For the *TPC-C* benchmark, we adjusted the data size by setting the number of warehouses from 1 to 32, and the total size of database is around 3 GB. In order to emulate a large amount of transactions, we make each client to issue requests repeatedly.

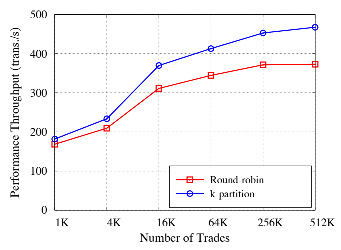
Figure 4 shows performance throughput of running *TPC-C* with varied number of warehouses, when using the *Round-robin* policy, and the proposed *k-partition* policy for placing data blocks. Specifically, Figures 4 (a) and 4(b) show the results of *TPC-C* having either payment transactions or new order transactions respectively. Figure 4(c) reports the results of *TPC-C* with 50% payment transactions and 50% new order transactions. Figure 4 (d) describes the results of *TPC-C* with a standard mix of five types of transactions, according to the specification. As seen, the newly proposed *k-partition* data placement policy can yield the best system performance in the



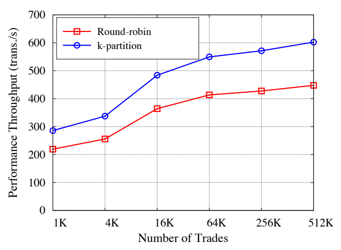
(a) TPC-E trade order transaction



(b) TPC-E trade update transaction

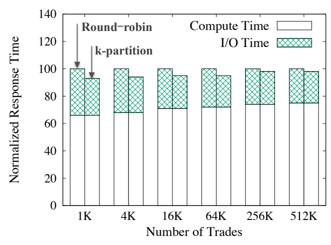


(c) TPC-E mix (50% order, 50% results)

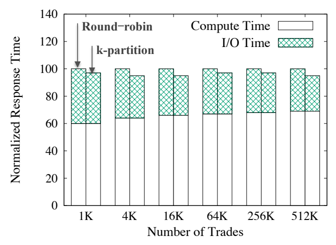


(d) TPC-E mix of all transaction

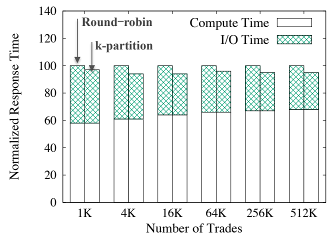
Fig. 6. Transaction throughput of *TPC-E* with varied number of trades



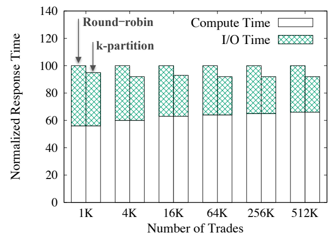
(a) TPC-E trade order transaction



(b) TPC-E trade update transaction

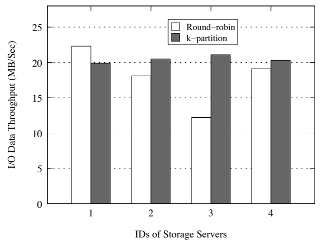


(c) TPC-E mix (50% order, 50% results)

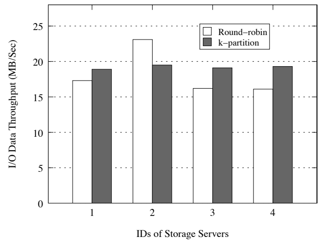


(d) TPC-E mix of all transaction

Fig. 7. Normalized response time of *TPC-E* with varied number of trades



(a) TPC-C benchmark



(b) TPC-E benchmark

Fig. 8. Data throughput of storage servers in the file system when running the benchmarks of *TPC-C* and *TPC-E*.

metric of transaction per second. In other words, *k-partition* can improve transaction performance by up to 21.2%.

We have also explored the completion time of running the benchmark of *TPC-C*, that consists of the time required for computation, and the time used for I/O processing. Figure 5 show the analysis on breakdown execution time of executing *TPC-C*. As we expected, the scheme of *k-partition* can reduce the time caused by allocating the data blocks, by more than 12.7% in average. The less completion time is resulted by the less I/O time when running the benchmark, because the computing time is same when employing the selected two data placement schemes.

2) TPC-E Evaluation: For the *TPC-E* benchmark, we altered the number of trades in each transaction access. Because the *trade update transaction* is employed to show the modifications to a set of trades, adjusting the number of trades can correspondingly change the amount of data in the transaction access. Specifically, we changed the number of trades from 1K to 512K (the total number of trades of entire trade table is 576K).

Figure 6 shows the results of transaction throughput when running *TPC-E*. Similar to the results of running *TPC-C* that were illustrated in Section V-C1, the newly proposed *k-partition* based data placement policy outperforms the round-robin scheme. That is to say, our proposal can yield better transaction performance when running the selected benchmarks.

Besides, the normalized time required for running the benchmark is recorded, and the relevant data are reported in Figure 7. As shown, the proposed scheme of *k-partition* does result in less time needed for processing I/O operations, so that the time required for completing the benchmark is consequently cut down.

3) Access Load Balance: The main motivation of access frequency-based data placement is to yield uniform I/O processing workloads over storage servers in the distributed file system. Therefore, we have measured the data throughput of storage servers when running the benchmarks, and Figure 8 presents relevant results.

In contrast to the default data distribution policy, i.e. *Round-robin* in the PVFS file system, the newly proposed scheme can noticeably benefit to balancing data throughput over storage servers of distributed file system. Consequently, not only the I/O process time can be decreased, but also the utilization ratio of storage servers can be greatly enhanced.

D. Summary

We have demonstrated that the proposed data placement policy can effectively and practically balance access workloads, and then reduce the I/O time for the target benchmarks. We carry out offline block partition, the time overhead to complete partition of data blocks by referring their access counts, is not presented. This is because the offline analysis does not affect the performance of file system, in the case of running the application.

With respect to comparing the proposed block data placement mechanism with the conventionally used round-robin placement policy, we emphasize the following two key observations. First, the proposed data placement policy can yield not only even data distribution, but also uniform data access workloads to the storage servers. Especially, in the case of the applications may have different levels of access frequency to the data blocks during their life cycles. Second, the longest I/O time caused by the storage server having the most access workloads is basically close to the shortest I/O time introduced by the server having the least access workloads. We then conclude that the proposed data placement policy can observably decrease the time overhead required by completing I/O requests when running BigData applications, such as database applications.

VI. Concluding Remarks

This paper proposes a novel block placement policy, which can ensure balanced access workloads with respect to storage servers in distributed/parallel file systems. To this end, we first analyze the history of block access sequence, for calculating the access frequency to the blocks on the storage servers. Then, the `k-partition` algorithm has been introduced to classify blocks into `k` groups, which are supposed to be correspondingly distributed to `k` storage servers. At last, the storage servers are able to process read/write access requests evenly, and there should not be any storage server taking a specific long time to complete the I/O requests of application.

The experiments on database-relevant benchmarks show that proper distribution of data blocks onto storage servers can definitely decrease the I/O time required by applications, as well as balance the I/O workloads to storage nodes. In brief, the proposed access frequency-based data placement policy does facilitate to the cases of the application required to be executed with multiple cycles. Therefore, it is possible to directing data placement on the basis of the block access history in the first cycle(s).

In the future work, we plan to explore more in dynamic data block movement among storage servers, by referring the access frequency to data blocks. Also, improving algorithm scalability is another direction of our future work.

Acknowledgements

This work was partially supported by "the Fundamental Research Funds for the Central Universities (No. XDJK2017B044)", "Hunan Provincial Natural Science Foundation of China (No. 2018JJ2309)" and "the Opening Project of State Key Laboratory for Novel Software Technology (No. KFKT2016B05)".

References

[1] J. Liao, L. Li, H. Chen , and X. Liu. Adaptive Replica Synchronization for Distributed File Systems. IEEE Systems Journal, Vol. 9(3): 865–877, 2015.

[2] J. Hartman, and J. Ousterhout. The Zebra striped network file system. ACM Transactions on Computer Systems, Vol. 13(3):274–310, 1995.

[3] L. Gu, D. Zeng, P. Li and S. Guo. Cost Minimization for Big Data Processing in Geo-Distributed Data Centers. IEEE Transactions on Emerging Topics in Computing, Vol.2 (3):314–323, 2014.

[4] A. D. Ferguson and R. Fonseca. Understanding Filesystem Imbalance in Hadoop. In Proceedings of the USENIX Annual Technical Conference, Poster, 2010.

[5] L. Wang, Y. Ma, and A. Zomaya et al. A Parallel File System with Application-Aware Data Layout Policies for Massive Remote Sensing Image Processing in Digital Earth. IEEE Transactions on Parallel and Distributed Systems, Vol. 26(6):1497–1508, 2015.

[6] H. Song, Y. Yin, and Y. Chen et al. A cost-intelligent application-specific data layout scheme for parallel file systems. In Proceedings of ACM International Symposium on High Performance Distributed Computing (HPDC '2011), pp. 37–48, 2011.

[7] H. Song, H. Jin, and J. He et al. A Server-Level Adaptive Data Layout Strategy for Parallel File Systems. In Proceedings of IEEE, International Parallel and Distributed Processing Symposium Workshops & Phd Forum. IEEE Computer Society, pp. 2095–2103, 2012.

[8] V. Pai, M. Aron, and G. Banga et al. Locality-aware request distribution in cluster-based network servers. ACM SIGOPS Operating Systems Review, Vol. 32(5):205–216, 1998.

[9] S. Weil, S. Brandt, and E. Miller et al. CRUSH: Controlled, scalable, decentralized placement of replicated data. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC '06), 2006.

[10] M. Bhadkamkar, J. Guerra, and L. Useche et al. BORG: Block-reORGanization for Self-optimizing Storage Systems. In Proceedings of (FAST '09), pp. 183–196, 2009.

[11] D. Yuan, Y. Yang, and X. Liu et al. A data placement strategy in scientific cloud workflows. Future Generation Computer Systems, Vol. 26(8): 1200–1214, 2010.

[12] J. Xie, S. Yin, and X. Ruan et al. Improving MapReduce Performance through Data Placement in Heterogeneous Hadoop Clusters. IEEE International Symposium on Parallel & Distributed Processing Workshops & Phd Forum, pp. 1–9, 2010.

[13] S. A. Yazd, S. Venkatesan, and N. Mittal. Boosting energy efficiency with mirrored data block replication policy and energy scheduler. SIGOPS Oper. Syst. Rev., Vol. 47(2):33-40, 2013.

[14] J. Wang, X. Zhang, and J. Zhang et al. Deister: A light-weight autonomous block management in data-intensive file systems using deterministic declustering distribution. Journal of Parallel and Distributed Computing. Vol. 108 (2017): 3–13, 2017.

[15] D.Borthakur. The Hadoop Distributed File System: Architecture and Design. The Apache Software Foundation, 2007.

[16] M. Eltabakh, Y. Tian, F. Ozcan and R. Gemulla et al. CoHadoop: flexible data placement and its exploitation in Hadoop. Proceedings of the VLDB Endowment, Vol. 4(9):575–85, 2011.

[17] M. Garey, and D. Johnson. Computers and intractability: A guide to the theory of NP completeness. Freeman, San Francisco, 1979.

[18] J. Wolf, S. Philip, and H, Shachnai. Disk load balancing for video-on-demand systems. Multimedia Systems, Vol. 5(6):358–370, 1997.

[19] S. Agarwal, J. Dunagan, N. Jain, and S. Saroiu et al. Volley: Automated data placement for geo-distributed cloud services. In Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10), pp. 17–32, 2010.

[20] MySQL Database. http://dev.mysql.com/downloads/[Accssed in July, 2014]

[21] Transaction Processing Performance Council. TPC-C Benchmark Revision 5.11.0. http://www.tpc.org/tpcc/. [Accssed in Feb., 2016]

[22] Transaction Processing Performance Council. TPC-E Benchmark Version 1.14.0. http://www.tpc.org/tpce/. [Accssed in Feb., 2016]

[23] C. Curino, E. Jones, and Y. Zhang et al. Schism: a workload-driven approach to database replication and partitioning. Proceedings of the VLDB Endowment, Vol. 3(1-2): 48–57, 2010.

[24] P. Carns, R. Ross, and R. Thakur PVFS: a parallel file system for linux clusters. Linux Showcase & Conference. USENIX Association, pp. 28-28, 2000.

[25] H. Jin, T. Cheocherngngarn, and D. Levy et al. Joint Host-Network Optimization for Energy-Efficient Data Center Networking. In Proceedings of the 11th International Symposium on Parallel and Distributed Processing (ISPA '2013), pp. 623–634.

[26] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. CIDR, Vol.11, pp. 261–272, 2011.

[27] D. Huang, D. Han, and J. Wang et al. Achieving Load Balance for Parallel Data Access on Distributed File Systems. IEEE Transactions on Computers, PP(99):1-1, 2017.

[28] K. Andreev, and H. Racke. Balanced graph partitioning. Theory of Computing Systems, Vol. 39(6): 929–939, 2006.

[29] C. Ferreira, A. Martin, and C.Souza et al. The node capacitated graph partitioning problem: a computational study. Mathematical Programming, Vol. 81(2):229–256, 1998.

[30] F. Isaila and W. F. Tichy. Clusterfile: A Flexible Physical Layout Parallel File System. In Proceedings of IEEE International Conference on Cluster Computing (Cluster ’2001), 2001.

[31] F. Wang, Q. Xin, B. Hong, and S. A. Brandt et al. File System Workload Analysis for Large Scientific Computing Applications. In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST ’2004), pp, 139–152, 2004.

[32] Yan C. and Cheung A. Leveraging lock contention to improve OLTP application performance. In Proceedings of the VLDB Endowment 2016, Vol. 9(5): 444–455, 2016.

[33] B. Yu, and J. Pan. Location-aware associated data placement for geo-distributed data-intensive applications. In Proceedings of IEEE Conference on Computer Communications (INFOCOM ’2015), pp. 603–611, 2015.

[34] H. C. Hsiao, H. Y. Chung, H. Shen and Y. C. Chao. Load Rebalancing for Distributed File Systems in Clouds. IEEE Transactions on Parallel and Distributed Systems, Vol. 24(5):951–962, 2013.

[35] C. Lin, and Y. Lin. An overall approach to achieve load balancing for Hadoop Distributed File System. International Journal of Web & Grid Services, Vol. 13(4):448-466, 2017.