A Framework for Power of 2 Based Scalable Data Storage in Object-Based File System

Ohnmar Aung and Nilar Thein

Abstract—As the amount of data in the today's storage systems has been growing over times, the ideas to expand the new resources are being considered to meet the system's requirement. Adding or removing resources will make throughput of the current state changes. To lower cost or make efficiency in maintaining the system's balance, one trial by another that serve nothing before is being replaced. B+ tree-based indexing algorithm is widely being popular for managing data dynamically in today's storage systems. Fast data insertion, deletion and searching are also concerned with the system's performance. Those criteria are heavily dependent on the order size and height of the tree used because it determines how large a B+ Tree will become and how much the system can hold data and throughput. The proposed system modifies the traditional B+ Tree in the form power of 2-based for data expansion and it is designed on object-based file system.

***Index Terms*—B+ tree, power of 2-based clustering**

I. Introduction

In modern storage systems, changes have occurred over time as new requirements are added on demand to supply needed capacity or bandwidth. Those file systems have been making storage management simplified and try to hide most of the complexities behind in order to get the systems performance high. More importantly, distributed file system workloads are inherently dynamic, with significant variation in data and metadata access as active applications and data sets change over time. Consequently, the performance of the storage system depends on the way objects included in the file system's attributes which can be managed by using appropriate method [1].

But, the higher the desired level to get, the more storage overheads will be cost and the longer the time to maintain data consistency. Even RAID-based storage systems get longer time to rebuild as increases in disk capacity outpace increases in bandwidth. These may be challenging to ensure high reliability and scalability for large-scale storage systems [2].

The B+ tree is easy to implement for being balanced structure in which all path from root to the nodes are same. Besides, the algorithm "B+ Tree" mostly used by the file system usually has been grown by splitting nodes (internal or leaf) in the tree and such expansion causes the tree level to be high. This results in frequent I/O operations to be taken much longer time throughout the whole process. Being raised more levels, system scales logarithmically and then storage's useful capacity grows more slowly as it gets larger. Such natures are now being facing in those systems deploying tree-based approach for data storage, which becomes a major problem to be solved. This has resulted from increasing B+ Tree size associated with its order. Until now, there is no system based on B+ tree indexing structure with limited order and height for getting performance better. The more new resources are required to be added, the more complicated the workload for exercising equally or greater in proportion to the number of I/O events directed toward the contents of each targeted file to be found.

The proposed system stores files as objects in the form of B+ tree. In contrast with traditional tree, the order of the tree will grow based on power of 2-based expansion as the system needs. Contrary from those having unknown order and height approaches, the system can scale well over whenever resource demands and takes no longer than those found in nature from theoretical point of view. And it also can promises to the complexity not being higher than usual.

Section II describes about some related work in the current storage systems. The theoretical background used in the proposed system is explained in Section III. The overview of the proposed system is described in Section IV. The discussion of the proposed system is concluded in Section V. [1], [2]

II. Related Work

Due to continual surveys, modern storage designs have been moved to OSD approach (Object-based Storage Devices). The concept is to access files in the form of objects in which both data and metadata are encapsulated which makes storage system simple as well as management feasible.

Hadoop is one of the most popular file systems intended for providing only one name-node which has to store the entire file system namespace in main memory. Having single name-node creates a single point of failure and a potential performance bottleneck for workloads which require multiple operations of huge amount of metadata [1].

Ceph, a scalable distributed object-based file system, emphasis on their object's replication under scalable hashing using three fundamental features: decoupled data and metadata, dynamic metadata sub-tree partition and automatic distributed object store. The system has gained both of scalable and replication to the certain level in which rebalancing the system state is achieved via CRUSH (Controlled Replication under Scalable Hashing). Recovering the system state depends on the weight of the underlying nodes. The higher the applied storage tree level reaches, the longer the time to be completed [1].

Cluster file systems such as Lustre, Gluster, relies on the

Manuscript received February 1, 2013; revised March 12, 2013.

The authors are with the University of Computer Studies, Yangon, Republic of Union of Myanmar (e-mail: ohnmaraung2008@gmail.com, nilarthein@gmail.com)

DOI: 10.7763/IJFCC.2013.V2.216

powerful, dedicated metadata servers for conducting all file metadata operations, leading to a hierarchical structure that scales well only in combination with high-speed, low-latency and symmetric network links [3].

One distributed network file system like Wofs that splits a file into many small objects, stores these objects in remote file servers, and uses a special B+ tree to manage the metadata of these objects. Besides, it uses the object-range locking policy to avoid data incoherence and improve performance [4].

Consider to limiting the order and height of B+ Tree becomes critical issue for system performance. The proposed system deeply takes into account "the order and height" of the tree for resource expansion and it also reduce unnecessary spaces whenever the system expands. Besides, the system can promise the complexity due to frequent insertion and deletion not being higher than original B+ tree. [1], [3], [4]

III. Background Theory

A. Storage Managements

The tasks involved in the traditional storage management systems partition available storage space into LUNs (i.e., logical units that are one or more disks or a subset of a RAID array), assign LUN ownership to different hosts, configure RAID parameters, create file systems or databases on LUNs, and connect clients to the correct server for their storage. This can be a labor-intensive scenario. Seeking to provide a simplified model for managing spaces is one of the major research issues in today's storage system [5].

Object-based Storage Device (OSD) is the most popular trend which can provide the file sharing capability needed for scientific and technical applications while delivering the performance and scalability needed to make the Linux cluster architecture effective. The unique design of the OSD differs substantially from standard storage devices such as Fibre Channel (FC) or Integrated Drive Electronics (IDE), with their traditional block-based interface. This is accomplished by moving low-level storage functions into the storage device and accessing the device through a standard object interface [6].

B. Tree-based Data Storage

Many tree-based algorithms are used to store data; however, they cannot handle the entire tree status of balancing after some operations like insertion and deletion. Those might store for fast efficient insertion well, but bad ending in deletion. Consequently, maintaining system's balance after deletion becomes a major problem in today's tree-based storage area.

Other balanced trees such as AVL trees and Red-Black Trees use the height of the sub-trees for balancing whereas WBT (Weighted Balanced Trees) is based on the size of the sub-trees below each node. RUSH is designed to make distributed storage by decentralized approach where the algorithm works as mapping replicated objects to a scalable collection of storage or disk. Yet, as changes occur which make the storage deeper and the round trip time to up and down across the tree depth cause complexity problem. In addition, where to place data in the tree makes time consumption which appears a problem that needs to be tackled [7]. [5], [6], [7]

IV. Overview of the Proposed System

In this section, we briefly discuss about the workflow of the proposed system. Firstly, the input data must be grouped into specific clusters according to its weight. Then, those objects are dynamically placed using the algorithm within their clusters.

A. Objects

Early storage systems are based on block units which later overhead quickly mounts up since metadata server has to manage each individual blocks and also have limited intelligence. Again, storage trend moves data to be striped as files across multiple file servers. This requires that each of the file servers still act as both metadata and data storage for the files. A problem comes with metadata bottleneck resulting from scaling up the systems or if security is important.

Being found difficulties treating with both of files and blocks, a new approach is adopted in which files are broken into smaller chunks called objects identified by unique numbers rather than traditional path names. This simplifies and speeds accessing the objects. The proposed system is designed to store files as objects and allow the storage systems to scale performance and capacity in the same manner as compute clusters [8].

B. Weight-Based Allocation

Frequent data insertion and deletion can make the system different from the current state and it also requires the system to be load balance. Since B+ tree is self-balanced structure, which is suitable to weight-based object allocation for the proposed system. Accessing the object is only to use the *object id* which is calculated based on particular weights and locates where to place in the tree.

All objects IDs in the system are organized by a B+ tree. The object index is derived from the calculation of its weight. Therefore, a single attribute < *object id*> is supported as an index which points to the actual location of the object in the cluster.

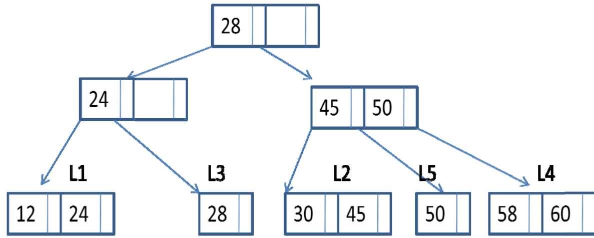
Since B+ tree consists of two types of nodes: internal nodes and leaf nodes. Internal points to other nodes in the tree whereas the leaf node stores the actual address of the object and internal nodes points to the index of the child nodes which are allocated by their weight. The placement of the leaf node points to actual data using data pointers. In addition, the leaf node also contains an additional pointer, called the sibling pointer, which is used to improve the efficiency of certain types of search [9].

Let consider, there may be any number of objects with varied size to be placed. The input sequence may be like that: 28, 50, 12, 45, 24, 60, 58 and 30 in KB. The algorithm starts as a single leaf node, L1, which is empty and hence, the key value 28 must be placed in leaf node L1. Again, search for the location where key value 50 is expected to be found. This is in leaf node L1. There is room in L1 so insert the new key.

Searching for where the key value 12 should appear also results in L1 but L1 is now full. By algorithm, there must be a

maximum of two keys in each leaf node and thus, L1 must be split into two nodes. The first node will contain the first half of the keys and second node will contain the second half of the keys.

However, it requires a new root node to point to each of these nodes and so, creates a new root node and promotes the rightmost key from node L1. A complete allocation of objects in the first sub-cluster is illustrated in Fig. 1. [8], [9]



Input Sequence (KB): 28, 50, 12, 45, 24, 60, 58, 30

Fig. 1. Object allocation in B+ Tree

C. Scaling Linearly with Power of 2-Based Scheme

Scaling storage linearly seems somewhat counter-intuitive on the surface since it is so easy to simply purchase another set of disks to double the size of available storage [10]. Capacity expansion requires the existing system to mange those disks for scaling as well. There must be enough system peripherals like CPU capacity as well as file system must scale to support the added data and meet the increased number of clients accessing those disks.

The order, or branching factor b of a B+ tree measures the capacity of nodes (i.e. the number of children nodes) for internal nodes in the tree. The actual number of children for a node, referred to here as m, is constrained for internal nodes so that $\lceil b/2 \rceil \leq m \leq b$. The root is an exception: it is allowed to have as few as two children. For example, if the order of a B+ tree is 7, each internal node (except for the root) may have between 4 and 7 children; the root may have between 2 and 7. Leaf nodes have no children, but are constrained so that the number of keys must be at least $\lfloor b/2 \rfloor$ and at most $b - 1$. In the situation where a B+ tree is nearly empty, it only contains one node, which is a leaf node [11].

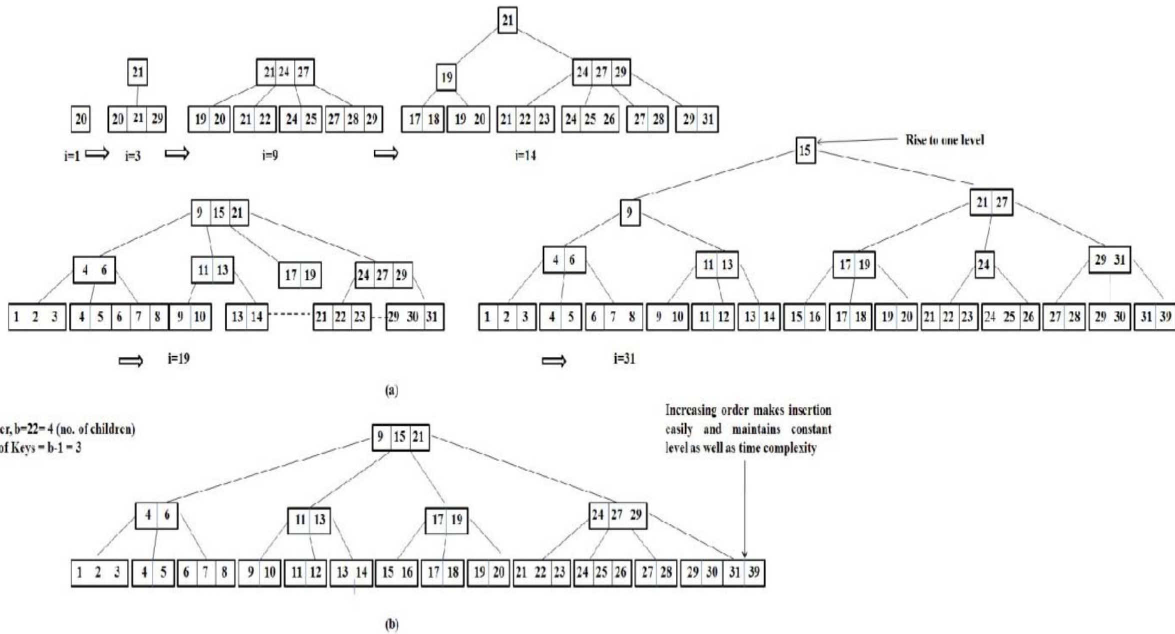


Fig. 2. (a) Object allocation in traditional B+ tree, (b) Object allocation in proposed B+ tree

Traditional B+ algorithm expands tree by making one level up. The performance of the algorithm depends upon the depth of the tree. The systems which used B+ tree can scale linearly based on the depth. Contrary from this, the proposed system is designed to scale by making fixed height and grows linearly depending upon the order of the tree. This can maintain the complexity at constant level. [10][11]

D. Proposed B+ Tree

As for data insertion in the existing media, the traditional B+ tree can take at least two steps (levels) for initial and then, the tree gets longer and longer as the number of stored item increased. No one can tell how large the tree depth and can result in delaying access time.

In afford to reduce the load traffic resulting from large data traversing; the proposed system makes little changes to the original tree to be getting better performance. Since objects of the proposed system are arranged by their weight, it must be needed to know which object in the present system has the nearest value to the new object to place. Step 1 is taken new object into account for searching nearest neighborhood. The second step of the algorithm is not different from the original view. The bucket found in the previous step is checked whether it is full or not. If the condition is “ok” (not full), data is only placed. Otherwise, bucket separation is performed and new leaf’s smallest key is addressed into the parent node. After passing two steps, the next one is only considered for increasing order of the parent node which is a major contribution of the proposed system for achieving high data available and removing unnecessary network traffic for frequent data insertion and searching time. In those cases experienced in the former B+ Tree, parent nodes is also split again and creates new keys and cause the tree level high. This

can be searching time further and further in parallel with the node number increased. The proposed B+ Tree simplifies it by only raising the tree order according to power of 2-based form. Having increased the order by 2 power, much more parent nodes as well as child nodes can be handled and also the time complexity remains stable.

E. Resource Expansion with Proposed B+ Tree

Beginning from the base 2 of the order (b), there must be at most one search key value (b-1) and two child pointer (*b*<=*n*<=*b*) for each internal node as well as the root node. When new objects are requested to be stored, the tree requires expanding. However, node allocation starting from 2^1 is too short to be explained and thus, it will be more clear in the example with 2^2 of the order value.

Therefore, it raised the order by power of 2-based form and now, the order value becomes 2^2 (*b*=4) and the number of children grows up to 4 whereas 3 for search key value in each internal node.

As shown in Fig 2(a), input sequences (*i*=1 to 31 times for example) comes 20, 29, 21... Etc at random and they are stored in the tree by ordering of their weights. When input count reaches to the certain value (e.g *i*=3, 14 and 31) that exceeds the level or number of children restricted for each node to handle, the next data insertion will cause splitting the existing node and creates one level up by forming new root. Traditional B+ tree is kept self-balancing (all path from the root to the leaf is same) rising the depth of the tree. This last longer searching time and access time for both of next insertion and deletion data will suffer from it. Node may be between b/2 and at most b like in the usual B+ tree. As for high, there must be at most 2 for the whole tree and no level improvement is allowed. Only the value of the order can be promoted up to power of 2-based for resource expansion. The reason for this is the following: In ordinary B+ tree model, if b is threshold for each node to have, then adding new record which exceeds the specified threshold makes node splitting in two conditions, either of parents or child node. Regular splitting of those two nodes is explained in Section 3.2. Instead of splitting parent node, increasing the order size can be the level of complexity constant and no high is raised. For input count *i*=31, depth of the tree becomes upper one level.

F. Sub-Cluster Selection and Weighting

Whenever new servers or resources are required in the systems, the capacity or throughput of the servers will be different from that of the existing servers. To make fast data lookup and reorganization in large distributed storage systems, they must allow the received nodes to reweight. In the proposed system, the users are allowed to insert data of any size as well as to delete. Objects in the same cluster should be related to each other. Suppose there may be objects o_1, o_2.....*to* o_n and each with varied sizes. The allocation scheme is based on per-object weight value.

Storage clusters are assigned by object's weights to control the relative amount of data using the selection algorithm in which clusters are specified like a set; *C={c1,c2,….,cn}* and the integer input to the algorithm, *w*, is typically an object's weight for those objects belongs to the related or similar weighted groups. The *select<c,w>* where *c* is the cluster types and *w* means the weight of the object iterates over each element c € C and chooses one of the sub-cluster *'c'* suitable for the current object to be collected. Data is placed in the hierarchy by recursively selection nested items (sub-clusters) in the respective sub-clusters locally. Sub-cluster classification is done with a simple selection algorithm and that won't be long *O(log n)* for the worst case. This results in fast data insertion and deletion for a large storage system as shown in Fig 2.

G. Assigning Object ID

Having carried two attributes *<c,w>* across the tree, it can calculate the current object *id* in which *OID= h(w) mod n* for fast data deletion and retrieval. It first decides in which sub-cluster an object has to be placed by weighting the object's size. If one of the sub-cluster suited for the object to be grouped is found, it started to find the position in the tree.

Since placing data in the B+ tree begins from the root, it compares its weight and the current root's weight. Based on the situation of either less than or greater than the current size, it descends down the tree until the node reaches the leaf. Once it is settled in the tree, the *object id* is computed.

H. Theoretical Complexities

Fig 3 shows the average time complexity for basic I/O operations like insertion, deletion and retrieving objects within a tree over ordinary variable depth versus the proposed fixed two level hierarchies. B+ tree performance is logarithmic with respect to the number of height. The vertical line represents the value of complexity whereas level, B+ tree scale as O (log_b *n*)- linearly with the hierarchy depth. When and where fixed level is defined, there is a slight performance advantage over normal range. The total time complexity of the tree takes O (log_b *n*) in general for b (order) of the tree with h (level) index. When increasing height, the depth becomes longer and it is taken time complexity more complicated.

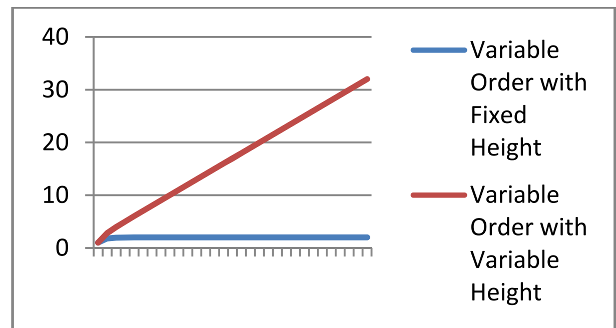


Fig. 3. Complexities comparison of traditional B+ tree and proposed approach

TABLE I: THE ARRANGEMENT OF CHANNELS PROPOSED B+ TREE BASED DATA INSERTION ALGORITHM

Insertion

Step 1: Perform a search to determine what bucket the new record should go into.

Step 2: If the bucket is not full, add the record.
Otherwise, split the bucket.
Allocate new leaf and move half the bucket's elements to the new bucket.
Insert the new leaf's smallest key and address into the parent.

Step 3: If the parent is full, check whether high is within range.
If high is ok, split it too.
Add the middle key to the parent node
Repeat until a parent is found that need not split.
If the root splits, create a new root which has one key and two pointers.
If high is not ok, promote the value of order. Go to Step 2.

The upper line that rose exponentially shows the results of value settings at variable order (increasing with power of 2-based form: $b=2^1, 2^2, 2^3,\ldots,2^n$) with variable height (one level increases whenever there is node splitting occurred) and it is shown that the algorithm complexity becomes higher and higher as the level increases. Different from those, the lower line remained stable proves that variable order (based on power of 2) with fixed height (up most two level hierarchy) can maintain the complexity constant. Whatever the order value changes, the proposed system complexity remain steady by handling with fixed height.

V. Conclusion

The Object Storage Architecture provides a single-system-image file system with the traditional sharing and management features of NAS systems and improves on the resource consolidation and scalable performance of SAN systems. Conventional distributed file systems has been seeking ways of reducing bottlenecks during accessing to the storage server. Managing the underlying storage system efficiently is a major research issue in modern storage areas. The proposed system overcomes these barriers by limiting the order and height of the B+ tree for data expansion. And object allocation is weighted by their sizes and this can match with the nature of B+ tree. The proposed approach can perform better than traditional B+ tree for efficient data insertion and deletion. It is shown that the modified B+ tree data storage hierarchy can maintain total time complexity of the algorithm stable.

References

[1] E. M. Estolano, C. Maltzahn, A. Khurana, A. Nelson, S. Brandt, and S. Weil, "Ceph as a Scalable Alternative to the Hadoop Distributed File System," vol. 35, no. 4, August 2 010, pp. 38-49.

[2] E. L Miller, Q. Xin T. J. E. Schwarz, and S. J, "Evaluation of Distributed Recovery in Large-Scale Storage Systems," *Computer Engineering Dept. Santa Clara University and Storage Systems Research Center*, University of California, Santa Cruz, 2004.

[3] G. Parissis and T. Apostolopoulos, "A distributed hash table-based approach for providing a file system neutral, robust and resilient storage infrastructure," Research Project, Department of Informatics, Athens, Greece, 2009.

[4] W. C. Chia and H. Yarsun, "Wofs: A Distributed Network File System Supporting Fast Data Insertion and Truncation," presented at International Workshop on Storage Network Architecture and Parallel I/Os, 2010.

[5] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable Performance of the Panasas Parallel File System," in *Proc. 6th USENIX Conference on File and Storage Technologies (FAST'08)*, San Jose, California, February 26-29, 2008.

[6] Panasa White Paper: Object Storage Architecture. [Online]. Available: http://www.panasas.com

[7] E. L. Miller and R. J. Honicky., "RUSH: Balanced Decentralized Distribution for Replication Data in Scalable Storage Clusters," Storage Systems Research Center, Jack Baskin School of Engineering, University of California, Santa Cruz ,2004.

[8] R. Harris. Parallel NFS: Finally. NFS Optimized for Clusters. [Online]. Available: www.datamobilitygroup.com.

[9] B+-TREE. [Online]. Available: http://www.mec.ac.in/resources/notes/notes/ds/bplus.htm

[10] Gluser White Paper: Gluster File System Architecture. [Online]. Available: www.gluster.com

[11] Wikipedia. [Online]. Available: http://www.wikipedia.bplus.html

Ohnmar Aung is now making research related with object-based file system. She is also a phd student as well as a staff of Republic of Union of Myanmar. She got Master of Computer Science in 2009 from University of Computer Studies, Yangon, Myanmar.