Comparison of Various Indexing Techniques For Distributed Object Storage

Vivek Vijaykumar Bajpai
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA
vbajpai@hawk.iit.edu

Vaibhav Uday Hongal
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA
vhongal@hawk.iit.edu

***Abstract*—This paper deals with the comparison and benchmarking, with respect to performance, of some of the well-known indexing algorithms used for distributed file system/distributed object storage using a set of parameters.**

***Keywords*—Distributed File Systems (DFS); Indexing Data Structures; Distributed Hash Table (DHT); B+ Tree; B-Link Tree**

I. Introduction

Keeping the unstructured data available safe and integrated, as the data sets continue to grow in today's world, is a key challenge for today's data centers. With a massive increase in the data to be stored, a distributed approach of storing data using Distributed File Systems (DFS) such as HDFS, Microsoft DFS, Novell Netware, Microsystem's NFS, or distributed object storage such as IBM COS, Amazon S3, Microsoft's Azure storage, became more and more popular. Only raw data distributed among multiple machines is of no use and one needs to perform different operations like search, listing, prefixed listing, ranged reads, ranged writes etc. on it.

Generally, an index is maintained, either centralized or distributed, for the data which provides quick access to it and reduces the time required for these operations, drastically. The design of this index is very crucial to the overall performance of the system and hence there are various well known data structures used for maintaining it. In this project, we have compared the performance of different commonly used data structures such as Distributed Hash Tables and B+ trees, based on certain parameters and propose the use of B-link trees for indexing.

II. Motivation

Indexing is importatnt because most of the data in object storage is unstructured and searching the entire namespace for a particular object is infeasible. There are many readymade solutions which are used for indexing of the file data in the DFS or OS, but most of them are not tailor made for this purpose.

Using a relational database is one of the option. Although, it is best suited for searching and gives very good read performance, the updates are pretty expensive, as relational databases are designed using OnLine Transaction Processing (OLTP) in mind, which reduces performance. Distributed NoSQL databases like Mongo DB and Couch are preferred to support availability, partition tolerance, and performance over consistency, which is pretty bad in case of a DFS. Traditional distributed hash tables are another common method for such an index, but does not make optimum use of principal of data locality. B+ trees helps us overcome that, but it again follows a "lock and write" approach and hence does not allow concurrent operations, eventually reducing the performance.

B-link tree is a newly introduced data structure which can be used for this purpose. It provides us best of both worlds i.e. concurrent operations and cheap contiguous access. In this project we have analyzed the performance of DHT and B+ tree data structures for indexing and compare them with a custom implementation of B-link tree, on various parameters.

III. Parameters For An Indexing Algorithm

Before we start our discussion about the indexing algorithms, lets see the parameters that are required to be provided by an ideal indexing algorithm.

1. *Low Read Latency*

   Latency is the time between the stimulus and the response. We need to make sure that the dat structure we are choosing should have a decently low read latency as reads will make a big chunk of the operations performed on an index.

2. *Low CAS operation/write Latency*

   CAS stands for compare and swap operations. To provide consistency, the index entries generally comprised of a revision number which is compared before each and every write, which replaces the normal write operation with a CAS operation.

3. *Low Delete Latency*

   We need to have a low delete latency as well, as the delete operation is performed on an index quite frequently.

4. *Cheap Prefixed Listing*

   Prefixed listing allows us to list all the keys starting with a particular prefix. Generally, the namespace is divided into multiple sub-spaces using these prefixes, and allow us to use the same benefits provided by the directory structure in a file system.

5. *Provide Concurrent Reads*

   The data structure should allow multiple processes to perform reads on the index and all of them should have the same view.

6. *Provide Concurrent Writes*

   Ability to perform concurrent writes by multiple processes is quite important as long as they are writing it to separate entries in the index.

7. *Provide Immediate Consistency*

   The index should be immediately consistent after the changes. Some of the data structures allow eventual consistency and in the meantime, provide different views to different processes, which is not acceptable for an index structure.

8. *Never Present A Corrupt View of Index*

   The index data structure should be resistant to data corruption as corrupted data in the index structure will lead to unusable orphan objects.

9. *Serializable*

   As the size of the data could grow very large, and could be too large to keep in memory all the time, we may need to serialize it and keep it on some secondary storage devices. The required data structure should be easily serializable for this reason.

10. *Easily Distributable*

    As the object storage is a distributed system, we will need a data structure which is easily distributable and will hence provide the same performance, availability and scalability as the object storage itself.

IV. Indexing Data Structures

A. *Distributed Hash Tables (DHT)*

Distributed Hash Tables are a method of hash table lookup over a decentralized distributed file system network. A distributed hash table (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table: (key, value) pairs, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures. The nodes in a DHT are connected through some underlying topology. DHTs use a structured key-based routing approach to provide the decentralization of Gnutella and Freenet, and the efficiency and guaranteed results of Napster. Most modern DHTs (e.g., CAN, Chord, Pastry, and Chimera) strive to provide: decentralization, scalability, and fault tolerance. Fig 1. below shows the simple DHT structure.

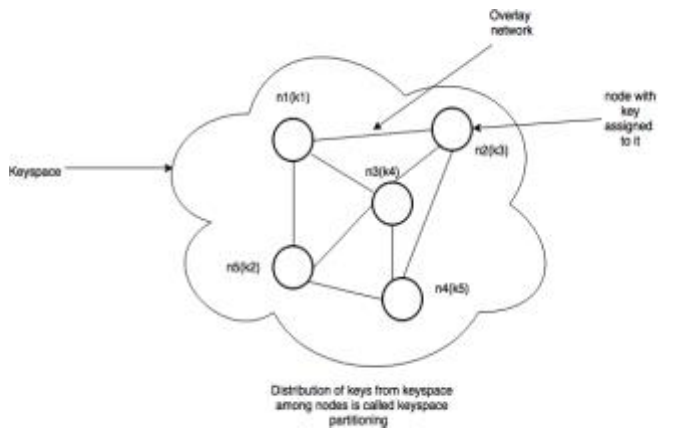

Overlay network
n1(k1)
node with key assigned to it
n2(k3)
Keyspace
n3(k4)
n5(k2)
n4(k5)
Distribution of keys from keyspace among nodes is called keyspace partitioning

Fig. 1. Simple DHT

DHT has the following main components:

1. *Keyspace*

The keyspace is the description of the keys to be associated with data in the network. A common keyspace is a 160-bit string, equivalent to 20 bytes or 480 hex digits. A hash algorithm, often SHA-1[1], is used to convert a file's key k into a hash value. Next, a request is sent to any node in the network asking to store the file with key k. The request is propagated through the network until a node responsible for k is found. Subsequent requests to get the file with key k follow a similar pattern to locate and retrieve the file.

2. *Keyspace Partitioning*

Keys are assigned to a node using a keyspace partitioning algorithm. It is critical that partitioning algorithm support adding and removing nodes efficiently. In a basic hash table, adding or removing nodes would increase or decrease the table size , requiring all the data to be re-hashed. Incase of DHTs, a form of "constant hashing" is used. Constant hashing is a technique where adding or removing one slot in the hash table does not significantly change the mapping of keys to table locations.

For example, suppose we treat keys as points on a circle. Each node is assigned an identifier i from the keyspace. Node i is then responsible for all keys "closest" to i. Suppose two neighbouring nodes have IDs i1 and i2. The node with ID i2 will be responsible for all keys with hash value h | i1 < h < i2. When a node is added, it is assigned an ID in the middle of an

[1] The SHA-1 "secure hash algorithm," designed by the NSA and published by NIST, generates 160-bit hash values. SHA-1 has recently become vulnerable to collision attacks, and its use as a cryptographic key is being retired by many companies.

existing node's range. Half the existing node's data is sent to the new node, and half remains on the existing node. No other nodes in the network are affected. Similarly, when a node leaves the network it gives its data to its predecessor or its successor node.

3. *Overlay Network*

Finally, some communication layer or overlay network is needed to allow nodes to communicate in ways that support systematically locating a node responsible for a particular key. DHT topologies can differ, but they all share a few common properties. First, a node normally only has links to some number of local neighbours. A node does not have a view of the entire topology of the network. Second, every node will either be responsible for key k, or it will have a link to a node closer to k. This allows a greedy algorithm to simply "walk towards" k from any node until k is found. Beyond this basic key-based routing, a few other properties are desirable. In particular, we want to enforce constraints so that the maximum number of neighbours for any node is low, and the maximum number of hops to find a node responsible for k is also low. Common values for both are O(log n).

Chord-DHT

As a practical example, we have examined the Chord system. Chord system has one sime goal – given a key k, Chord will return a node responsible for k. The below figure shows a Chord system.

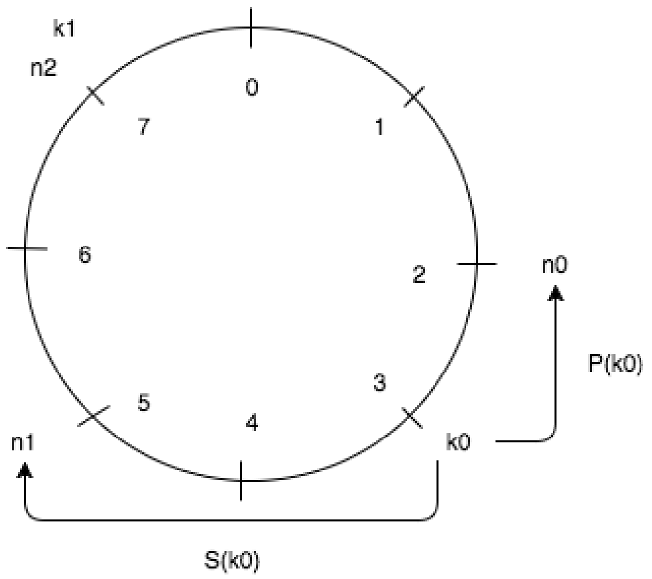


Fig. 2. Chord System'c circular space with m = 3 bits, generating 2^m = 8 nodes

Chord Keyspace: Chord's keyspace is a 2m collection of m - bit keys. Both keys and nodes are assigned positions on the keyspace circle using the SHA-1 hash function (Fig. 13.2). A key hash directly to its position. A node hashes its IP address to get its position.

Chord Keyspace Partitioning: The keyspace circle contains 2^m positions arrayed clockwise from 0 to 2^{m-1}. The successor S (k) of a key k at position p_k on the circle (i.e., in the keyspace) is the first node n_i at position p_i such that $p_i > p_k$, that is, the first node clockwise starting from k's position. The predecessor P(k) at position p_{i-1} is the first node n_{i-1} counterclockwise starting from just before k's position with $p_{i-1} < p_k$.

For example, the keyspace in Fig. 2. above uses m = 3 bits to generate $2^3 = 8$ positions. Key k0 hashes to position $p_{k0} = 3$. The successor of k0 is node n_1 at position $p_1 = 5$, and the predecessor of k0 is node n_0 at position $p_0 = 2$. Key k1 at position $p_{k1} = 7$ has successor and predecessor S (k1) = n_2 and P(k1) = n_1, respectively. Key k is assigned to the first node n_i at position p_i such that $p_i \geq pk$, that is, k is assigned to its successor S (k). In the example in Fig. 2. k0 is assigned to S(k0) = n_1.

Chord Overlay Network: To support both guaranteed and efficient lookup, every node n_i maintains a local routing table with the following information.

- P(n_i)
- a finger table

The finger table is a routing table with (up to) m entries pointing to nodes further clockwise along the keyspace circle. The jth entry of the finger table contains the first node n_j that succeeds n_i by at least 2^{j-1} positions. The finger table allows each request to jump about half as far as the previous request, so it takes O (log n) jumps to find S (k).

Algorithm:

1. Use consistent hashing to assign the keys to the nodes.
2. Node picks up its neighbors depending on the underlying topology.
3. At each step, the message containing key and metadata is forwarded to the neighbor whose id is closest to the key.
4. When there are no such neighbors, the key is arrived at the closest node, which then becomes the owner of the key.

DHT and Parameters

DHT satisfies all the parameters required for an ideal indexing data structure except one.

✔ Low read latency
✔ Low CAS operation latency
✔ Provides concurrent reads
✔ Provides concurrent writes
✔ Provides immediate consistency
✔ Never present a corrupt view of the index
✔ Serializable for disk storage
✔ Is easily distributable

The only problem with DHT is expensive prefix listing. A prefix search/listing is nothing but all entries in the table that contain text beginning with the specified prefix will be returned. DHTs only directly support exact match queries i.e. fixed search for an object given its key. Whereas, prefixed listing is difficult to implement and an expensive operation in DHT. This is because DHTs use hashing to distribute keys uniformly and can't rely on any structural properties of the key space, such as an ordering among keys. For instance, if we search for a key starting with say "ab", all the keys starting with "ab" are not necessarily located in the same network region and hence we will have to span the entire hash table to find all the matching values which is expensive operation.

B. B+ -Trees

B+ -tree is a balanced tree used to maintain a set of keys and a mapping from each key to its associated data. A B+ -tree consists of a root, internal nodes and leaves. In contrast to B-tree, only the leaves in a B+ -tree hold the keys and their associated data. The data of the keys in the internal nodes is used to allow navigating through the tree i.e. it contains the pointers to the descendants of the internal node. Here, a tree search starts with the root and chooses a descendant according to the value of the keys. A simple B+ -tree can be seen in figure below.

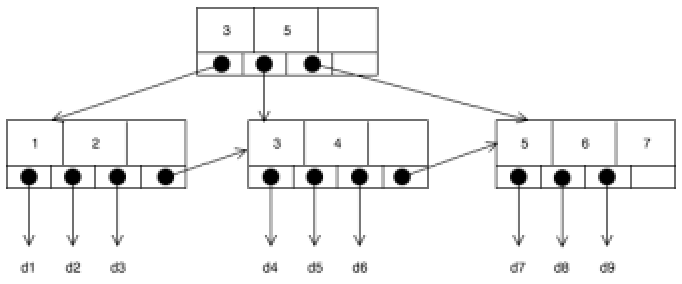


Fig. 3. B+ Tree
Keys: 1 through 7
Data values: d1 through d9

B+ -trees differ from B-trees in one aspect, namely that no data resides in the interior nodes of the tree. Since all the data is contained at the level of the leaves, the leaves can be linked together, allowing sequential access to the data once the leaves are reached. This also means that interior nodes contain only referential data, acting as a guide to the information kept at the leaves.

B+-trees distinguish internal and leaf nodes, keeping data only at the leaves, whereas ordinary B-trees would also store keys in the interior. B+-tree insertion, therefore, requires managing the interior node reference values in addition to simply finding a spot for the data, as in the simpler B-tree algorithm. Deletion in B+-trees, as in B-trees, is precisely the converse of B-tree insertion. If a node falls below its minimum number of entries after the deletion, its neighboring nodes are checked. If they have more than the minimum number of keys, a fraction of the surplus keys from the larger neighbor are redistributed to the node. Only if both neighbors are minimal in size are nodes merged together.

Algorithm:

```
function search(k):
    return tree_search(k, root);

function tree_search(k, node):
    if node is a leaf then
        return node;
    switch k do:
        case $k \le k\_0$
            return tree_search(k, p_0);
        case $k\_d \le k$
            return tree_search(k, p_[d+1]);
        case $k\_i \le k \le k\_[i+1]$
            return tree_search(k, p_[i+1]);
```

B+ Trees and Parameters

B+ trees overcome the problem with DHTs and provide the following:

- ✔ Low read latency
- ✔ Low CAS operation latency
- ✔ Provides concurrent reads
- ✔ Provides immediate consistency
- ✔ Never present a corrupt view of the index
- ✔ Serializable for disk storage
- ✔ Is easily distributable
- ✔ Cheap prefix listing

However, B+ trees have a problem of their own. Locking mechanism locks the entire path of the node that is being modified. This disallows other processes to concurrently modify the same node. Following figure shows this with an example.

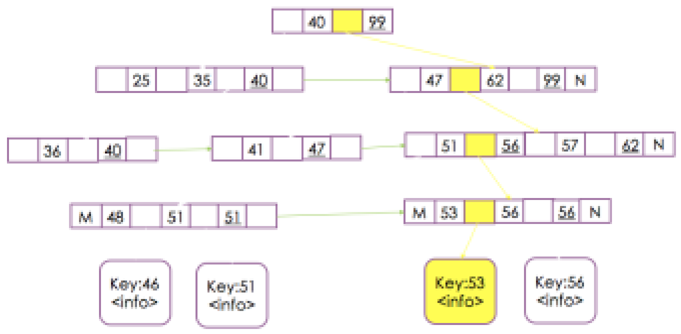


Fig. 4. B+ Tree Locking

Suppose a process P1 wants to insert/modify the data at key 53, then it locks the entire path from root to the leaf node where data is found, as shown in figure 4 above (highlighted in yellow). If another process say P2 wants to write any data or insert a node which falls in the path locked, it wouldn't be able to do so until lock is release by P1. This disallows concurrent writes in B+ trees.

Previously, many solutions have been proposed for this problem but none of them come without a price. One strategy for minimizing the amount of locking during initial descent is to lock not the whole path starting from the root but only the sub-path beginning at the last 'stable' node (i.e. a node that won't split or merge, as a result of the currently planned operation). Another strategy is to assume that no split or merge will happen. Another trick is to ensure that each node 'descended through' is stable by preventative splitting/merging; that is, when the current node would split or merge under a change bubbling up from below then it gets split/merged right away before continuing the descent.

C. B-link Trees

The B-link tree is a modified B* tree with a single "link" pointer to its next sibling (next node on the same level), present in all the leaf and the intermediate nodes. We also add an additional key to each and every node called as "max key" which is used to denote the maximum key which could be present in that particular node. It is used in the search operation explained later. The B-link tree helps us overcome the locking problem we faced in the B+ tree. The following diagram represents a typical B-link tree.

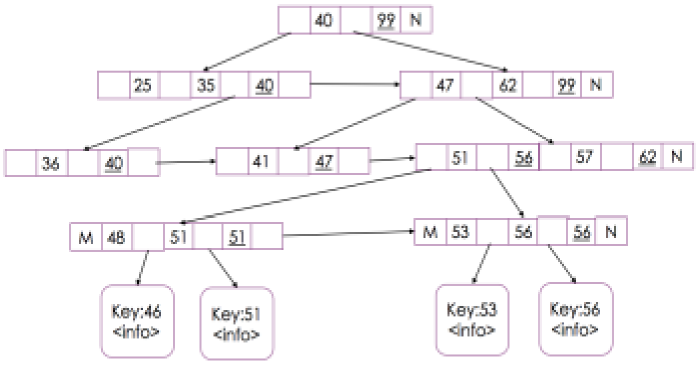


Fig. 5. B-link Tree

Insertion (Better locking mechanism)

The way insertion works in B-link trees eliminate the problem in B+ tree where we need to lock the entire path to the leaf node to perform insertion/deletion on that node. In B-link tree, the Write lock is restricted to the node in which the key is being written/deleted.

For example, consider the order of the B-link tree shown in Fig. 6, and we need to insert a key "9" in it. We know that the node p is unsafe and will need a split if a new key is inserted. We can acquire a lock on the node p, and perform the insertion. After the split happens, we will end up with two nodes q & q'. We will add a link from q to q' and q' won't have any link from the parent node. In the next step, we will release the lock on the leaf node and acquire a lock on the parent node. This time, there is no change required in q and q' and hence no write lock required. After acquiring the lock on the parent node, we can create a new link from the parent to q', and the max key from the leaf node is inserted in the parent. If the parent is also unsafe and needs splitting, we can back propagate the operation up the tree, until we find a safe node for insertion, or we have reached the root.

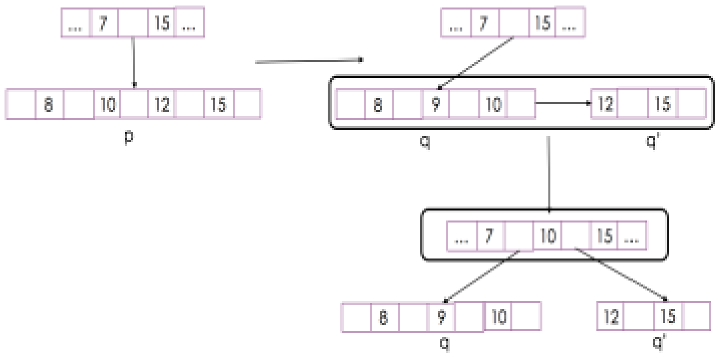


Fig. 6. B-link Tree Insertion

Searching

The searching operation in a B-link tree has some additional complexity due to newly split nodes without a link to their parent node. While searching for a key "v", we make sure that we reach all the way to the leaf node where v may exist. If we reach the end of the node and we see a "max key" which is smaller than v, it indicates that we may have a newly split node linked to this node. We use the pointer in the node and load the next node in the memory. We perform the same operation on this node until we find v or we reach a max key greater than v. The search operation is a little more expensive in B-link tree due to extra disk lookups. The following pseudo code explains the search process.

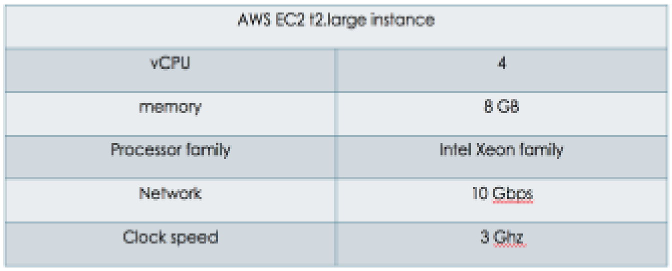
```
Procedure search(v)
        current <- root
        A – get(current)
        while current is not a leaf do
        Begin
                current <- scanNode(v,A)
                A <- get(current)
        end
        while t <- scanNode(v,A) = link ptr of A do
        Begin
                current <– t
                A <- get(current)
        end
        if v is in A then done SUCCESS
        else done FAILURE
```

This way, B-link tree provides us all the parameters which we were seeking for the index structure.

V. Comparison And Benchmarking

In this project, we have benchmarked the above discussed, DHT, B+ Tree and B-Link Tree indexing data structures and follow graphs show the results of this process.

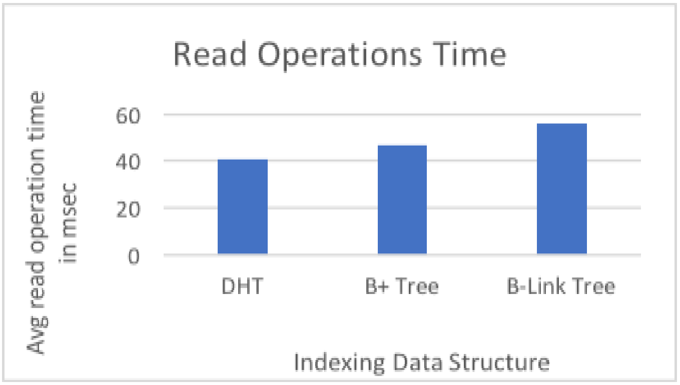
The benchmarking experiment was performed on Amazon EC2 t2.large instance. Table 1 shows the specification of the machine on which the benchmarking was performed.



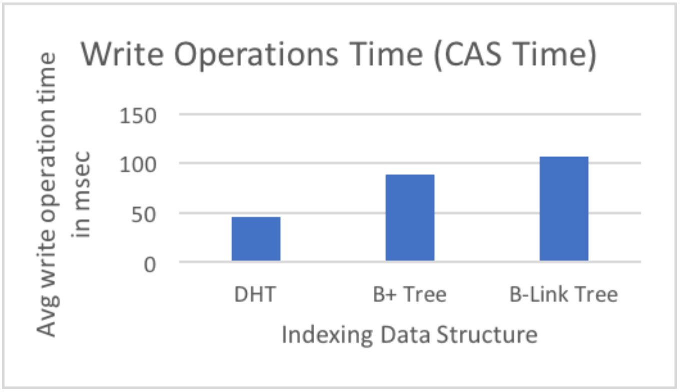
AWS EC2 t2.large instance	
vCPU	4
memory	8 GB
Processor family	Intel Xeon family
Network	10 Gbps
Clock speed	3 Ghz

Tab. 1. Benchmarking machine specifications

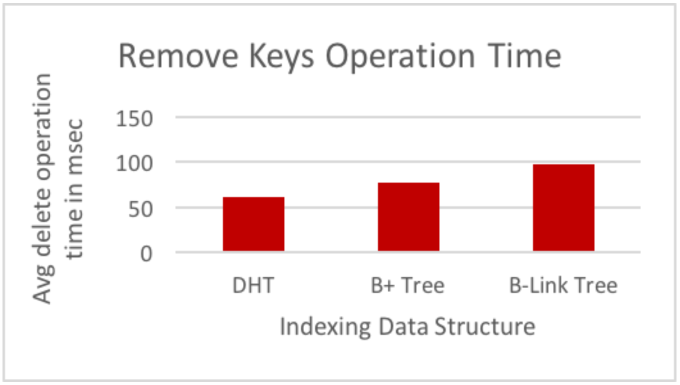
1. *Read Operation Time*



2. *Write Operation Time*

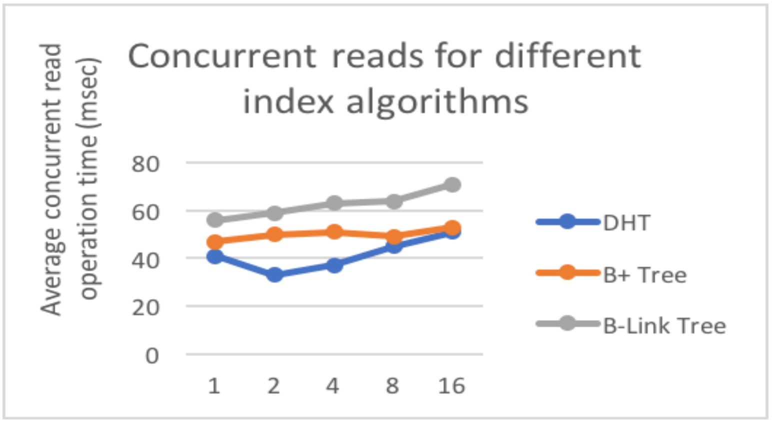


3. *Delete/Remove Operation Time*

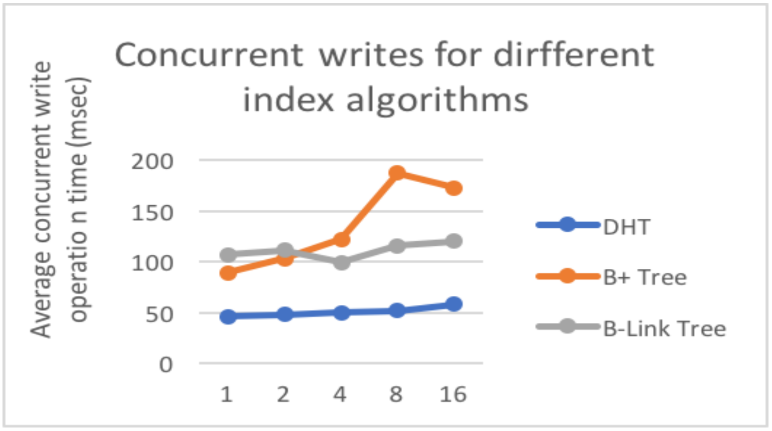


As we see from above three bar graphs, DHT outperforms both B+ trees and B-link trees in terms of read, write and delete operations. But, as discussed in previous sections, it does not provide the prefixed listing/search at low costs and fails there.

4. *Concurrent Reads*



5. *Concurrent Writes*



One major thing to be noticed from above graph is about the concurrent write latency in case of B+ Tree and B-link Tree. As the B+ trees have the problem of locking the entire path during a write/delete operation, the concurrent write latency is high when compared to B-link tree. B-link tree on the other hand outperforms the B+ tree in this case. As we saw in the insertion algorithm, the write lock is applied on only the node which is being modified and all other nodes can be modified independently of this operation. This gives us ability to update the index by different processes concurrently, which improves the write performance multifold.

VI. Conclusion

From the above experiment we can conclude that the B-link tree is a very good candidate for maintaining the index in a distributed object storage system. Although it has a higher read/write latency, but it performs good enough on all the parameters we were seeking for the index data structure. The DHT has the problem of very expensive prefixed listing, which is a deal breaker for it being used for our purpose. Prefixed listing is very important in case of the object storage system. The B+ tree provides us the benefits of prefixed listing and serial access of the keys, but it has serious locking problem which rules it out for use in a distributed, multi-threaded environment. The B-link tree overcome both these issues. An additional advantage of using B-link tree for the index structure is that when the tree is searched serially, the link pointer is useful for quickly retrieving all the nodes in the tree in “level-major” order. The B-link tree is also easily serializable and distributable as the different sub-trees can be split and saved on different machines, and can be joined back when needed.

VII. References

[1] Philip L Lehman and S. Bing Yao, **Efficient Locking for Concurrent Operations on B-Trees,** ACM Transactions on Database Systems, Dec 1981.

[2] Itua Ijagbone, Scalable Indexing and Searching on Distributed File Systems, May 2016.

[3] Andrew W Leung, **Organizing, Indexing and Searching Large scale file systems,** Technical report UCSC-SSRC 09-09 December 2009.

[4] Jerome H. Saltzer, **File System Indexing and Backup,** Laboratory Of Computer Science, MIT.

[5] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan, **Chord: A Scalable Peertopeer Lookup Service for Internet Applications,** Laboratory Of Computer Science, MIT.

[6] Lars Seipel, Alois Schuette, **Providing File Services using a Distributed Hash Table,** Department of Computer Science, University of Applied Sciences Darmstadt, Germany.

[7] Marcos K. Aguilera, Wojciech Golab, Mehul A. Shah, **A Practical Scalable Distributed Btree**, PVLDB, Auckland, New Zealand, Aug 2008.