Incrementally Distributed B+ Trees: Approaches and Challenges

Pallavi Tadepalli
Overstock.com
6350 South 3000 East
Salt Lake City, UT 84047
1-801-947-3864
patadepalli@overstock.com

H. Conrad Cunningham
University of Mississippi
201 Weir Hall
University, MS 28677
1-662-915-5358
cunningham@cs.olemiss.edu

ABSTRACT
B^+ trees have proven efficient and effective in the role of indexes for data stored in databases. With the explosion in the number of datasets being stored in a distributed manner, a scalable and efficient index is needed to locate data. In this paper, the issues in designing a distributed B^+ tree are examined with a specific emphasis on incrementally distributing the tree across a network.

Categories and Subject Descriptors
E.1 [**Data Structures**]: Distributed data structures

General Terms
Algorithms, Design.

Keywords
Distributed B^+ tree, Grid Nodes

1. INTRODUCTION
Data are ubiquitous. Data sets may be very large and either concentrated at a single site or spread across a network that could be a grid, peer-to-peer, sensor or some other kind. Thus there can be several independent data sources that can be accessed without a central authority. When there are distributed data sources in a network with limited or no central authority, searching for the appropriate data sources that contain relevant data is an issue. Thus the issue can be described as a problem of data location. To address the issue there needs to be a comprehensive index designed and implemented as a modified, distributed B^+ tree. This can be used as a scalable indexing and searching mechanism to locate distributed data. To distribute the B^+ tree, replicating a part of the tree on various grid nodes is essential along with keeping a low overhead in maintenance of various copies.

2. PROBLEM DEFINITION
A distributed data structure is effective in "managing large volumes of distributed and dynamically changing data" [14]. According to [4], the problem of data location on a grid can be solved by an efficient index that is created using complex distributed data structures that are variants of the B-tree [2] like B* trees. If an index was created for each data item spread over hundreds of nodes on a grid, the size of the index would be huge. Essentially it is like creating an index for a very large database. B and B^+ trees have been used to create traditional database indexes. These indexes are generally used to search for a record or range of records given a particular key. Storing an index of such a magnitude in a central location creates an access bottleneck. In distributed computing with large volumes of dynamically changing data, high-throughput access is essential. Distributing a B^+ tree enables it to handle concurrent data access requests in parallel which could lead to higher throughput. The assumption is that the results will be combined at the requesting "client" site.

By distributing a search tree, increased parallelism on various operations is expected. Yet, the throughput is limited by the single rooted structure of the tree since all operations at the root of the tree. This scenario is known as the *root bottleneck*. By replicating the root and other nodes of the search tree there should be an increase in efficiency, decrease in latency and improvement in performance. In a distributed environment, multi-user access to the tree can cause concurrency issues such as contention management which are solved by the use of concurrent algorithms. The manner of distributing the search tree could affect node placement, the number of replicas needed and the manner of replication. Once tree nodes are replicated, it is necessary to keep track of the various tree nodes and copies. Thus node replication and replica coherence are inherent requirements in a distributed structure to support concurrent operations.

3. BACKGROUND

3.1 Distributed Indexing Structures
The problem of indexing distributed data in a decentralized network has been studied in many different ways and there have been many attempts to solve the problem in various distributed storage systems using a diversity of distributed data structures.

An existing distributed data structure is the Distributed Hash Table (DHT) [18]. DHTs were invented as a result of the widespread growth and use of peer-to-peer file sharing systems like Gnutella [7], Napster [17] and KaZaa [12]. The goal of DHTs has been to provide efficient location of data sources. Given a

ACMSE'09, March 19–21, 2009, Clemson, SC, USA.

key, the data item can be located in O(log N) network messages[20, 21] where N is the number of nodes. DHTs actually represent various scalable location and routing protocols like Content-Addressable Network (CAN) [19], Pastry [20], Chord [21] and Tapestry [23, 24].

While DHTs have proven advantages of being simple, scalable and supportive of fair load distribution [5], they have been used to build spanning trees for traversals without keeping any of the features of DHT's themselves. Although DHTs are scalable, they are deficient in supporting range queries due to the underlying hashing property [18].

DHTs are in contrast to trees that are capable of adapting to the data distribution, supporting range and nearest neighbor queries, maintaining the keys in order and performing well in the worst case [15].

Range queries are important in several application domains like distributed query processing in peer-to-peer databases, where support for distributed SQL-type relational queries is essential, resource location in a decentralized manner in distributed computing, and range queries in location-aware and scientific computing [18]. Tree structures such as B^+ trees support the range predicates (<, = >) thus allowing range queries [8].

Suitable tree indexing structures are multi-way trees, more commonly B and B^+ trees [2]. Figure 1 shows a snapshot of a B^+ tree. B and B^+ trees have been de facto standard indices for file organization and databases. The original B-tree algorithms were designed to minimize access latency [13]. For distributed computing, high throughput access is essential and distributing the tree can improve efficiency while solving the issue of scalability. This will also lead to reduction in transaction processing time [14].

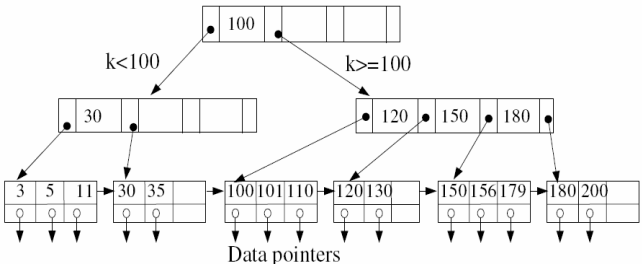

Figure 1. Snapshot of a B^+ Tree [1].

By distributing the B^+ tree there is an attempt to increase the ease and efficiency of data access and reduce communication costs. This leads to multiple concurrent requests instead of parallel execution of a single request [3].

3.2 Other Distributed B-Tree Algorithms

There has been significant prior work on distributed search structures implemented using B^{link} trees as concurrent or shared memory implementations. A B^{link} tree is a B^+ tree with every node containing a pointer to its right sibling [9]. Johnson and Krishna used B^{link} trees to build a dB tree with replicated index nodes across a message-passing multiprocessor architecture to improve parallelism and alleviate the bottlenecks [9]. This work was the extension of the ideas presented in [10] where replicated interior nodes and local restructuring decisions that minimized communication overhead, led to enhanced parallelism. In [9], the authors propose lazy update algorithms as a cost reduction mechanism for maintenance of replicated copies. A lazy update algorithm may "block an operation until the local data is sufficiently up-to-date" [13]. These are concurrent algorithms that avoid synchronization, making them easy implementations.

Wang and Weihl [22] proposed parallel B-trees using multi-versioning that can be implemented on shared memory processors and message-passing multiprocessors as well as distributed systems. Their algorithm is more suitable for cache-coherent shared memory and is designed for software cache management [13]. With distributed structures, comes the need for replication and replica coherence. The amount of synchronization among replicated copies is significantly reduced in the multi-version memory algorithm.

The next section describes desirable characteristics of a distributed B^+ tree.

4. DESIDERATA

A distributed B^+ tree index should have the following properties:

Correctness: The dictionary operations of insert, delete and search should be correctly executed in a concurrent manner. The ACID (Atomicity, Consistency, Isolation and Durability) properties should be preserved. Given a key, the search should return the appropriate pointers to the record if it exists or should return nothing if it is not present in the index.

Efficiency: Since B^+ tree nodes are distributed over several grid nodes, an operation may require multiple hops between grid nodes and several remote messages to complete. Efficiency is measured by lower total time, fewer hops between grid nodes and also fewer remote messages for an operation to complete.

Scalability: As the volume of data increases, the size of the B^+ tree will also increase, which leads to an increase in the number of tree nodes and correspondingly more grid nodes over which the tree is spread. Thus as the tree scales to a greater number of nodes, the performance of the index should not degrade too badly. That is, it should be within an acceptable proportion of a scaled-down implementation, preferably logarithmic in the number of nodes.

Load Balance: By distributing a tree, tree nodes are spread over the various grid nodes. No particular grid node should have an excessive number of tree nodes as compared to other grid nodes as it may become a bottleneck in the system due to an imbalance of resources at a particular site [14]. It is possible that due to excessive splits, a grid node may run out of space and, hence, it is desirable that the distribution of the tree nodes be balanced over all the available grid nodes.

Decentralization: The tree is fully distributed over grid nodes, with each grid node being equally important in allowing operations to be performed. Any dictionary operation can begin on any grid node.

Fault Tolerance: If a grid node holding a portion of the tree is down, or if the tree nodes are moved to another grid node, then the operations on the index must be able to continue barring major failures in the underlying network.

Node Replication: For increased efficiency, various tree nodes should be replicated amongst the grid nodes. There is, however, a tradeoff between the space used for replication and the throughput. Node replication should be balanced over the grid nodes and be based on the access structure of the index rather than the tree structure while keeping the overhead of maintenance low. This should improve performance and space usage. Throughput is also affected when tree modifying operations like insertion/deletion are included.

Replica Coherence: The various copies of the tree nodes spread over the grid nodes need to be monitored and updated so that all copies reflect the same values.

The next section describes the issues in building a distributed B^+ tree along with an outline of the authors' approach.

5. DISTRIBUTED B^+ TREE INDEX

5.1 Building and Distribution of the B^+ Tree

To build a distributed B^+ tree across multiple grid nodes, some possible approaches that can be taken as specified below.

1. Build individual B^+ trees on the various grid nodes containing data that is being indexed and then merge them into a single logical structure that is distributed.
2. Build a B^+ tree in a central location using standard B^+ tree algorithms and then break up the tree and spread it across several grid nodes. This could involve some level of replication during the initial breaking up and distributing.
3. Build a B^+ tree incrementally on a single grid node through a series of insertions and gradually spread it over other grid nodes as new data are inserted. In the process of distribution, some initial replication also takes place.

This work takes the third approach, building and distributing the B^+ tree through a series of insertions. This approach is closest to the characteristics of a B^+ tree that is built dynamically. Hence, this approach dynamically builds and distributes a B^+ tree across grid nodes (physical nodes). It builds a concurrent and distributed B^+ tree i.e. the B^+ tree supports concurrent operations. This algorithm, especially the concurrency aspect has been adapted from [16] which modifies the LC-SIX (lock coupling – Shared Intent Exclusive) algorithm as described in their paper. A description of the third approach is provided in section 5.4 below.

5.2 Concurrent Distributed B^+ Tree

The model of this implementation includes a distributed B^+ tree where tree nodes are spread over various data nodes on the grid. Physically B^+ tree nodes are distributed across several grid nodes while appearing as a single logical structure. Leaf nodes contain pointers to the actual records. The root and parts of the tree are replicated.

5.3 Model of the Application

Figure 2 shows the model of the application where a B^+ tree is distributed over four different grid nodes (A, B, C and D). A portion of the tree is located on each of the four grid nodes along with the root being replicated across all the grid nodes. There are pointers from root nodes on each grid node to the portion of the tree being held on a specific grid node. The figure also shows the interaction of the client with the interface that spawns several different threads to call insert and search operations on each of the grid nodes (GNs).

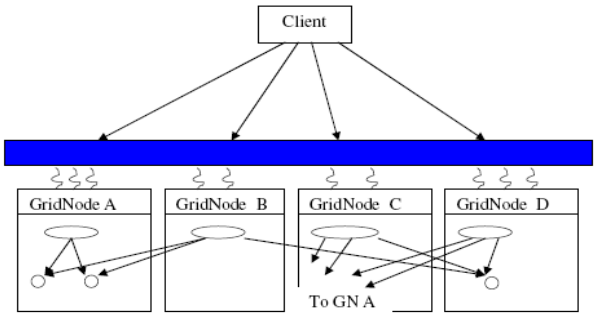

Figure 2. Model of the Application.

5.4 Incrementally Distributed B^+ Trees

The idea behind building a B^+ tree through a series of insertions is to use the same operations in building a B^+ tree to build a distributed B^+ tree.

This is achieved by having insert and split operations behave similarly to those in regular B^+ trees but with certain other side effects necessary for distributing a B^+ tree. The problem of building a distributed B^+ tree can be divided into several aspects.

1. How to distribute a B^+ tree across multiple grid nodes through a series of inserts (and split operations)?

The answer to this lies in the actions that occur during these operations. In the incrementally distributed B^+ tree approach surveyed in this paper, the rules are as follows:

a. one grid node should be designated as the primary grid node. This is important since tree building activity (when the tree is being first built from a single node) begins on this grid node. This is a configurable property and the initial insert of a single key begins by creating a root node on the primary grid node. There are two ways to handle this: i) Either the grid node that receives the first insert of a record sets itself as the primary grid node or ii) the value of the primary grid node is configured in advance and the initial tree building operations commence on that grid node despite being initiated from some other grid node. The pitfalls to the first approach are that, in a concurrent and distributed environment, it is hard to determine the "first" occurrence of an operation and if the same operation (initial insert of a single node) occurred almost simultaneously on two separate grid nodes, correctly determining the primary grid node is not a trivial task.

b. After the initial insert that creates a root node on the primary grid node, the root node is replicated on all other grid nodes. This provides multiple entry points to access the tree without making the primary grid node a bottleneck. Inserts continue on the primary grid node until the tree grows to a stage when an internal node is split. At this stage, distribution begins. Until this stage, while inserts are occurring on the primary grid node, root nodes on other grid nodes are all updated to point to the children on the primary grid node (PGN). Thus the root node on the PGN and the

other grid nodes effectively point to the same child nodes as shown in Figure 3. Thus for each insert, split operation, pointers from other roots need to be updated to point to the correct child nodes. This allows search operations being initiated from any grid node to return the same values. The root on the primary grid node is designated as the primary copy of the root. Designating a specific node as the primary copy has the following effects: i) It identifies the originator of the node. All operations on the node and its copies are first triggered on the primary copy before being propagated to the various node copies. ii) The primary copy of a node keeps track of all node copies and is responsible for maintaining coherence amongst the various copies.

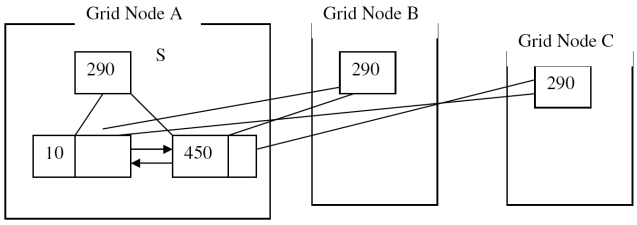

Figure 3. Splitting and replicating the root(leaf) node

c) As mentioned in b, distribution begins when it is determined that an insert operation will cause a split in an internal node. Note that when a root node is first split, a leaf node is being split and this does not count as splitting of an internal node. When an internal node is to be split, the splitting operation is allowed to complete on the original grid node. The newly split internal node now is made up of two nodes N1 (left) and N2 (right). For distribution, the sub-tree rooted at N2 is now moved to the next grid node as shown in Figure 4. Determination of the "next" grid node may be random or it may be the physically closest grid node to the current grid node. By current grid node, it means the grid node on which the splitting of the internal node takes place.

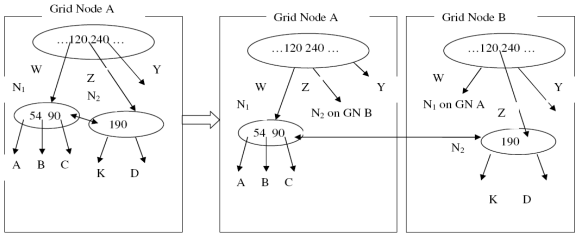

Figure 4. Distribution after splitting of an internal node

2. What nodes of the B^+ tree should be replicated on various grid nodes in the process of distributing the B^+ tree, to make it efficient and scalable?

This is addressed in Section 5.5.

3. What is the coherence protocol used to manage the various tree node replicas?

The need for coherence amongst nodes and their copies arises in two node types:

i. Root node and its copies on various grid nodes
ii. Internal nodes and their copies on various grid nodes.

There are three reasons that coherence among nodes is an issue:

a. Splitting of an internal node
b. Insertion into a root node
c. Insertion into an internal node

Each of these requires updates to take place on the various copies of the affected nodes. Whenever a node and/or its copies are to be updated, the update is assumed to be an atomic action. It is not necessary to immediately update every copy every time a node changes. Thus the updates can be *lazy*. For lazy updates to succeed it is necessary to determine the commutative relationships amongst operations.

The commutative relationships between various operations are as follows [9]:

a) Insert operations are commutative.
b) Splitting is not commutative since the final value of a copy node will be dependent on the order of the processing of the split operations.
c) Relayed split operations commute with relayed inserts but not with performed initial inserts. This is because there is a danger of losing an insert if the split causes the removal of an item from the range. Relayed operations are operations that were passed on from another node to be performed locally on the current node. Initial operations are those that were invoked directly on the current node.
d) Initial split operations and relayed insert operations do not commute because splitting causes the creation of a new sibling. The inserted key may or may not appear in the sibling depending upon whether it occurs before the split or not.

5.5 Node Replication

Distributed search structures invariably have replication of nodes. In [13] the authors propose path replication and full replication as two different strategies. In their observation, path replication had lesser message and space overhead thus improving scalability of distributed B-trees. In path replication, each processor that has a leaf node holds all the nodes that are on the path from leaf to the root node. Thus the number of copies is dependent on the placement of leaf nodes within a processor. Johnson and Colbrook suggest keeping neighboring leaf nodes on the same processor as much as possible [3]. They developed a placement algorithm to reduce the number of copies of internal nodes that must exist, thereby minimizing the number of inter-processor messages required.

In [3], Cosway contends that while the path-to-replication method ensures equal processing capacity for each level of the tree, its distribution pattern is fixed based on tree structure instead of access structure. This leads to poor performance under non-uniform access patterns when insertions and deletions take place. Cosway proposed a rule that calculates replication recommended by the capacity balancing rule and adjusts the actual replication based on the threshold value. It is shown that this rule specified by Cosway is more useful than path-to-root model especially when conditions are not ideal in the real world. It "provides more copies for frequently used nodes high in the tree and fewer copies for less frequently used nodes lower in the tree"[3]. Based on Cosway's work it was determined that an ideal replication-control algorithm should replicate only those tree nodes as determined by their frequency of access measured over time and dynamically react to changes in access patterns and produce low overhead. This can also be achieved by creating a "balanced distribution of

nodes and copies instead of relying on random placement" [3]. Thus replication-control becomes a caching problem.

The incrementally distributed B^+ tree approach to creating a replication-control algorithm includes determining the frequency of access to specific tree nodes. In the process, infrequently accessed replicas are deleted when they occupy too much space or when newer replicas need to be created to adjust to changes in the frequency of access. In such a scenario, the least recently used page replacement algorithm is being adapted from cache optimizing algorithms to select the least recently used replica that is occupying space useful to another replica. The next subsection describes this in more detail.

5.6 Caching in the Distributed B^+ Tree

As ascertained in the previous section that replication control is basically a caching problem, this work has attacked this problem by establishing two levels of caching.

The first level of cache is created by using unused area in a B^+ tree internal node as a cache area. This cache may be referred to as a "persistent cache" because of the persistent nature of the B^+ tree itself [11]. The size of the cache is inversely proportional to the number of keys in the node. Thus as the number of keys in the node increases, the size of the cache decreases.

In addition to the persistent caching area of the B^+ tree internal node, there is a cache located on each grid node (GN) that will be persistent within that grid node. This acts as the second level of the cache. The size of the cache on the GN should be large enough to hold a certain minimal number of items that could be a percentage of the number of keys in the leaf nodes of that GN. By default, every time the user invokes a search operation and requests a key on a grid node (GN), the {key, data pointer} item is placed on the GN cache along with a timestamp. (Note: A {key,data pointer} item consists of a key and pointer to some data outside the tree. This item usually resides in leaf nodes.) If the cache is full, an item is removed based on the combination of the least recently used (LRU) and least frequently accessed algorithms. An access counter is associated with the {key,data pointer} item stored on the GN cache and it is incremented whenever it is requested if the time elapsed since the previous access is less than the time lag. If the time between accesses is greater than the time lag, the access count is decremented until it reaches zero. Once the access count is at zero, the item becomes a target for removal from the cache based on a combination of least recently used (LRU) and least frequently accessed algorithms. If the {key, data pointer} item is originally located on a leaf node residing on the same GN, then when the access count reaches a threshold value, the {key, data pointer} item is moved into the first available node from root downwards that has a persistent caching area available on that GN. If there is no persistent caching area available even after traversing all the way down to one level above the leaf level, then that {key,data pointer} item is retained on the GN cache with an incremented access count and the size of the GN cache is incremented.

The reasoning for putting items in the persistent cache from root node downwards is that a B^+ tree generally gets filled bottom up especially when only insert operations are being considered or even when there are likely to be more insert operations compared to delete operations. As a result the nodes higher up in the B^+ tree are likely to have more space in the persistent cache. When the persistent caching area in the internal node runs out because of the number of keys is degree (d) minus 1, the items in the cache are selected randomly and moved to the GN cache with access count set to 1. The random choice of the item to be moved to the GN cache is necessitated by the structure of the tree node cache that can only hold the {key, data pointer} item since it is using the space reserved for actual {key, pointer} pairs. (Note: A {key, pointer} item consists of a key and a pointer to a tree node. This item generally resides in internal nodes.) As a result no extra attributes such as access counts and timestamps can be stored within the tree node cache. If a cached {key, data pointer} item is deleted from the leaf node, then a message is broadcast to the GN cache and the B^+ tree root that the item has been deleted. Thus all corresponding cached values are invalidated. On a separate thread, the item is deleted from the tree node cache and GN cache.

6. CONCLUSION

Data grids are increasingly being used in the modern day in both businesses and scientific communities. Data are stored in data sources that are geographically dispersed, in flat files, relational tables, persistent object structures or XML files [6]. The problems that are being addressed here are data location (how to find the data). This involves distributing and replicating B^+ tree nodes across several physical nodes and update propagation (propagating inserts and deletes to various replicas of a tree node). Previous work on distributed B-trees has provided insights on concurrent algorithms, data balancing algorithms and, at some level, the node replication issue. This work addresses the various issues in design and implementation of a distributed data structure like a distributed B^+ tree and outlines the authors' approach to tackle the problem.

7. ACKNOWLEDGMENTS

This work was proofread by Ravi Darbhamulla who provided valuable suggestions to improve the paper. This work was supported by computing resources provided by the Department of Computer and Information Science, University of Mississippi.

8. REFERENCES

[1] Amiri, K., More on Indexing: B^+ trees, Microsoft PowerPoint Presentation, Imperial College, London. Available at: http://www.doc.ic.ac.uk/~amiri/advDB/Btrees.pdf, Last Accessed: 20 November 2008.

[2] Comer, D. The Ubiquitous B-tree. *ACM Computing Surveys*, Vol. 11, No. 2, pp. 121-137, 1979.

[3] Cosway, P. Replication Control in Distributed B-Trees, Master's Thesis, Technical Report MIT/LCS/TR-705, MIT Laboratory for Computer Science, 140 pages, February 1997. Available at: ftp://ftp-pubs.lcs.mit.edu/pub/lcspubs/tr.outbox/MIT-LCS-TR-705.ps.gz

[4] Crowcroft, J.A., Hand, S.M., Harris, T.L., Herbert, A.J., Parker, M.A., Pratt, I.A. FutureGRID: A Program for Long – Term Research into GRID systems Architecture. In *Proceedings of UK e-Science All Hands Meeting*, pp. 21, 2-4 September, 2003. Available at: citeseer.ist.psu.edu/crowcroft03futuregrid.html.

[5] Dahan, S., Nicod, J., Philippe, L. The Distributed Spanning Tree: A Scalable Interconnection Topology for Efficient and Equitable Traversal, In *Proceedings of 5th International Symposium on Cluster Computing and the Grid*, CGRID05, Cardiff, UK, CD-ROM, 8 pages, IEEE Press, May 2005.

[6] Foster, I.T., Vöckler, J., Wilde, M., Zhao, Y. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation, In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pp. 37-46, 24-26 July, 2002.

[7] Gnutella Resources, http://www.gnutella.com, OSMB, LLC, 2001.

[8] JMH. GiST: A Generalized Search Tree for Secondary Storage, Available at: http://gist.cs.berkeley.edu/ Last Modified 06 August 1999.

[9] Johnson, T., Krishna, P. Lazy Updates for Distributed Search Structure, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* , Vol. 22, No. 2, pp. 337-346, June 1993, Washington D.C., USA.

[10] Johnson, T., Colbrook, A. A Distributed, Replicated, Data-Balanced Search Structure. *International Journal of High Speed Computing,* Vol. 6, No. 4, pp. 475-500, 1994.

[11] Kato, K. and Masuda, T. Persistent Caching: An Implementation Technique for Complex Objects with Object Identity, *IEEE Transactions on Knowledge and Data Engineering,* Vol. 18 No. 7 pp. 631-645, July 1992.

[12] Kazaa. http://www.kazaa.com, Sharman Networks, 2002-2005.

[13] Krishna, P., Johnson, T. Implementing Distributed Search Structures. Technical Report TR94009, University of Florida, 1992. Available at: citeseer.ist.psu.edu/krishna92implementing.html.

[14] Krishna, P.A., Johnson, T. Highly Scalable Data Balanced Distributed B-trees. Technical Report 95-015, University of Florida, Dept of Computer and Information Science and Engineering, 1995. Available at ftp.cis.ufl.edu/tech-reports, http://citeseer.ist.psu.edu/article/krishna95highly.html.

[15] Kroll, B., Widmayer, P. Distributing a Search Tree Among a Growing Number of Processors. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) International Conference on Management of Data*, pp. 265-276, 1994.

[16] Lim, S., Kim, M.H. Restructuring the Concurrent B+ tree With Non-Blocked Search Operations, *Information Sciences – Informatics and Computer Science: An International Journal*, Vol.147 No.1-4, pp. 2002.

[17] Napster. http://www.napster.com, Napster, LLC, 2003-2006.

[18] Ramabhadran, S., Ratnasamy, S., Hellerstein, J.M., Shenker, S. Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables, In *Proceedings of the Twenty third Annual ACM Symposium on Principles of Distributed Computing(PODC*), St Johns, Newfoundland, Canada, 2004.

[19] Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S. A Scalable Content-Addressable Network. In *Proceedings of the ACM Special Interest Group of Communication (SIGCOMM) Conference*, 2001.

[20] Rowstron, A., Druschel, P., Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.

[21] Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM Special Interest Group on Data Communication (SIGCOMM) Conference*, pp. 149–160, San Diego, California, 27-31 August 2001.

[22] Weihl, W.E., Wang, P. Multi-Version Memory- Software Cache Management for Concurrent B-trees. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing,* pp. 650-655, December 1990.

[23] Zhao, B. Y., Kubiatowicz, J. D., and Joseph, A.D. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing., University of California, Berkeley, Technical Report CSD-01-1141, 2001.

[24] Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., and Kubiatowicz, J. Tapestry: A Resilient Global-scale Overlay for Service Deployment *IEEE Journal on Selected Areas in Communications,* Vol. 22, No. 1, pp. 41-53, January 2004.