# Speeding up Search in Peer-to-Peer Networks with A Multi-way Tree Structure

H. V. Jagadish [*]
University of Michigan, USA
jag@eecs.umich.edu

Beng Chin Ooi [†]
National University. of
Singapore, Singapore
ooibc@comp.nus.edu.sg

Kian-Lee Tan [†]
National University of
Singapore, Singapore
tankl@comp.nus.edu.sg

Quang Hieu Vu [†]
National University of
Singapore, Singapore
hieuvq@nus.edu.sg

Rong Zhang [†]
Fudan University, China
rongzh@fudan.edu.cn

## ABSTRACT

Peer-to-Peer systems have recently become a popular means to share resources. Effective search is a critical requirement in such systems, and a number of distributed search structures have been proposed in the literature. Most of these structures provide "log time search" capability, where the logarithm is taken base 2. That is, in a system with $N$ nodes, the cost of the search is $O(log_2 N)$.

In database systems, the importance of large fanout index structures has been well recognized. In P2P search too, the cost could be reduced considerably if this logarithm were taken to a larger base. In this paper, we propose a multi-way tree search structure, which reduces the cost of search to $O(log_m N)$, where $m$ is the fanout. The penalty paid is a larger update cost, but we show how to keep this penalty to be no worse than linear in $m$. We experimentally explore this tradeoff between search and update cost as a function of $m$, and suggest how to find a good trade-off point.

The multi-way tree structure we propose, BATON*, is derived from the BATON structure that has recently been suggested. In addition to multi-way fanout, BATON* also adds support for multi-attribute queries to BATON.

## 1. INTRODUCTION

Peer-to-peer (P2P) systems have become very popular of late, and are widely used for sharing resources, such as music files. Search is a crucial operation in P2P systems, and there has been considerable recent work in devising effective distributed search techniques. The proposed structures include a ring as in Chord [13], a multiple dimensional grid as in CAN [18], a multiple list as in SkipGraph [4], or a tree as in BATON [11]. Most search structures, including all the ones just mentioned, bound the cost of the search to a logarithm of the search space: for a system with $N$ nodes, the search cost is bounded at $O(logN)$. However, upon closer examination we see that this logarithm is to base 2. If we could take the logarithm to a larger base, we could have a substantially smaller cost.

The importance of having a large base for the logarithm has been well-recognized in centralized database systems. When tree structures are constructed for indexing, the fanout is made as large as possible, within the limits of the page size. We always use B-trees in database systems, and not binary trees! An equivalent transformation, from the low fanout structures currently proposed in the literature, to practical high fanout distributed search structures, is the topic of this paper.

At first glance, one may think that it is not too hard to increase the fanout in any distributed search structure. For instance, consider Chord. Rather than doubling successive ranges, we can increase them by a factor of $m$; correspondingly, in the finger table we would have $m-1$ entries at each level, instead of just one entry. With this simple change, the search cost (in number of hops required) has gone down by a factor of $log_2 m$. The storage requirement, for the finger tables, has increased by a factor of $m/log_2 m$, but the amount of storage for finger tables is usually not large enough that this matters. So we appear to have won. The difficulty is that the update cost has gone up by a factor of $(m/log_2 m)^2$. This quadratic dependence on $m$ makes it difficult to increase $m$ very much. We seek a better solution, preferably one in which the cost for updates increases no worse than linearly with $m$.

The only existing distributed index that is notable in permitting high fanout is the P-Tree [8]. In this proposal, each P2P node maintains a leaf node in the B$^+$-tree and a path of virtual index nodes from the root to that leaf node. Search is very effective, but updates are expensive, possibly requiring substantial synchronization effort. In other words, this technique too has the same weak spot as multi-way Chord discussed above.

[*] Supported in part by the US National Science Foundation grant IIS-0219513

[†] Supported in part by IDA CCC grant as part of Best-Peer[17] project

*SIGMOD 2006*, June 27–29, 2006, Chicago, Illinois, USA.

In this paper, we propose a new multi-way distributed tree structure. We call it BATON*, since it is an extension of BATON[11], just as R*-trees are an extension of R-trees. BATON* extends BATON to permit a fanout of $m > 2$. With this, the cost of search becomes $O(log_m N)$, as expected. Moreover, the cost of updating routing tables is just $O(m \cdot log_m N)$, as compared to $O(log_2 N)$ in BATON – a degradation that is better than linear in $m$, and therefore a cost one may be willing to pay. Furthermore, BATON* has better fault tolerance properties than BATON, and is capable of quicker load balancing. In fact, the system's fault tolerance, measured as the number of nodes that must fail before the network is partitioned, increases linearly with $m$. Similarly, the expected cost of load balancing decreases linearly with $m$. We analyze the impact of $m$ on both search and maintenance cost, and derive an estimation model for $m$ that reduces the total cost given some composition of queries and updates. Likewise, we analyze the impact of $m$ on fault tolerance of BATON*.

BATON* can also support queries over multiple attributes in an effective way. In addition to permitting the use of multiple attributes in a single index, BATON* further introduces the notion of attribute classification, based on importance of the attribute for querying, and the notion of attribute groups. Another variation of BATON [12] has been proposed for supporting multi-dimensional queries. However, as analyzed in Section 5, it cannot support multi-attribute queries efficiently. Realizing that most of the time users only query over a small number of attributes while others are rarely queried, we propose a flexible method of indexing attributes in which only attributes, which are frequently queried, should be indexed separately; other attributes can be divided into small groups, and these groups are indexed as in multi-dimensional space.

To sum up, in this paper, we have the following major contributions:

- In Section 3, we introduce BATON*, a significant extension of BATON to multi-way tree structure. Therefore, we can reduce the cost for search from $O(log_2 N)$ to $O(log_m N)$.
- We analyze BATON*'s resilience towards node failure and network partitioning and its capability of load balancing.
- Using BATON*, we present a flexible method to support multi-attribute queries efficiently in Section 5.
- In Section 6, we show the experimental results of our proposed system over PlanetLab [7], a testbed for large-scale distributed systems.

In addition to the main sections mentioned above, this paper includes necessary background information about BATON in Section 2; in Section 4, there is a description of how Chord can be extended to have a fanout of $m$; we present related work in Section 7 and conclude in Section 8.

## 2. BACKGROUND

Since our proposed multi-way structure is adapted from BATON (BAlanced Tree Overlay Network) [11], we present here a brief review of the relevant features of BATON.

BATON is an overlay structure based on the binary balanced tree in which each peer in the network maintains a node of the tree. A node may connect to other nodes by up to four different kinds of links: parent links pointing to parent nodes, children links pointing to child nodes, adjacent links pointing to adjacent (in linear order) nodes that maintain adjacent ranges of values, and neighbor links pointing to selected neighbor nodes at the same level and have a distance equal to a power of two from the node. In BATON, each node in the tree, both leaf and internal, is assigned a range of values in which the range of values directly managed by a node is required to be greater than the range of values managed by its left adjacent node while smaller than the range of values managed by its right adjacent node. In other words, if we travel from the left to the right of the tree following adjacent links, data is in increasing order. An example of BATON is shown in Figure 1.

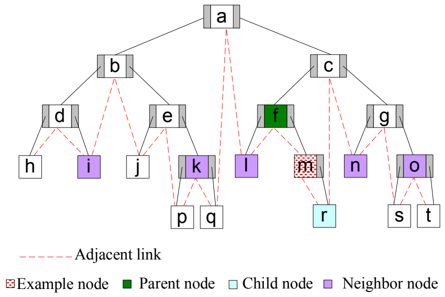


**Figure 1: BATON structure**

In BATON, a tree is considered *balanced* if and only if at any node in the tree, the height of its two subtrees differ by at most one. There are two important results in BATON. The first result shows that a tree is balanced if every node in the tree that has a child also has both its left and right routing tables full, i.e., none of the valid links in the routing table is Null. The second result shows that if a node $x$ contains a link to another node $y$ in its left or right routing tables, the parent node of $x$ must also contain a link to the parent node of $y$ unless the same node is parent of both $x$ and $y$. This result gives an efficient way to forward requests among nodes in the network.

Based on these two key results, BATON keeps the tree structure balanced by forcing every node in the tree that has a child to have both its left and right routing tables full. Therefore, when a node receives a *Join* request from a new node, it can only accept the new node as its child if its routing tables are full and it does not have two children. Otherwise, the *Join* request is forwarded to either (a) its parent if at least one of its routing tables is not full or (b) one of its adjacent nodes if it has both routing tables and children full. In case of node departure, if the departure of a node does not affect the tree balance, the node can safely leave the network. This is the case where the departing node is a leaf node and all of its neighbor nodes have no children. Otherwise, the departing node has to find a replacement node, which is a node satisfying the previous case, to take its place. The replacement node is found by sending a *FindReplacement* request downward the tree. Here, the cost of finding a position for a new node and the cost of finding a replacement node in case of node departure are all $O(logN)$ while the costs of updating routing tables to reflect changes are $6 \cdot logN$ and $8 \cdot logN$ for node join and node departure respectively.

In BATON, when a node receives a query request, if the searched value does not fall into its own range of values, the request is always forwarded to a node in its left routing table whose upper bound is still greater than the searched value or a node in its right routing table whose lower bound is still lower than the searched value if such a node exists. Otherwise, the query request is forwarded to either its left child/right child or its left adjacent/right adjacent node. Note that a query request can only be forwarded to a higher level node in two cases: (1) the higher level node contains the searched value, and hence it is necessary, (2) the processing node is a lower level leaf node without children and there are insufficient neighbor nodes inside its routing tables (in this case it is far from the root). As a result, BATON can avoid a bottleneck at the root and nodes near it.

Two load balancing schemes are used in BATON. In the first scheme, an overloaded/underloaded node performs load balancing with its adjacent nodes. In the second scheme, an overloaded/underloaded *leaf* node performs load balancing with a far away *leaf* node (a lightly loaded node may forcibly leave its current position and force-join as a child of the overloaded node to share the workload). The tree structure may become unbalanced in this case, and network restructuring becomes necessary. In BATON, network restructuring operations are akin to rotations in an AVL tree. Load balancing is performed by shifting nodes via adjacent links towards the place causing imbalance.

# 3. BATON*: A MULTI-WAY TREE STRUCTURE

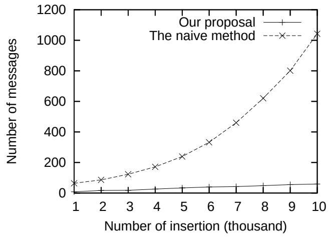


**Figure 2: Average additional messages required for doing load balancing**

BATON creates a binary search tree structure. As we attempt to increase the fanout, we find some challenges that must be overcome:

- In BATON, all nodes, whether leaf or internal, are responsible for a range of data values. There is an "in-order" traversal linearization of these value ranges. If a node has more than two children, this sort of in-order linearization is no longer possible. This implies that we no longer have a clean definition of adjacency of value range for internal nodes, and it is not clear what adjacent links should be kept. Even if we choose to be generous and maintain multiple adjacencies, one has to be chosen when needed for search traversal and for load balancing. This choice is not clear when there are multiple adjacent nodes.

  For example, if an internal node is overloaded by just one range of values, it also can only do load balancing with two of its adjacent nodes (not with all $2 \cdot (m-1)$ adjacent nodes). All the above things bring back the problem of propagating load balancing operations when a node is overloaded, which is avoided in BATON. Figure 2 shows a comparison between our proposal and this method in doing load balancing. The result shows that this naive method is very bad in solving load balancing problem.

- It appears straightforward to expand the neighbor routing tables to capture neighbors at distances that are powers of $m$ distant rather than powers of 2 distant. This turns out to be the wrong answer since it leaves the search network under-connected.

## 3.1 The BATON* Structure

In this section, we propose a new structure called BATON*, which addresses the challenges listed above in extending BATON to a multi-way structure. The new structure is shown in Figure 3. There are three significant differences between the multi-way BATON* and the binary BATON.

- Each peer node in BATON* can have up to $m$ children instead of two as in the original structure. In addition to maintaining links to children, the parent node also has to keep track of the ranges of values managed by their children.

- Neighbor routing tables at a node maintain links to selected neighbor nodes at the same level which have a distance equal to $d \cdot m^i$, where $d = 1..m-1$ and $i \geq 0$, from the node itself. As shown in Figure 3, left routing table of node $o$ maintains links to nodes $n$, $m$, $l$, $k$, $g$, which have a distance from node $o$ respectively $1 \cdot 4^0, 2 \cdot 4^0, 3 \cdot 4^0, 1 \cdot 4^1$, and $2 \cdot 4^1$. Similarly, the right routing table of node $o$ maintains links to nodes $p$, $q$, $r$, $s$. As a result, the maximum number of links in routing tables of a node at level $L$ is bounded at $(m-1) \cdot log_m$*(number of nodes at that level)*$= (m-1) \cdot L$. Note that if we increase the fanout of a node to reduce the cost of search, we have to increase the size of routing tables, and hence increase the cost of update tables. Consequently, depending on application requirements, an appropriate value of fanout factor $m$ should be chosen. We will show later how to select such an $m$.

- For a system of order $m$, a range of values managed by a node is greater than ranges of values managed by the first $\lceil m/2 \rceil$ children nodes while less than ranges of values managed by the last $\lfloor m/2 \rfloor$ children nodes. For example, in Figure 3, range of values managed by $o$ is greater than those of $y$, $x$, $n$, but smaller than those of $z$, $d$, $p$, $q$.

Note that in BATON*, a node manages only *a range of values* and $m$ links instead of *(m-1) ranges of values* and $m$ links as in the original multi-way tree structure. It also does not look like a B-tree, where every internal node must have a number of children between $m/2$ and $m$. Rather, in BATON*, all internal nodes, except immediate parents of leaf nodes, are required to have full $m$ children. Immediate

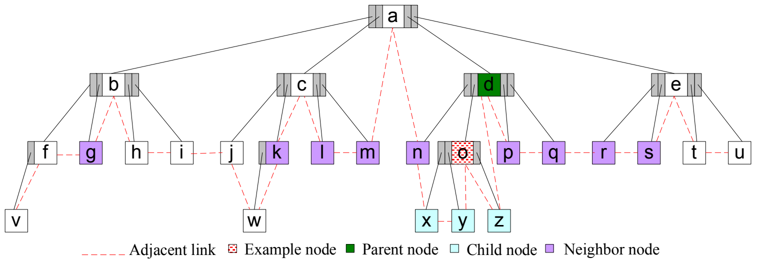


**Figure 3: BATON* structure**

parents of leaf nodes may have fewer children, and this is where growth takes place.

## 3.2 Definition and Theorems

For an $m$-fanout tree, with $m > 2$, the definition of a binary balanced tree is extended naturally to the form:

***Definition 1:*** *A tree is* balanced *if and only if at any node in the tree, the height of* any two subtrees *of its children differ by at most one.* □

The two crucial results underpinning BATON continue to apply to BATON*, with suitable extension in wording from 2 to $m$, in spite of all the changes described above. We establish these extended results next.

***Theorem 1:*** *A tree is a balanced tree if every node in the tree that has a child also has both its left and right routing tables full.*

**Proof:** Assume that there exists a subtree rooted at node $x$, which is imbalanced. As a result, there must exist two subtrees rooted at $y$ and $z$, children of $x$, whose heights differ by more than 1. Let $N_y$ and $N_z$ be numbers of $y$ and $z$, $H_y$ and $H_z$ be heights of their corresponding subtrees. Without loss of generality suppose that $z$ is a right sibling of $y$, and $H_y - 1 > H_z$. Also let $w$ be the farthest leaf node of the subtree rooted at $y$, $v$ be its parent, and $N_v$ be number of $v$. Since $z$ is a right sibling of $y$, we have $N_z = N_y + t$, where $(1 \leq t < m)$ (1). By the way nodes are numbered, we have $(N_y - 1) \cdot m^{H_y - 1} + 1 \leq N_v \leq N_y \cdot m^{H_y - 1}$ (2). Since $v$ has a child $w$, by the system requirement, it has to have a full right routing table in which there must be a node $u$ with number $N_u = N_v + d \cdot m^i$, where $1 \leq d \leq m$ and $i \geq 0$. By choosing $d = t$, $i = H_y - 1$, we have $N_u = N_v + t \cdot m^{H_y - 1} \Rightarrow N_v = N_u - t \cdot m^{H_y - 1}$ (3). From (2) and (3), we have $(N_y - 1) \cdot m^{H_y - 1} + 1 \leq N_u - t \cdot m^{H_y - 1} \leq N_y \cdot m^{H_y - 1}$ $\Rightarrow (N_y + t - 1) \cdot m^{H_y - 1} + 1 \leq N_u \leq (N_y + t) \cdot m^{H_y - 1}$ (4). From (1) and (4), we have $(N_z - 1) \cdot m^{H_y - 1} + 1 \leq N_u \leq N_z \cdot m^{H_y - 1}$. As a result, $N_u$ has to belong to a descendant node at height $H_y - 1$ of the subtree rooted at $z$. It means that the height of the subtree rooted at $z$ is at least $H_y - 1$. Since this contradicts the assumption, there does not exist a subtree that is unbalanced. The proof is illustrated in Figure 4. □

***Theorem 2:*** *If a node $x$ contains a link to another node $y$ in its left or right routing tables, the parent node of $x$ must also contain a link to the parent node of $y$ unless the same node is parent of both $x$ and $y$.*

**Proof:** Let $y$ be a neighbor node of $x$, $w$ be the parent of $x$, $v$ be the parent of $y$, and $N_x$, $N_y$, $N_w$, $N_v$ be the number of $x$, $y$, $w$, $v$ respectively. By the way nodes are numbered, we have $N_w = N_x \ div \ m$ and $N_v = N_y \ div \ m$. By the way neighbor links are established, we have $N_y = N_x \pm d \cdot m^i$. Let us consider two cases. Case 1: if i = 0, then $N_y = N_x \pm d$. As a result, $N_v = (N_x \pm d) \ div \ m$. Since $1 \leq d \leq m - 1$, $N_w - 1 \leq N_v \leq N_w + 1$ or $N_w - 1 \cdot m^0 \leq N_v \leq N_w + 1 \cdot m^0$. Consequently, $v$ has to be a neighbor of $w$ or be $w$ itself. Case 2: if i $\geq$ 1, $N_v = (N_x \pm d \cdot m^i) \ div \ m = N_w \pm d \cdot m^j$, where $j = i - 1$. As a result, $v$ has to be a neighbor of $w$.

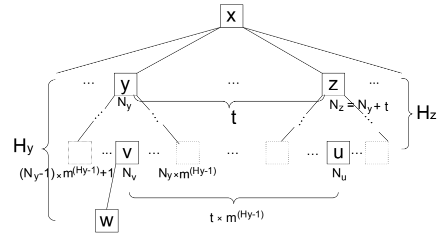

x
y
z
$N_y$
t
$N_z = N_y + t$
$H_z$
$H_y$
v
u
$(N_y\text{-}1) \times m^{(H_y\text{-}1)}+1$
$N_v$
$N_y \times m^{(H_y\text{-}1)}$
$N_u$
w
$t \times m^{(H_y\text{-}1)}$

**Figure 4: Proof of theorem 1**

In addition to the above theorems, we establish two new theorems as below.

***Theorem 3:*** *The total number of nodes at level l is (m - 1) times greater than the total number of nodes at all previous levels, 0 through $l - 1$.*

**Proof:** Let $x$ be the total number of nodes at level $l$, and $y$ be the total number of nodes at all previous levels, then $x = m^l$, and $y = 1 + m^1 + m^2 + ... + m^{l-1} = (m^l - 1)/(m - 1)$. As a result, $y \cdot (m - 1) = m^l - 1 < x$. □

This theorem implies that the vast majority of nodes in the system have a high level number, and are possibly leaf nodes. This dominance of leaf nodes will turn out to be crucial for lowering the cost of load balancing as we shall see shortly.

***Theorem 4:****The maximum size of a routing table of a node at level l is $m \cdot l$.*

**Proof:** This theorem follows from the way we construct routing tables. Since there are total $m^l$ nodes at level $l$, and a node maintains neighbor nodes at distance $d \cdot m^i$, where $d = 1..m - 1$, the maximum size of a routing table is $d \cdot log_m m^l = d \cdot l$. □

***Corollary 4.1:****The maximum size of a routing table of a node in the network is $m \cdot log_m N$.*

**Proof:** It is trivial since the maximum height of the tree is $log_m N$. □

This corollary provides us with the tools necessary for efficient update. If we can restrict the cost of an update to be that of the construction of a single neighbor routing table, then we have the necessary sub-quadratic dependence.

Algorithms for various operations are adapted from BA-

TON to BATON* as described next.

## 3.3 Node Join and Node Departure

**Algorithm 1** : Join (node n, node newNode)

```
If (Full(LeftRoutingTable(n)) and
   Full(RightRoutingTable(n)) and Not Full(Children(n)))
  n.AcceptChild(newNode)
  If (Adjacent(newNode) = n)
    SplitData(n, newNode)
  Else
    SplitData(AdjacentSibling(newNode), newNode)
Else
  If ((Not Full(LeftRoutingTable(n))) or
     (Not Full(RightRoutingTable(n))))
    Join(Parent(n), newNode)
  Else
    m=SomeNodesNotHavingEnoughChildrenIn
        (LeftRoutingTable(n), RightRoutingTable(n))
    If (there exists such an m)
      Join(m, newNode)
    Else
      a = one of its adjacent nodes
      Join(a, newNode)
```

A node can only accept a new joining node as a child if it has full neighbor routing tables but does not have $m$ children. Otherwise, it has to forward the join request to either its parent, its lower level adjacent node or a neighbor node that does not have enough children. A node can only leave its current position if it does not cause the tree to become unbalanced. Otherwise, it has to find a replacement node by sending a request to its lower level adjacent node.

In BATON, the range of values managed by a newly inserted node is always obtained as a split from the range managed by its parent. In BATON*, limiting the range assigned to the new node thus would force newly inserted nodes to be adjacent to their parent. Instead, we allow newly inserted nodes to appear anywhere in the adjacency order (we can actually choose this to maximize local load equalization). The range of values managed by the new node is obtained from one of its adjacent nodes, either the parent or a sibling.

In BATON*, the cost of finding a place for a new joining node or finding a replacement node is $O(log_m N)$ since the height of the tree is $O(log_m N)$. The cost of updating routing table is $O(m \cdot log_m N)$ for the neighbor routing tables since the maximum number of neighbor nodes a node can have is $O(m \cdot log_m N)$, and each of these has to add a new entry, or remove an entry from its routing tables. Also, a newly inserted node has to construct its own routing tables, with up to $O(m \cdot log_m N)$ entries, each of which can be obtained in constant time through its parent. In addition, there is a parent link and two adjacency links to create/delete. There can be no children links for a node being newly inserted or deleted. Adding these up, the total cost of insertion or deletion is $O(m \cdot log_m N)$.

## 3.4 Query Processing, Data Insertion and Data Deletion

The algorithms for query processing as well as data insertion and data deletion are also a little different in BATON* than in BATON. The basic operation in all of these is the basic equality search. A node $u$ receiving a search request checks to see if there is a neighbor node it knows about who is more appropriate to handle the search. If the searched value is greater than the $u$'s own upper bound while there is no right hand side neighbor node of $u$ whose lower bound is less than the searched value, the search has to be forwarded to a suitable child. In BATON, this would simply be the right child. In BATON*, the available information regarding the bounds of the ranges maintained by the various children are considered to find the rightmost child whose lower bound is less than the searched value, and the search request is forwarded to it. Similarly, if the searched value is less than the node's lower bound while there is no left hand side neighbor node whose upper bound is greater than the searched value, the node has to try to find the leftmost child whose upper bound is greater than the searched value, to forward the search request. The algorithm is described as in Algorithm 2.

**Algorithm 2** : Search-Exact (node n, query q)

```
If ((n.LowerBound<=q.Value) and
   (q.Value<=n.UpperBound))
  LocalSearch(n, q)
Else
  If (n.UpperBound<q.Value)
    m=TheFarthestNodeSatisfyingCondition
       (m.LowerBound<=q.Value)
    If (there exists such an m)
      Search-Exact(m, q)
    Else
      l=TheFarthestChildSatisfyingCondition
         (l.LowerBound<=q.Value)
      If (there exists such an l)
        Search-Exact(l, q)
      Else
        Search-Exact(RightAdjacentNode(n), q)
  Else //(n.LowerBound>q.Value)
    //A similar process is followed towards the left
```

Algorithms for range query, data insertion and data deletion are modified from those in BATON in corresponding ways.

## 3.5 Node Failure, Fault Tolerance, Network Restructuring and Load Balancing

Algorithms for node failure, network restructuring and load balancing are all the same in BATON* as in BATON. Node failure is recovered via the node's parent. Load balancing is done between two adjacent nodes or between a lightly loaded leaf node and a heavily loaded leaf node. As needed to achieve load balance, network restructuring is done by shifting nodes via adjacency links. Details of these algorithms are omitted here, since they can be looked up in [11] and used with the obvious changes from 2 to $m$. Instead, we focus here on fault tolerance, which has been greatly improved due to higher fanout, and the load balancing that has been made easier by the new design.

### *3.5.1 Fault Tolerance*

First, let us consider a case where a node is isolated from the network. If a node has full routing tables, it is easy to realize that the lower the level of a non-leaf node, the more the number of links the node has because low level nodes always have more neighbor links than high level nodes do while the number of parent links, adjacent links, and children links are approximately equal among nodes . As a result, high level nodes (closer to the root) are easier separated from the network than low level nodes are. As in BATON, there are two special cases. In the first case, the root is a node, which is easily isolated because it has no neighbor link. In this case, if all of its $m$ children and 2 adjacency links of the root are broken, it is isolated from the remaining system. In the second case, if a node's routing tables are not full, it is also easily isolated from the network. However, that node is likely to be a newly joined node which can rejoin the network at low cost. Note that in BATON*, the longer the time a node in the system is, the higher the level it is in the tree because new nodes always join as children of existing nodes.

Now, let us consider a case where a group of connected nodes is separated from the network. These nodes are isolated if all of their links to nodes outside the isolated group are broken. In other words, the minimum number of links that has to be broken for a group of nodes to be separated from the network is $S = \sum$ *links of all nodes in group* - $\sum$ *intra-group links among nodes within group*. Since high level nodes always contain fewer links than low level nodes do, we will analyze only the case where the network is separated by low level nodes and high level nodes (in other cases, $S$ always has higher values). In other words, the network is separated between the head of the tree and the base of the tree as in Figure 5. Since the number of low level nodes (with a large level number) dominates the system, the number of high level nodes, which can be separated from the network, is not considerable. Note that removal of nodes high in the tree causes no more disruption than removal of nodes near the leaf – they are no more important for purposes of search and update. In particular, the root of the tree can be removed and not cause the tree to fall apart into a disjoint forest.

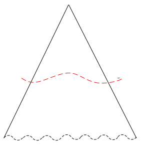


**Figure 5: Network partition**

For further analysis, let us try to make $S$ as small as possible. $S$ has the minimum value when all nodes in the first part are arranged so that all levels belonged to the group since all neighbor links, which contribute the largest number among all kinds of links, are inter-connected links. In this case, the remaining links, which have to be broken, are adjacency links of nodes to leaf nodes and children links of nodes in the last level to their children. Let $k$ be the number of nodes in the first part, and $k'$ be the total number of nodes in the last level of the first part, then $S = 2{\cdot}k{+}m{\cdot}k'$. Following theorem 3, we have $k' > (m-1)\cdot(k-k')$ or $k' > k\cdot(m-1)/m$. As a result $S < 2\cdot k + m\cdot k\cdot(m-1)/m = 2\cdot k + (m-1)\cdot k$ (1). In the worst case, if the last level of the first part is just one level above the last level of the tree, adjacency links may point to the same node, and hence $2\cdot k$ nodes may be double counted in the previous formula. As a result, $S$ is only less than $(m-1)\cdot k$ (2).

(1) and (2) $\Rightarrow S < [(m-1)\cdot k, (m+1)\cdot k]$. In other words, let $f$ be the number of failed nodes in the system, then the maximum number of nodes, which may be separated from the network as a result of the failure is from $f/(m+1)$ to $f/(m-1)$. We can easily realize that when $m$ is large enough, this rate is very small. In conclusion, our system is highly fault tolerant.

### 3.5.2 *Load Balancing*

Local load balancing, with adjacent nodes, is easy to do, and is the cheapest form of load balancing where it works. However, such local balancing will not suffice where there are global imbalances. For example, when data distribution is skewed, using only local load balancing may lead to ripple data migration, which is costly. A solution for this problem is that we have to remove nodes from underloaded regions and add them to overloaded regions. Since our system employs the tree structure, internal nodes are not easily to be removed, this solution is only possible for leaf nodes. In special, a workload can be adjusted between a heavily loaded leaf node and a lightly loaded leaf node. As fanout increases leads to an increasing number of leaf nodes, when the fanout $m$ is large, finding a leaf node satisfying condition to do load balancing is easy.

In general, if a node is overloaded, it tries to do load balancing with its adjacent nodes first. If there is no lightly loaded adjacent nodes, it then tries to find a lightly loaded leaf node to do load balancing. Once such a node is found, that node has to perform a forced leave at its current position and a forced join at the new position to share the workload of the overloaded leaf node by leaving the current position and joining in the new position. The tree structure may become imbalance after that. Thus, network restructuring is triggered if necessary. The cost of network restructuring is comparable irrespective of fanout, and depends solely on the number of nodes intervening between the old and new positions of a moved node.

## 3.6 Tuning the Fanout $m$

The cost for search is decreased logarithmically as the fanout is increased. The cost for a node insertion and deletion goes up. Since, search is much more frequent than node insertion and deletion in most systems, this is a good tradeoff to have. Exactly how large a fanout to choose is determined by the precise ratio of queries to node updates. In this section, we provide a simple cost model for calculating the optimum value of $m$.

Since search operations and update operations are dominant operations in our system[1], we just build a simple cost model ($C$) based on them. Let $\alpha$ be the percentage of search operations, and $1-\alpha$ be the percentage of update operations in the system. Then $C = \alpha\cdot S + (1-\alpha)\cdot U$, $0 \leq \alpha \leq 1$, where $S$ and $U$ denote the search and update cost respectively. Since the approximate costs of a search op-

[1] Insertions and deletions can be considered as search operations while node join operations and node departure operations can be considered as update operations since they will trigger update operations.

eration and an update operation are respectively $O(log_m N)$ and $O(m \cdot log_m N)$ (in term of routing messages), we have $C(m) = \alpha \cdot log_m N + (1-\alpha) \cdot m \cdot log_m N$. Here, $\alpha$ can be considered as the tuning knob. If all operations in the network involve search operations, $\alpha$ should be big, and vice versa.

Differentiating $C(m)$, we have $C'(m) = \frac{ln N}{m \cdot ln^2 m} \cdot [(1-\alpha) \cdot m \cdot (ln\ m - 1) - \alpha]$. Let $F(m) = (1-\alpha) \cdot m \cdot (ln\ m - 1) - \alpha$. Since $F'(m) = (1-\alpha) \cdot ln\ m \geq 0$, $F(m)$ is an increasing function. Thus, $F(m) = 0$ has a unique solution $m_0$. It leads to the result $C'(m) = 0$ when $m = m_0$, $C'(m) < 0$ when $m < m_0$, and $C'(m) > 0$ when $m > m_0$. As a result, $C(m)$ is minimum at $m_0$.

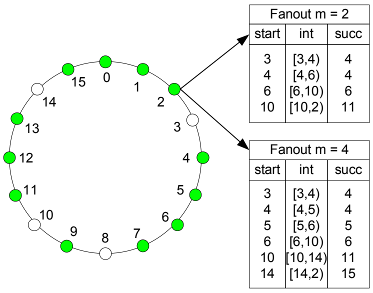
Even though we cannot easily find a closed form solution for $F(m) = 0$, it is not hard to find a numerical solution by numerical methods such as Newton's method [2]. Upon finding $m_0$, we calculate $C(m_1)$, $m_1 = \lfloor m_0 \rfloor$ and $C(m_2)$, $m_2 = \lceil m_0 \rceil$, and select the value corresponding to the smallest cost. The result is a good value for fanout. For example, in a system with $10,000$ nodes, $\alpha = 0.6$, using Newton's method with initial value 2, iteration = 1,000, we have $m_0 = 3.9673$. By checking $C(m_1 = 3)$ and $C(m_2 = 4)$, we get a fanout value of 4 because $C(3) = 15.0905 > C(4) = 14.6165$.

# 4. A NOTE ON MULTI-WAY STRUCTURES

It is straightforward to extend many well-known distributed search structures to have a fanout of $m$ rather than 2 at each step, using techniques similar to those used in BATON*. However, the cost of update will typically be quadratic in fanout, unlike in BATON*. Here, we present a sketch of the algorithm and the analysis for Chord. Arguments for other distributed search structures are similar.

Let us call the extended structure of Chord for a higher fanout Chord*. Similar to BATON*, we have to modify the way to organize routing tables (or finger tables). Instead of keeping links to nodes at distance $2^i$, a routing table now keeps links to nodes at distance $d \cdot m^i$, where $d = 1..m-1$. In particular, the way to calculate finger[k].start is modified as below: finger[$k$].start = $(n + d \cdot m^{k-1})$ $mod$ $N$. Figure 6 shows two different routing tables corresponding to different fanouts: $m = 2$, and $m = 4$. Except for changes in routing tables, all Chord's algorithms are unchanged in Chord*.

The problem with extending Chord is the cost of updating routing tables. In the modified Chord*, the size of routing table is $(m-1) \cdot log_m N$. As a result, the cost of updating routing table for a new joining node or a departure node grows up to $O(((m-1) \cdot log_m N)^2)$ now. Compared to the previous cost of Chord $O((log_2 N)^2)$, the increasing factor is $((m-1)/log_2 m)^2 \simeq (m/log_2 m)^2)$. This is expensive compared to BATON* since BATON* only requires an increasing factor of $m \cdot log_m N / log_2 N = m/log_2 m$ (a linear cost with m). Further, an overlay structure such as Chord that relies on hashing cannot support range queries efficiently [11].



| Fanout m = 2 | | |
|---|---|---|
| start | int | succ |
| 3 | [3,4) | 4 |
| 4 | [4,6) | 4 |
| 6 | [6,10) | 6 |
| 10 | [10,2) | 11 |

| Fanout m = 4 | | |
|---|---|---|
| start | int | succ |
| 3 | [3,4) | 4 |
| 4 | [4,5) | 4 |
| 5 | [5,6) | 5 |
| 6 | [6,10) | 6 |
| 10 | [10,14) | 11 |
| 14 | [14,2) | 15 |

**Figure 6: A CHORD ring**

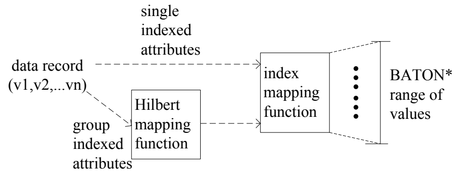


**Figure 7: Data indexing**

# 5. A FLEXIBLE METHOD FOR SUPPORTING MULTI-ATTRIBUTE QUERIES

In applications that involve multiple attributes, it is not uncommon for queries to involve only a small number of attributes (instead of all the attributes). As a result, if we consider multi-attribute queries as multi-dimensional queries and use systems such as CAN [18] or [12] to support them, it is not efficient because most of the time the queries will involve a large search region. An alternative method proposed by MAAN [6] and Mercury [5] is to index each attribute value separately. Therefore, multi-attribute queries are solved by choosing one attribute for indexed search while other attributes in the queries are used as post-filters. Even though this method overcomes the problem of the previous one, it is still not efficient when the number of attributes increases.

To support multi-attribute queries over BATON*, we propose a method in which we divide the whole range of BATON* attributes into several sections: each section is used to index an attribute (if it frequently appears in queries) or a group of attributes (if these attributes rarely appear in queries). Since BATON* can only support queries over one dimensional data, if we index a group of attributes, we have to convert their values into one dimensional values: in our system we choose Hilbert Space Filling Curve. For example, if we have a system with 10 attributes: $a_1$, $a_2$,... $a_{10}$ in which only 4 attributes from $a_1$ to $a_4$ are frequently queried (i.e. 90% of all queries), we will build 4 separate indexes for them. The remaining attributes are divided equally into two groups to index, three attributes in each group. In this way, we can significantly reduce the number of replications from 10 down to 6. The index structure is described as Figure 7.

## 5.1 Multi-attribute Query Processing

Generally, a multi-attribute query Q can be defined as a set of subqueries $q_i$, in which each subquery involves an attribute. The result of the query Q is an intersection of all results from all subqueries. As a result, we can find the final result of the query Q by just finding results satisfying an arbitrary subquery, and using other subqueries as filters. We call the selected subquery a dominant subquery. After taking an arbitrary subquery as the dominant subquery, we have to convert it into queries, which can be executed over BATON* overlay network since the domain of values of a subquery is just a section in the overlay network. To do converting we need to consider two cases. If the dominant subquery involves an attribute that is indexed separately, it is transformed into a BATON* query in a straightforward way – using the mapping function. However, if the dominant subquery involves an attribute that is indexed together with other attributes (multi-dimensional index), it has to be mapped first into smaller subqueries by using the Hilbert Space Filling Curve. After that, these subqueries are converted into BATON* queries. Finally, these converted subqueries are forwarded in parallel over BATON*.

## 5.2 Data Insertion and Deletion

When data is inserted, it has to be indexed in each index structure. Since inserted data can be considered as a multi-attribute query in which subqueries are exact match queries corresponding to the attribute values, the data insertion algorithm is similar to the query processing algorithm described above except that all sub queries have to be considered to index data in all index structures.

To delete existing data, all of its replications have to be deleted. Like the data insertion algorithm, nodes that keep index values are found by a corresponding multi-attribute query. After that, the data index is removed from those nodes.

## 5.3 Heuristics to Enhance Performance

In this section, we introduce several techniques that can help to achieve a better performance by reducing the number of search steps as well as the number of messages in query processing.

First, for a multi-attribute query involving many attributes, if we can choose a query that has only a few of the satisfied results or results are stored in a small number of nodes, the query processing just needs a few steps to complete as well as to filter unwanted results. As a result, we propose some heuristics that can be applied to select the dominant subquery as follows. (1) If there is a subquery involving an attribute that is indexed separately, it is a candidate for selection because searching over a single index is always faster than searching over a group index. (2) If there is a subquery that is an exact match query, it is a candidate for selection because an exact match query always returns fewer results than a range query. (3) If there are multiple subqueries involving attributes that are indexed together, instead of selecting only one dominant subquery, we should take all these subqueries as dominant subqueries to search since we can filter the region of the search over multi-dimensional space.

Second, in some applications, there exist attributes that are always queried together. In such a case, we should group them together into a group to index no matter whether they are frequently queried or rarely queried. As discussed above, subqueries involving attributes that are indexed together can be used as dominant subqueries for searching. Thus, we can always reduce the cost of search in this kind of query.

Third, from the experiments, we realize that the system can achieve better performance if we make the structure more flexible. It means that we do not need to define only two kinds of attributes and divide rarely attributes equally into groups as discussed above. Depending on the frequency of attributes in queries, we can group them into groups with different number of attributes.

Fourth, for some attributes, their values can only fall in a small range of values while for others, their values can fall in a large range of values. As a result, if we divide the whole range of values of BATON* into equal sections, those attributes with small range of values will have a sparse index whereas those attributes with large range of values will have a dense index. This leads to the problem of skewed data distribution. Even though, BATON* can handle skewed data distribution well, it takes some cost to do load balancing. To avoid this problem, we propose that values of sections are divided based on needs. It means that attributes with bigger range of values have to be granted bigger index space.

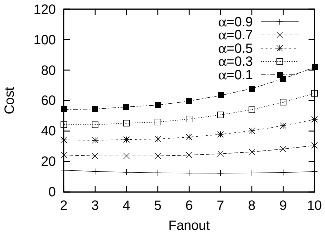


**Figure 9: Optimal values of fanout**

Finally, as discussed in the previous section, subqueries, which are created from converting a query in multi-dimensional index, are forwarded in parallel. However, we realize that most of the time these subqueries are forwarded to only one node or a few nodes since they are all related to nearby range of values. As a result, instead of sending multiple query messages to the same node, we can combine these messages into one message. This technique can significantly reduce the number of query messages which has to be forwarded.

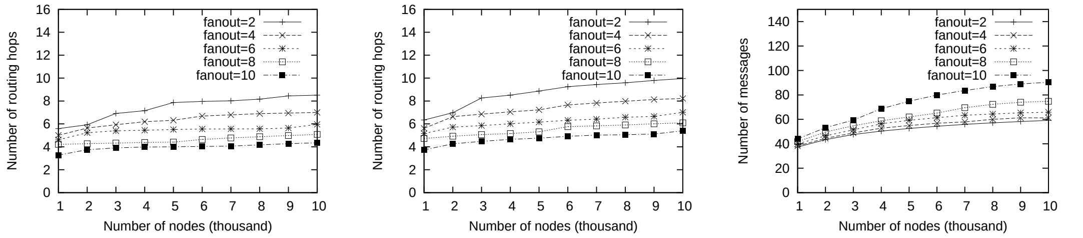
# 6. EVALUATION

To evaluate the performance of our proposal, we implemented a simulation system in Java and ran it over Planetlab [7], a testbed for large-scale distributed systems. In our implementation, each peer node is identified both physically by a pair of IP address and port number and logically by its position in the tree structure. Each Planetlab node is used to simulate hundreds of peer nodes. There is a fake server, which creates events and sends them to peer nodes for processing (it exists for experimental purpose only). Communication between nodes is via sockets.

To evaluate the performance of our system, we tested the network with different numbers of nodes from 1,000 to 10,000 using different fanout values from 2 to 10. A number of data equal to the network size multiplies 1000, which are numbers from 1 to 1,000,000,000, are inserted to the network in batches. For each test, 1,000 exact match queries and 1,000 range queries are executed, and the average cost of operations are taken. Searched ranges are created randomly by getting the whole range of values divided by the total number of nodes multiplies $\alpha$, where $\alpha = 1..10$. Note that in some experiments, where it is not necessary to vary the network size or the fanout, the default value of network size is 10,000 nodes while the default value of fanout is 4.

## 6.1 Single-attribute Query

We can see that with the new BATON* structure the cost of exact match query and range query is reduced when the fanout is increased as in Figure 8(a) and 8(b). In particular, increasing fanout brings greater benefit when the number of



(a) Cost of exact match query (b) Cost of range query (c) Cost of updating routing table

**Figure 8: Effect of varying fanout values**

**Table 1: A comparison between an estimated and the best experimental fanout**

| $\alpha$ | $m_1$ | $m_2$ | $C(m_1)$ | $C(m_2)$ | Error |
|---|---|---|---|---|---|
| 0.9 | 8 | 7 | 12.5145 | 12.3545 | 0.0128 |
| 0.7 | 5 | 5 | 23.6634 | 23.6634 | 0.0000 |
| 0.5 | 4 | 3 | 34.41 | 33.9298 | 0.0136 |
| 0.3 | 3 | 3 | 44.1678 | 44.1678 | 0.0000 |
| 0.1 | 3 | 2 | 54.4058 | 54.1764 | 0.0042 |

nodes in the system is big because the larger the number of nodes, the more the height of the tree structure is reduced when fanout is increased. However, a faster search process always requires a bigger storage to keep index information, and there is no exception in this case. Besides reducing cost of search, increasing fanout also leads to increasing routing table size, and hence increasing cost of updating routing table as shown in Figure 8(c). As a result, depending on application, we should select a suitable fanout to tradeoff between these costs as described in the cost model in Section 3.6. Figure 9 shows the experimental results of using different fanout values for different $\alpha$ or knob values. Together with Table 1, the results show that the cost of our estimated good fanout values are not much different from the cost of the real best fanout values returned from the experiment. In the table, $m_1$ is the estimated good fanout, $m_2$ is the best experimental fanout, $C(m_1)$ is the cost of doing experiment with fanout $m_1$, $C(m_2)$ is the cost of doing experiment with fanout $m_2$, and Error is calculated by the formula: $E = |(C(m_1) - C(m_2))/C(m_1)|$.

## 6.2 Multi-attribute Query

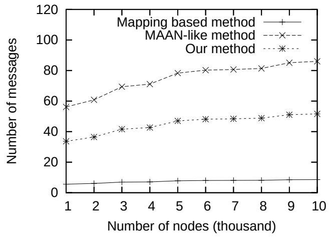
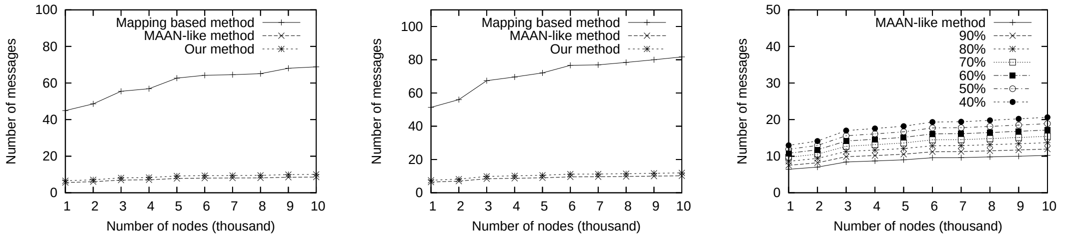


**Figure 10: Cost of insertion**

To evaluate the performance of multi-attribute query, we used the same schema as in our previous example. There are totally 10 attributes in which 4 attributes are frequently queried while 6 others are rarely queried. For simplicity, we just assume all attributes values are integers. We compare our BATON* against two other methods. The first is a straightforward mapping based method, which considers 10 attributes as 10 dimensions and maps the data points into a single-dimensional space using Hilbert curve so that the transformed values can be indexed using BATON*. Accordingly, during query processing, the multi-attribute query is transformed into a set of single-dimensional queries. Such transformation approach has been adopted in various existing works such as [21]. The second method is similar to those used in MAAN [6] and Mercury [5] where the data is indexed separately for each attribute and an attribute is randomly chosen among all the attributes appearing in the query for searching. In more detail, 10 overlays of BATON* were used, one for each attribute.

The queries used in this experiment is composed of 90 percent of queries involving 4 common attributes and 10 percent of queries involving rarely queried attributes. The results are shown in Figure 10 and Figure 11(a), 11(b). We observe that the insertion cost of the mapping based method is the lowest since the data point is indexed only once. However, the search cost of the mapping based method is very high. It is much higher than the other schemes, and is clearly unacceptable. On the other hand, indexing 10 attributes as in the MAAN-like method incurs high insertion cost as a data item is indexed 10 times. Our proposed method strikes a balance between the query cost and insertion cost, and the attribute grouping could be tuned based on the query patterns and loads. In fact, the experiment results show that our method requires much less cost of insertion than the MAAN-like method while doesn't require much higher search cost.

In the experiment above, 90 percent of queries involve common attributes. In this experiment, we study the effect of varying the percentage of queries involving common attributes. Figure 11(c) shows the results as we vary the Percentages from 40 percent to 90 percent. The result shows that the search performance decreases when the attributes, indexed separately are queried less. In the worst case, if all attributes are equally queried, the search cost is twice the cost of the method which indexes all attributes. However, it is still acceptable, if the main target of the system is to reduce the cost of data indexing. It is the case where the system consists of a huge number of attributes, and hence indexing each attribute separately is impossible due to the huge cost of data indexing.



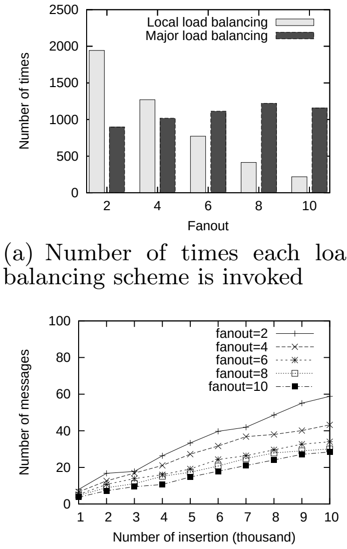
(a) Cost of exact match query

(b) Cost of range query

(c) Effect of varying query distribution on range query

**Figure 11: Cost of searching multi-attribute queries**

## 6.3 Load Balancing



(a) Number of times each load balancing scheme is invoked

(b) Average additional messages required for doing load balancing

**Figure 12: Cost of load balancing**

Increasing fanout not only helps to reduce the search cost but also helps to achieve better load balancing. To verify this claim, we test the network with a skewed data distribution, which is generated using a Zipfian distribution with parameter 1.0 and evaluate the cost of load balancing as we vary fanout. For simplicity, in our system, we assume that the query distribution follows the data distribution. As a result, the workload of a node is solely determined by the amount of data stored at that node. When a node joins the network it is assigned a default upper and lower load limit by its parent. If the number of stored data at the node is greater than the upper bound boundary, it is considered as an overloaded node and vice versa. If a node is overloaded and it cannot find a lightly loaded leaf node, it is likely that all other nodes also have the same work load, so it automatically increases the boundaries of storage capability.

As we discussed above, increasing fanout leads to increasing number of leaf nodes. As a result, nodes can often load balance by sharing the work load between a heavily loaded leaf node and a lightly loaded leaf node. When this is not possible, a more expensive "major" load balancing operation is required, involving the forced removal and forced reinsertion of a lightly loaded node elsewhere in the system. Figure 12(a) shows the number of times each load balancing scheme is involved in a 10,000 nodes network with different fanouts. The results show that when the fanout is increased, the number of times when the local load balancing is invoked is significantly decreased while the number of times when the major load balancing is invoked is not increased much. This may appear counter-intuitive at first, however it is because many of the times when the local load balancing scheme is invoked, it is due to ripple data migration between adjacent nodes when a heavily loaded leaf node can't find a lightly loaded leaf node. With higher fanouts, a heavily loaded leaf node can find a lightly loaded leaf node easier and hence can avoid the problem of ripple data migration. This leads to the result that the higher the fanout is, the fewer the number of load balancing efforts is needed. The result is confirmed in Figure 12(b), which shows the costs of load balancing in different network sizes and different fanouts.

## 6.4 Fault Tolerance

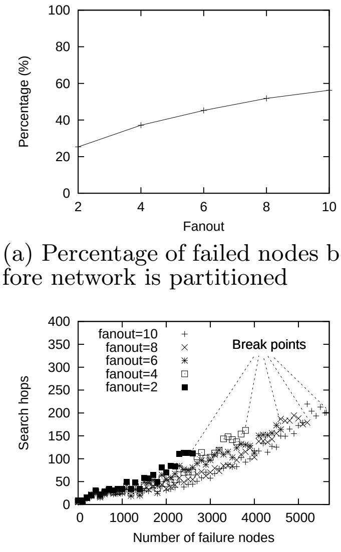
To evaluate the system's fault tolerance in case of massive failure we initialized the system with 10,000 nodes. After that, we let nodes randomly fail step by step without recovering. At each step, we check to see if the network is partitioned or not. Figure 13(a) shows the average percentage of nodes that must fail before the network is partitioned. The result confirms that our system is highly fault tolerant since it is only partitioned when approximately one fourth of nodes fails in case of fanout = 2. Increasing fanout leads to increasing fault tolerance: more than half the nodes fail before network with fanout 10 is partitioned. Moreover, most of time, only a small number of nodes is separated from the majority as shown in Table 2.

The problem brought by massive failure is massive destruction of links connected to failed nodes. Since the search process has to bypass the failure nodes, and there is no way to know exactly which path can safely lead to the destination node without interruption of failure nodes, the search query has to be forwarded forth and back several times to find a way to the destination node. Since the greater the number of failed nodes, the greater the number of links is destroyed, it is expected that the increasing number of failure nodes will increase the search cost. Figure 13(b) shows such an effect. The results also further confirm that the higher fanout not only improves the fault tolerance, but also reduces the search cost for the same number of failure nodes. It is be-

**Table 2: Sizes of the smaller partition in the segmented network in case of massive node failure with different fanouts ($m$)**

| | | Number of nodes | | | |
|---|---|---|---|---|---|
| | | 1→2 | 2→10 | 10→20 | >20 |
| Number of occurrences in all test cases | $m$=2 | 12 | 8 | 0 | 0 |
| | $m$=4 | 13 | 6 | 1 | 0 |
| | $m$=6 | 18 | 2 | 0 | 0 |
| | $m$=8 | 11 | 9 | 0 | 0 |
| | $m$=10 | 16 | 3 | 1 | 0 |

cause higher fanout provides higher number of connection paths among nodes. As a result, it is easier to find a connected path between the search node and the destination node. Note that in the figure, the "break points" are points in which the network is segmented. We can notice that the "break points" of network fanouts ranging from 2 to 10 occur from left to right, confirming the resilience of trees with higher fanout.



(a) Percentage of failed nodes before network is partitioned

(b) Search cost in case of massive failure

**Figure 13: Effect of massive failure**

# 7. RELATED WORK

Extensive work has been done to support search in the area of distributed data. Most of them are based on scalable and distributed data structures (SDDS) such as [16]. However, these structures cannot be used in P2P systems where there is neither global index nor centralized servers. Since our paper focuses on search in P2P systems, we only discuss related work in P2P systems.

## 7.1 Single-attribute Query

Existing structured P2P systems can be classified into three categories: distributed hash table based systems, skip list based systems, and tree based systems.

Distributed hash table based systems use distributed hash tables to index data. As a result, they can support exact match queries well. Moreover, distributed hash tables help these systems to distribute the workload among nodes equally. Some of the better known P2P systems that belong to this category are Chord [13], which utilizes a ring structure, CAN [18], which utilizes a multi-dimensional grid structure, Tapestry [22] and Pastry [19], which utilize the Plaxton mesh. Unfortunately, these systems cannot support range queries since distributed hash tables destroy the ordering of data. It means that they cannot support common queries such as "find all research papers published from 1995 to 2000". To support range queries, variants of the distributed hash method are proposed. A locality sensitive hashing, which allows similar ranges to be hashed to the same peer with high probability is proposed in [9]. However, this method can only provide approximate answers. Another approach, which adds the ranges into hash functions, is proposed in [20]. As a result, the system can always find a superset of the range query. Even though this method can provide exact answers, exact query search is not efficient. [3] can also provide exact answers by using locality preserving hashing. However, the load may not be balanced since the workload distribution may be skewed if data is skewed.

Skip list based systems such as Skip Graph [4] and Skip Net [10] are based on skip-list structure. They can support both exact match queries and range queries by partitioning data into ranges of values. However, they cannot guarantee data locality and load balancing in the whole system.

Tree based systems also have their own problems. P-Grid [1], which utilizes a binary prefix tree structure, cannot guarantee the bound of search steps since it cannot control the height of the tree. The scheme proposed in [15] also suffers from the same problem. There, an arbitrary multi-way tree structure is used, in which each node maintains links to its parent, children, sibling and neighbors. P-Tree [8] utilizes the B$^+$-tree structure on top of the CHORD overlay network. In P-Tree, peer nodes are organized as a CHORD ring in which each of them maintains a data leaf node and a left most path from the root to that node of the B$^+$-tree. This leads to significant overhead in building and maintaining consistency of the B$^+$-tree. Specifically, a tree structure has been built for each joining node, and periodically, peer nodes have to exchange their stored B$^+$-tree structure for checking consistency. BATON [11], as mentioned, utilizes a binary balanced tree structure. As a result, it can control the height of the tree, and hence avoid the problem of P-Grid. Nevertheless, similar to other P2P systems, BATON's search cost is bounded at $O(log_2N)$.

## 7.2 Multi-attribute Query

MAAN [6], which extends Chord [13], supports multi-attribute queries by mapping each attribute value to a value in the Chord identifier space via uniform locality preserving hashing. Queries are processed by using single attribute-dominated query resolution approach. It means that only one dominated attribute is used to search while other attributes are carried along the query for filter purpose. As a result, it takes $O(logN + N \cdot s_k)$ routing hops to resolve the query, where $s_k$ is the selectivity of of the query on the dominated attribute $a_k$.

Unlike MAAN, which uses the same identifier space for all attributes, Mercury [5] uses separate identifier spaces, called routing hubs, for separate attributes. Each routing hub is a collection of nodes connected with each other to form an overlay structure. Similar to MAAN, inserted data are indexed on all attributes and queries are processed by a dominated attribute. Consequently, queries are passed to a hub corresponding to the dominated attribute while inserted data are replicated to all routing hubs.

DIM [14] supports multi-attribute queries in sensor networks by considering them as multi-dimensional range queries. It employs a locality preserving geographic hash, which looks like k-d tree, to map multi-dimensional space to two-dimensional space. In fact, DIM is similar to CAN in many ways. The problem of DIM is that the routing cost is bounded at $O(\sqrt{N})$, which is quite expensive. Moreover, DIM is also not efficient if the query only involves a small number of attributes since the projected volume from higher dimensions to two dimensions is rather large. It also suffers from load imbalance when data is skewed.

## 8. CONCLUSION

In this paper, we proposed a balanced multi-way tree structure, BATON*, which allows us to reduce the cost of routing message to $O(log_m N)$. Additionally, by increasing the fanout of the tree, we increase the number of links among nodes, and hence increase the system's fault tolerance. Moreover, increasing the fanout leads to an increase in the number of leaf nodes, which facilitates better load balancing. Over this structure, we proposed a method to support multi-attribute queries efficiently. Our method relies on the construction of multiple independent indices for groups of one or more attributes. We suggest techniques for partitioning attributes into such groups. A careful experimental analysis, on the PlanetLab [7] infrastructure, confirms efficiency of our proposed system.

We have incorporated BATON* into BestPeer[17] and as a future work, we shall evaluate BATON* as part of BestPeer on data sharing applications.

## 9. REFERENCES

[1] K. Aberer. P-Grid: A self-organizing access structure for p2p information systems. In *Proceedings of the 6th CoopIS Conference*, 2001.

[2] F. S. Action. *Numerical Methods that Work.* Harpercollins College Div, 1970.

[3] A. Andrzejak and Z. Xu. Scalable, efficient range queires for grid information services. In *Proceedings of the 2nd International Conference on Peer-To-Peer Computing*, pages 33–40, 2002.

[4] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, 2003.

[5] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *Proceedings of the 2004 ACM SIGCOMM Conference*, 2004.

[6] M. Cai, M. Frank, J. Chen, and P. Szekely. Maan: A multi-attribute addressable network for grid information services. In *Proceedings of the 2003 IPTPS*, 2003.

[7] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: An overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3), 2003.

[8] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-Trees. In *Proceedings of the 7th WebDB*, pages 25–30, 2004.

[9] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*, 2003.

[10] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 2003.

[11] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st VLDB Conference*, pages 661–672, 2005.

[12] H. V. Jagadish, B. C. Ooi, Q. H. Vu, R. Zhang, and A. Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *Proceedings of the 22nd ICDE Conference*, 2006.

[13] D. Karger, F. Kaashoek, I. Stoica, R. Morris, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[14] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, 2003.

[15] C. Y. Liau, W. S. Ng, Y. Shu, K.-L. Tan, and S. Bressan. Efficient range queries and fast lookup services for scalable p2p networks. In *Proceedings of the 2nd DBISP2P*, pages 78–92, 2004.

[16] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - A scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[17] W. S. Ng, B. C. Ooi, and K.-L. Tan. Bestpeer: A self-configurable peer-to-peer system. In *Proceedings of the 18th ICDE Conference*, 2002.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable contentaddressable network. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 161–172, 2001.

[19] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference of Distributed Systems Platforms*, pages 329–350, 2001.

[20] O. D. Sahin, A. Gupta, D. Agrawal, and A. El Abbadi. A peer-to-peer framework for caching range queries. In *Proceedings of the 20th ICDE*, 2004.

[21] C. Schmidt and M. Parashar. Flexible information discovery in decentralized distributed systems. In *Proceedings of the 2003 HPDC*, pages 226–235, 2003.

[22] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report CSD-01-1141, Univ. California, Berkeley, CA, Apr. 2001.