Querying Peer-to-Peer Networks Using P-Trees

Adina Crainiceanu Prakash Linga Johannes Gehrke Jayavel Shanmugasundaram

Department of Computer Science
Cornell University
{adina,linga,johannes,jai}@cs.cornell.edu

ABSTRACT

We propose a new distributed, fault-tolerant peer-to-peer index structure called the **P-tree**. P-trees efficiently evaluate *range queries* in addition to equality queries.

1. INTRODUCTION

Peer-to-Peer (P2P) networks are emerging as a new paradigm for structuring large-scale distributed systems. The key advantages of P2P networks are their scalability, their fault-tolerance, and their robustness, due to symmetrical nature of peers and self-organization in the face of failures. The above advantages made P2P networks suitable for content distribution and service discovery applications [1, 7, 8, 9]. However, many existing systems only support location of data items based on a key value (i.e. equality lookups).

In this paper, we argue for a richer query semantics for P2P networks. We envision a future where users will use their local servers to offer data or services described by semantically-rich XML documents. Users can then *query* this “P2P data warehouse” or ”P2P service directory” as if all the data were stored in one huge centralized database. As a first step towards this goal we propose the P-tree, a new distributed fault-tolerant index structure that can efficiently support *range queries* in addition to equality queries.

As an example, consider a large-scale computing grid distributed all over the world. Each grid node (peer) has an associated XML document that describes the node and its available resources. Specifically, each XML document has an *IPAddress*, an *OSType*, and a *MainMemory* attribute, each with the evident meaning. Given this setup, a user may wish to issue a query to find suitable peers for a main-memory intensive application - peers with a “Linux” operating system with at least 4GB of main memory:

```
for $peer in //peer where $peer/@OSType = 'Linux'
 and $peer/@MainMemory >= 4096
return $peer/@IPAddress
```

A naive way to evaluate the above query is to contact every peer in the system, and select only the relevant peers. However, this approach has obvious scalability problems because all peers have to be contacted for every query, even though only a few of them may satisfy the query predicates. P2P index structures that support only equality queries will also be inefficient here: they will have to contact *all* the peers having “Linux” as the *OSType*, even though a large fraction of these may have main memory less than 4GB.

In contrast, the P-tree supports the above query efficiently as it supports both equality and range queries. In a stable system (no insertions or deletions), a P-tree of order d provides $O(m+log_d N)$ search cost for range queries, where N is the number of peers in the system, m is the number of peers in the selected range and the cost is the number of messages. The P-tree requires $O(d \cdot log_d N)$ space at each peer and is resilient to failures of even large parts of the network. Our experimental results (both on a large-scale simulated network and in a small real network) show that P-trees can handle frequent insertions and deletions with low maintenance overhead and small impact on search performance.

In the rest of the paper we use the following terminology and make the following assumptions. We target applications that offer a single data item or service per peer, such as resource discovery applications for web services or the grid. We call the XML document describing each service a *data item*. Our techniques can be applied to systems with multiple data items per peer by first using a scheme such as [4] to assign ranges of data items to peers, and then considering each range as being one data item. We call the attributes of the data items on which the index is built the *search key* (in our example, the search key is a composite key of the *OSType* and *MainMemory* attributes). For ease of exposition, we shall assume that the search key only consists of a single attribute. The generalization to multiple attributes is similar to B+-tree composite keys [3].

2. THE P-TREE INDEX

The P-tree index structure supports equality and range queries in a distributed environment. P-trees are highly distributed, fault-tolerant, and scale to a large number of peers.

2.1 P-tree: Overview

Centralized databases use the B+-tree index [3] to efficiently evaluate equality and range queries. The key idea behind the P-tree is to maintain *parts* of *semi-independent* B+-trees at each peer. This allows for fully distributed index maintenance, without need for inherently centralized and unscalable techniques such as primary copy replication.

Conceptually, each peer views the search key values as being organized in a ring, with the highest value wrapping around the lowest value (see Figure 1). When constructing its semi-independent B+-tree, each peer views its search key value as being the smallest value in the ring (on a ring, any value can be viewed as the smallest). In a P-tree, each peer stores and maintains only the *left-most root-to-leaf path* of its corresponding B+-tree. Each peer relies on a selected sub-set of other peers to complete its tree.

Seventh International Workshop on the Web and Databases (WebDB 2004), June 17-18, 2004, Paris, France

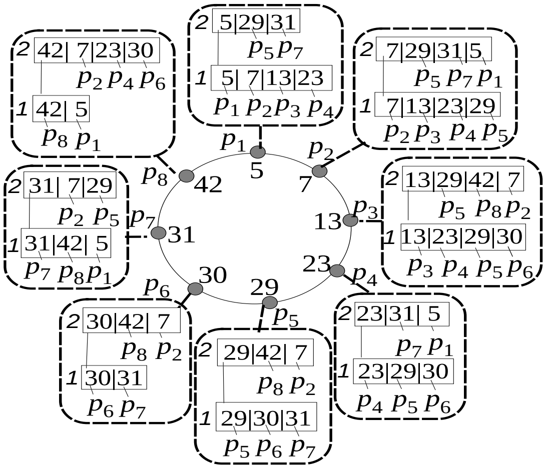

Figure 1: Full P-tree

As an illustration, consider Figure 2. The peer p_1, which stores the item with value 5, only stores the root-to-leaf path of its B+-tree. To complete the remaining parts of its tree - i.e., the sub-trees corresponding to the search key values 29 and 31 at the root node - p_1 simply points to the corresponding nodes in the peers p_5 and p_7 (which store the data items corresponding to 29 and 31, respectively). p_5 and p_7 also store the root-to-leaf paths of their independent B+-trees, as shown in Figure 1, so p_1 just points to the appropriate nodes in p_5 and p_7 to complete its own B+-tree.

To illustrate an important difference between P-trees and B+-trees, consider the semi-independent B+-tree at peer p_1. The root node of this tree has three sub-trees stored at the peers with values 5, 29, and 31, respectively. The first sub-tree covers values in the range 5-23, the second sub-tree covers values in the range 29-31, and the third sub-tree covers values in the range 31-5. These sub-trees have *overlapping* ranges, and the same data values (31 and 5) are indexed by multiple sub-trees. Such overlap is permitted because it allows peers to independently grow or shrink their tree; this in turn eliminates the need for excessive coordination and communication between peers.

The above P-tree structure has the following advantages. First, since the P-tree maintains the B+-tree-like hierarchical structure, it provides $O(log_dN)$ search performance for equality queries in a consistent state. Second, since the order of values in the ring corresponds to the order in the search key space, range queries can be answered efficiently by first finding the smallest value in the range (using equality lookup), and then scanning the relevant portions of the ring. Third, since each peer is only responsible for maintaining the consistency of its leftmost root-to-leaf path nodes, it does not require global coordination and does not need to be notified for every insertion/deletion. Finally, since each peer only stores tree nodes on the root-to-leaf path, and each node has at most $2d$ entries, the total storage requirement per peer is $O(d \cdot log_dN)$.

2.2 P-tree: Structure and Properties

Consider the peer p storing an item with search key value $p.value$. Formally, the data structure for the P-tree nodes at p is a double indexed array $p.node[i][j]$, where $0 \leq i \leq p.maxLevel$ and $0 \leq j < p.node[i].numEntries$. Each entry in this array is a pair $(value, peer)$, which points to the peer storing the data item with search key value $value$. For convenience, we define level 0 in the P-tree at p as having d entries $(p.value, p)$. We define $\texttt{succ}(p)$ to be the peer p' such that $p'.value$ appears right after $p.value$ in the clockwise traversal of the P-tree ring. In Figure 1, $\texttt{succ}(p_1) = p_2$, $\texttt{succ}(p_8) = p_1$, and so on. We similarly define $\texttt{pred}(p)$. To easily reason about the ordering of peers in the P-tree ring, we introduce the comparison operator $<_p$. Intuitively, $<_p$ compares peers on the ring based on their values, by treating $p.value$ as the smallest value. For example, for the $<_{p_3}$ operator, we treat $p_3.value$ as the smallest value in the ring in Figure 1. We thus have $p_6 <_{p_3} p_7$, $p_8 <_{p_3} p_1$, $p_1 <_{p_3} p_2$, and so on. We define the operator $\leq_p$ similarly.

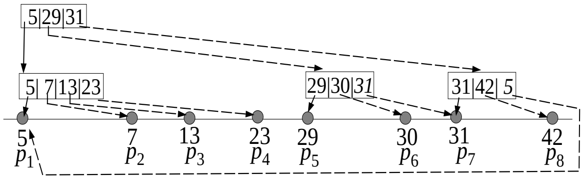

Figure 2: P-tree nodes for p_1's tree

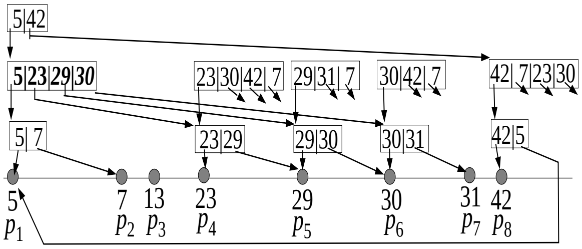

Figure 3: Inconsistent P-tree

We define the "reach" of a node at level i at peer p, denoted $reach(p, i)$. Intuitively, $reach(p, i)$ is the "last" peer that can be reached by following the right-most path in the sub-tree rooted at $p.node[i]$. In Figure 1, $reach(p_1, 2) = p_1$ since the last entry of $p_1.node[2]$ points to $p_7.node[1]$, whose last entry in turn points to p_1. Formally, let $lastPeer(p, i)$ denote the peer in the last entry of $p.node[i]$. Then $reach(p, 0) = p$, and $reach(p, i + 1) = reach(lastPeer(p, i + 1), i)$.

We now define the key properties that characterize a consistent P-tree index. If a P-tree satisfies all of the four properties then it is called *consistent*; else it is called *inconsistent*. Consider a set of peers P, and a P-tree of order $d > 1$.

Property 1 (Number of Entries Per Node) All non-root nodes have between d and $2d$ entries, while the root node has between 2 and $2d$ entries. Formally, for any $p \in P$:
$\forall i < p.maxLevel\ (p.node[i].numEntries \in [d, 2d])$
$p.node[p.maxLevel].numEntries \in [2, 2d]$

Allowing the number of entries in a node to vary makes nodes more resilient to insertions and deletions as the invariant will not be violated for every insertion/deletion.

Property 2 (Left-Most Root-to-Leaf Path) This property captures the intuition that each peer stores the nodes in the left-most root-to-leaf path of its semi-independent B+-tree. In other words, the first entry of every node in a peer p points to p. Formally, for all peers $p \in P$, and for all levels $i \in [0, p.maxLevel]$, $p.node[i][0] = (p.value, p)$.

This condition limits the storage requirements at each peer to be $O(d \cdot log_dN)$. It also prevents the P-tree nodes at a peer from having to be updated after every insertion/deletion.

Property 3 (Coverage) This property ensures that all search key values are indeed indexed by the P-tree; i.e., it ensures that no values are "missed" by the index structure. As an illustration of the type of problem that could occur if the coverage property is not satisfied, consider the example in Figure 3. Peer p_1 has three index nodes. Consider the second level node in p_1 (with entries having values 5, 23, 29, and 30). The sub-tree rooted at the first entry of this node (with value 5) is stored in p_1, and this sub-tree indexes the range 5-7. The sub-tree rooted at the second entry of this node (with value 23) is stored in p_4 and indexes the range 23-29. However, neither of these sub-trees is indexing the value 13. Therefore, if a search is issued at p_1 for the value 13, the index can be used only to reach up to p_2, which stores the value 7. After reaching p_2, the search will have to do a sequential scan along the ring to reach p_3 that stores the data item for value 13. Although p_3 is the successor of p_2 in this example, in general, there could be many "missed" values in between p_2 and p_3, and the search performance can deteriorate due to the long sequential scan along the ring (although the search will eventually succeed).

As illustrated, "gaps" between adjacent sub-trees imply that search cost for certain queries can no longer be guaranteed to be logarithmic. The coverage property addresses this problem by ensuring that there are no gaps between adjacent sub-trees. A similar issue is ensuring that the sub-tree rooted at the last entry of each root node wraps all the way around the P-tree ring. These two properties together ensure that all values are reachable using the index.

Formally, let $p.node[i][j] = (val_j, p_j)$ and $p.node[i][j+1] = (val_{j+1}, p_{j+1})$ be two adjacent entries in the node at level i at peer p. The coverage property is satisfied by this pair of entries iff $p_{j+1} \leq_{p_j} succ(reach(p_j, i-1))$.

The coverage property is satisfied by the root node of a peer p if $p \leq_{veryLastPeer} succ(reach(p, p.maxLevel))$, where $veryLastPeer = lastPeer(p, p.maxLevel)$.

The coverage property is satisfied for the entire P-tree iff the above conditions are satisfied for every pair of adjacent entries and root nodes, for every peer in the system.

As an implementation issue, note that checking the coverage requires only one message if we store an additional entry in each node to estimate the reach of that node.

Property 4 (Separation) The coverage property deals with the case when adjacent sub-trees are too far apart. A different concern arises when adjacent sub-trees overlap. Some overlap is possible and desirable because the sub-trees can then be independently maintained. However, excessive overlap (see the three last entries in the level 2 node at p_1 in Figure 3) can compromise logarithmic search performance. The separation property ensures that the overlap between adjacent sub-trees is not excessive by ensuring that two adjacent entries at level i have at least d non-overlapping entries at level $i-1$. This ensures that the search cost is $O(log_d N)$. Formally, let (val_j, p_j) and (val_{j+1}, p_{j+1}) be two adjacent entries as before. The separation property is satisfied between these two entries iff $p_j.node[i-1][d-1].peer <_{p_j} p_{j+1}$.

The separation property is satisfied for the entire P-tree iff the separation property is satisfied for every pair of adjacent entries for every peer in the system.

3. P-TREE ALGORITHMS

We now describe fully distributed algorithms for searching and updating P-trees. The main challenge is to ensure the consistency of the P-tree in the face of concurrent peer insertions, deletions, and failures. The key idea is to allow the P-tree to be in a state of *local inconsistency*, where some P-tree nodes do not satisfy coverage or separation. Local inconsistency allows searches to proceed correctly, with perhaps a slight degradation in performance[1] even if peers are continually being inserted and deleted from the system. Our algorithms will eventually transform a P-tree from a state of local inconsistency to a fully consistent state.

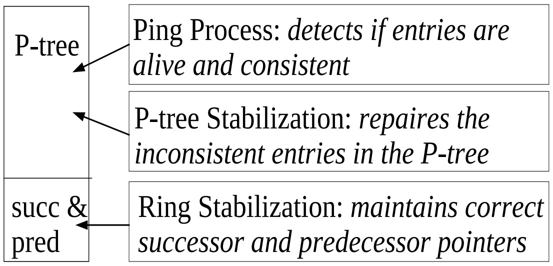
P-tree
succ &
pred
Ping Process: *detects if entries are alive and consistent*
P-tree Stabilization: *repaires the inconsistent entries in the P-tree*
Ring Stabilization: *maintains correct successor and predecessor pointers*

Figure 4: P-tree maintenance

3.1 High-Level System Architecture

Figure 3 shows the high-level architecture of a P-tree component at a peer. The underlying ring structure is maintained by one of the well-known successor-maintenance algorithms from the P2P literature; in our implementation we use the algorithm described in Chord [9]. Thus, the P-tree ring leverages all of the fault-tolerant properties of Chord.

Although the underlying ring structure provides strong fault-tolerance, it only provides linear search performance. The logarithmic search performance of P-trees is provided by the actual P-tree nodes at the higher levels.

The consistency of the P-tree nodes is maintained by two co-operating processes, the *Ping Process* and the *Stabilization Process*. There are independent copies of these two processes that run at each peer. The Ping Process at peer p detects inconsistencies in the P-tree nodes at p and marks them for repair by the Stabilization Process. The Stabilization Process at peer p periodically repairs the inconsistencies detected by the Ping Process. Even though the Stabilization Process runs independently at each peer, we can formally prove that the (implicit and loose) cooperation between peers as expressed in the Ping and Stabilization Process leads eventually to a globally consistent P-tree.

We now describe how peers handle search, insertion of new peers, and deletion of existing peers, and then we describe the Ping Process and the Stabilization Process.

3.2 Search Algorithm

We assume that each query originates at some peer p in the P2P network. The search takes as input the lower-bound (lb) and the upper-bound (ub) of the range query, and the peer where the search was originated. The search procedure at each peer selects the farthest away pointer that does not overshoot lb and forwards the query to that peer. Once the algorithm reaches the lowest level of the P-tree, it traverses the successor list until the value of a peer exceeds ub. At the end of the range scan, a $SearchDoneMessage$ is sent to the peer that originated the search.

Example: Consider the range query $30 \leq value \leq 39$ that is issued at peer p_1 in Figure 1. The search algorithm

[1] We study and quantify this degradation in Section 4.

first determines that the farthest away known peer with a value not overshooting 30 is p_5 with value 29 (the second entry at the second level of p_1's P-tree nodes). The search message is thus forwarded to p_5. p_5 follows a similar protocol, and forwards the search message to p_6 (which appears as the second entry in the first level of p_5's P-tree). Since p_6 stores the value 30, which falls in the desired range, this value is returned to p_1; similarly, p_6's successor (p_7) returns its value to p_1. The search terminates at p_7 as the value of its successor does not fall within the query range.

In a consistent P-tree, the search procedure will go down one level of the P-tree every time a search message is forwarded, until the lowest level is reached. This is similar to the behavior of B+-trees, and guarantees that we need at most $log_d\ N$ steps. If a P-tree is inconsistent, however, the search cost may be more than $log_d\ N$. Note that even if the P-tree is inconsistent, it can still answer queries by using the index to the maximum extent possible, and then sequentially scanning along the ring, as illustrated in the example under Property 3 in Section 2.2 (note that the fault-tolerant ring is still operational even in the presence of failures).

It is important to note that every search query cannot always be guaranteed to terminate in a P2P network. For example, a peer could crash in the middle of processing a query, in which case the originator of the query would have to time out and try the query again. This model is similar with that used in most other P2P systems [7, 8, 9]. We can prove the following properties about search.

Lemma: (Correctness of Search) If we search for a value v that is in the fault-tolerant ring for the entire duration of the search, either v will be found or the search will timeout.

Lemma: (Performance of Search) In a stable network of N peers with a consistent P-tree of order d, a range query that returns m results takes at most $O(m+log_d N)$ messages.

3.3 Peer Insertions and Deletions/Failures

As in many P2P networks, we assume that a new peer p indicates its desire to join the system by contacting an existing peer. p issues a regular query to the existing peer in order to determine p's predecessor, `pred`(p), in the P-tree value ring. There are now three things that need to be done to integrate the new peer p into the system. First, p needs to be added to the virtual ring. This is done by the ring stabilization protocol. Second, the P-tree nodes of p need to be initialized. They are initialized with the P-tree nodes copied from `pred`(p) in which the first entry is replaced with an entry corresponding to p. Finally, some of the P-tree nodes of existing peers may need to be updated to take into consideration the addition of p. This is eventually done by the Ping and Stabilization Processes in the existing nodes.

In a P2P network, peers can leave or fail at any time, without notifying other peers in the system. There are two main steps involved in recovering from such failures/deletions. The first is to update the ring, for which we rely on the standard successor maintenance protocol. The second step is to make existing P-tree tree nodes aware of the deletion/failure. Again, no special action is needed for this step because we just rely on the Ping Process to detect possible inconsistencies and on the Stabilization Process to repair them.

3.4 The Ping Process

The Ping Process runs periodically at each peer; its pseudocode is shown in Algorithm 1. The Ping Process checks whether a peer has been deleted/failed (line 4), and if so, it removes the corresponding entry from the P-tree node and decrements *numEntries* (line 5). The Ping Process also checks whether the entries are consistent with respect to the *coverage* and *separation* properties (line 7). If any entry is inconsistent, its state (a variable associated with each P-tree entry) is set to either `coverage` or `separation`. Note that the Ping Process does not repair any inconsistencies — it merely detects them. Detected inconsistencies are repaired by the Stabilization Process.

Algorithm 1 : p.Ping()

```
1: for $l = 1; l \leq p.maxLevel; l = l + 1$ do
2:     $j = 1$
3:     repeat
4:         if $p.node[l][j].peer$ has failed then
5:             Remove($p.node[l], j$)
6:         else
7:             $p.node[l][j].state =$
               CheckCovSep($p.node[l][j-1], p.node[l][j]$)
8:             j++
9:         end if
10:    until $j \geq p.node[l].numEntries$
11: end for
```

Algorithm 2 : p.Stabilize()

```
$l = 1$
repeat
    root=p.StabilizeLevel($l$)
    $l++$
until (root)
$p.maxLevel = l - 1$
```

3.5 The Stabilization Process

The Stabilization Process is the key to maintaining the consistency of the P-tree. At each peer p, the Stabilization Process wakes up periodically and repairs the tree level by level, from bottom to top (see Algorithm 2), within each level starting at entry 0. This bottom-to-top, left-to-right repair of the tree ensures local consistency: the repair of any entry can rely only on entries that have been repaired during the current period of the Stabilization Process.

Algorithm 3 describes the Stabilization Process within each level of the P-tree data structure at a peer. The first loop from lines 2 to 17 repairs existing entries in the P-tree. If an entry $p.node[l][j]$ is not consistent (with resect to the previous entry), it is repaired by either inserting a new entry, if coverage is violated (line 7), or by replacing the current entry (line 9), in case separation is violated. In both cases, we make a conservative decision: we pick as new entry the *closest* peer to *prevPeer* that still satisfies the *separation* and *coverage* properties. By the definitions in Section 2.2, this is precisely the peer $newPeer =$ `succ`$(prevPeer.node[l-1][d-1].peer)$, which can be determined using just two messages — one to *prevPeer*, and another one to $prevPeer.node[l-1][d-1].peer$. (We can also reduce this overhead to one message by caching relevant entries). After the adjustments in lines 7 or 9, the current entry is consistent. We now have to check whether the pair $(p.node[l][j], p.node[l][j+1])$ satisfies coverage and separation, which happens through function CheckCovSep

Algorithm 3 : p.StabilizeLevel(int l)

```
1:  $j = 1$
2:  while $j < p.node[l].numEntries$ do
3:    if $p.node[l][j]$.state $\neq$ consistent then
4:      prevPeer = $p.node[l][j-1].peer$
5:      newPeer =
        succ($prevPeer.node[l-1][d-1].peer$)
6:      if $p.node[l][j].state$ == coverage then
7:        INSERT($p.node[l]$,$j$,newPeer)
          $p.node[l].numEntries++ (max\ 2d)$
8:      else
9:        REPLACE($p.node[l]$,$j$,newPeer)
10:     end if
11:     $p.node[l][j+1].state$ =
        CheckCovSep($p.node[l][j], p.node[l][j+1]$)
12:   end if
13:   if COVERS($p.node[l][j], p.value$) then
14:     $p.node[l].numEntries = j + 1$
15:   end if
16:   $j$++
17: end while
18: while $\neg$ COVERS($p.node[l][j-1], p.value$)
    $\wedge\ j < d$ do
19:   prevPeer = $p.node[l][j-1].peer$
20:   newPeer =
      succ($prevPeer.node[l-1][d-1].peer$)
21:   INSERT($p.node[l]$,$j$,newPeer)
22:   $j$++
23: end while
24: if COVERS($p.node[l][j-1], p.value$) then
25:   return true
26: else
27:   return false
28: end if
```

in line 11. Line 13 contains a sanity check: if the current entry already wraps around the tree, i.e., its subtree covers up to *p.value*, then this level is the root level, and we can stop (line 14). The loop in lines 18 to 23 makes sure that $p.node[l]$ has at least d entries (unless this is the root level). Lines 24 to 28 return whether this level is the root of the tree. We can prove the following lemma:

Lemma: (Eventual Consistency) Given that the fault-tolerant ring is connected and no peers enter or leave the system after time t, there is a time t_0 such that after time $t + t_0$ the P-tree is consistent.

4. EXPERIMENTAL EVALUATION

We evaluate the performance of P-trees using both a simulation study and a small real distributed implementation.

4.1 Experimental Setup

We wrote (in Java JDK 1.4) a peer-to-peer simulator to evaluate the performance of P-tree over large-scale networks. In the simulator, each peer is associated with a search key value and a unique address. The peer with address 0 is used as the reference peer, and any peer that wishes to join the P2P network contacts this peer. We simulate the functionality of the Ping Process by invalidating/deleting necessary entries before the Stabilization Process runs. We use the *message cost* (the number of messages exchanged between peers for a given operation) as the performance metric.

Parameter	Range	Default
NumPeers	$1,000 - 250,000$	$100,000$
Order	$2 - 16$	4
SPTimePeriod	$1 - 700$	25
IDRatio	$0.001 - 1000$	1

Table 1: Parameters

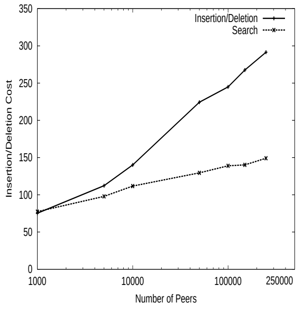

Figure 5: Nb. of peers

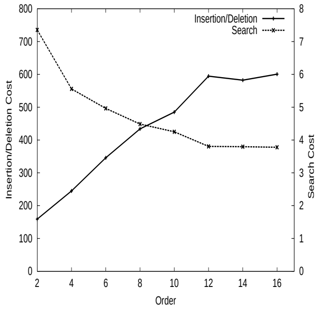

Figure 6: Order

The parameters varied in our experiments are shown in Table 1. *NumPeers* is the number of peers in the system and *Order* is the order of the P-tree. *SPTimePeriod* is the number of operations after which the Stabilization Process is run (on all peers at all required levels). *IDRatio* is the ratio of insert to delete operations in the workload.

For each set of experiments, we vary one parameter and we use the default values for the rest. Since the main component in the cost of range queries is the cost of finding the data item with the smallest qualifying value (the rest of the values are retrieved by traversing the successor pointers), we only measure the cost of equality searches. We calculate the cost of a search operation by averaging the cost of performing a search for a random value starting from every peer. We calculate the insertion/deletion message cost by averaging over 100 runs of the Stabilization Process.

4.2 Experimental Results

Varying Number of Peers Figure 5 shows the message cost for insertion/deletion and search operations, when the number of peers is varied. The scale for the update operations is on the left side and the one for search operations is on the right side. The search cost increases logarithmically with the number of peers (note the log scale on the x-axis). The logarithmic performance is to be expected as the height of the P-tree increases logarithmically with the number of peers. The figure also shows that the message cost for insertions and deletions is a small fraction of the total number of

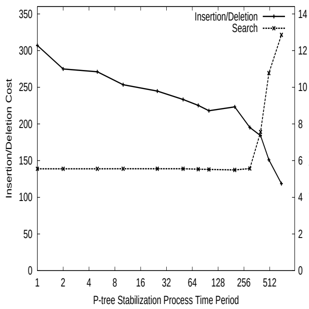

Figure 7: SP frequency

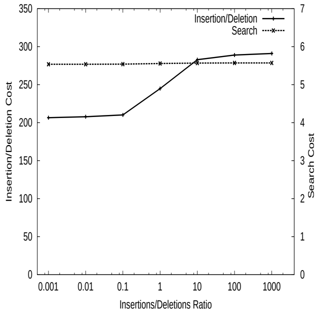

Figure 8: I/D ratio

peers in the system, implying that the effects of insertions and deletions are highly localized. In particular, the message cost for insertions and deletions also increases roughly logarithmically with the number of peers.

Varying Order Figure 6 shows the effect of varying the order of the P-tree. As expected, the search cost decreases with increasing order because each subtree is wider, which in turn reduces the height of the entire tree. The cost for insertions/deletions, on the other hand, increases because each peer has a larger number of entries (note that the number of entries per peer is bounded by $2d \cdot log_d N$, which is strictly increasing for $d > 2$). Thus the associated cost of maintaining the consistency of these entries in the presence of peer insertions/deletions increases. The implication of this result is that P-trees having high orders are not likely to be very practical. This is in contrast to B+-trees, where higher order reduces both search and insertion/deletion cost.

Varying Stabilization Process Frequency Figure 7 shows the effects of varying the frequency at which the Stabilization Process is invoked. When the Stabilization Process is called relatively infrequently, the search cost increases because large parts of the trees are inconsistent. However, the cost per insertion/deletion decreases because the overhead of calling the Stabilization Process is amortized over many insertions and deletions. This illustrates a clear tradeoff in deciding the frequency of the Stabilization Process (and similarly the Ping Process) - frequent invocations of the Stabilization Process will decrease the search cost, but will increase the overhead of maintaining the P-tree structure in the face of multiple insertions/deletions.

Varying Insertions/Deletions Ratio Figure 8 shows the results of varying the ratio of insert to delete operations. The cost per operation is higher when there are more insertions. This is attributable to the fact that we run our experiments after building a tree of 100,000 peers. Since a growing tree is likely to have a high fill factor, there is a higher likelihood of an overflow due to an insertion, as opposed to an underflow due to a deletion. When we ran experiments on a shrinking tree (not shown), deletions had a higher message cost.

4.3 Results from a Real Implementation

We now present some preliminary results from a real distributed implementation of P-trees. Our implementation was done using C#, and we implemented the full functionality of P-trees, including the Ping and Stabilization algorithms. Our experiments were done on six 2.8GHz with 1GB of RAM Pentium IV PCs connected via a LAN. We varied the number of ”virtual peers” from 20 to 30. We mapped 5 virtual peers to each of the 4-6 physical machines. Each virtual peer had associated a unique search key value. The virtual peers communicated using remote-procedure calls (RPCs). We set up the Ping Process and the Stabilization Process to run once per second at each virtual peer. We used the elapsed (wall-clock) time as our performance metric, and each result was averaged over 5 independent runs.

The experimental results are shown in Table 2. As shown, the average search time for a single data item in a fully consistent P-tree is about 0.044s, for 20 to 30 virtual peers. The average search time with a failure of 25% of the virtual peers (uniformly distributed in the value space) is also relatively stable at about 3s. The time for the P-tree to stabilize to a fully consistent state after failures varies from 13-19s. The search and stabilize times are of the order of seconds because we run the periodic processes only once per second.

Real (Virtual) Peers	**4 (20)**	**5 (25)**	**6 (30)**
Search (stable)	0.044s	0.043s	0.043s
Search (inconsistent)	3.085s	2.976s	2.796s
Stabilization	13.25s	19s	17.25s

Table 2: Experimental Results

5. RELATED WORK

Systems like [7, 8, 9] implement distributed hash tables to provide efficient equality lookup. The hash function used by these systems destroys the order in the key value space, making them impractical for processing range queries. Moreover, these systems rely on the fact that values are almost evenly distributed in the indexing space and they cannot be easily adapted to a skewed distribution. The P-trees can be viewed as a *non-trivial* adaptation of Chord to skewed data distribution. [5] uses order-preserving hash functions to process range queries. However, they can only provide approximate answers, while the P-trees provide exact answers to range queries. Distributed database index structures (e.g. [6]) are inadequate in a P2P framework as they do not allow peers to leave the system at will. Skip Graphs [2] support range queries, but they only provide probabilistic guarantees even when the index is fully consistent. Moreover, the search performance of the Skip Graphs is $O(d \cdot log_d N)$ while the search performance of the P-trees is only $O(log_d N)$. The scheme presented in [4] supports range queries, but the performance of the system depends on certain heuristics for insertions. The search performance can be linear in the worst case even with a consistent index.

6. CONCLUSION

We have proposed the P-tree index, a novel P2P index structure well suited for applications such as resource discovery for web services and the grid, by supporting range queries in addition to equality queries. Results from our simulation study and real implementation show that P-trees efficiently support search, insertion and deletion, with average message cost per operation being approximately logarithmic in the number of peers.

7. REFERENCES

[1] K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *CoopIS*, 2001.

[2] J. Aspnes and G. Shah. Skip graphs. In *SODA*, 2003.

[3] D. Comer. The ubiquitous b-tree. In *Computing Surveys, 11(2)*, 1979.

[4] A. Daskos et al. Peper: A distributed range addressing space for p2p systems. In *DBISP2P*, 2003.

[5] A. Gupta et al. Approximate range selection queries in peer-to-peer systems. In *CIDR*, 2003.

[6] D. B. Lomet. Replicated indexes for distributed data. In *PDIS*, 1996.

[7] S. Ratnasamy et al. A scalable content-addressable network. In *SIGCOMM*, 2001.

[8] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[9] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.