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

Peer-to-peer (P2P) systems provide a robust, scalable and decentralized way to share and publish data. However, most existing P2P systems only provide a very rudimentary query facility; they only support equality or keyword search queries over files. We believe that future P2P applications, such as resource discovery on a grid, will require more complex query functionality. As a first step towards this goal, we propose a new distributed, fault-tolerant P2P index structure for resource discovery applications called the **P-tree**. P-trees efficiently evaluate *range queries* in addition to equality queries. We describe algorithms to maintain a P-tree under insertions and deletions of data items/peers, and evaluate its performance using both a simulation and a real distributed implementation. Our results show the efficacy of our approach.

1. INTRODUCTION

Peer-to-peer (P2P) systems are emerging as a new paradigm for structuring large-scale distributed computer systems. Some of the key advantages of P2P systems are (a) their scalability, due to resource-sharing among cooperating peers, (b) their fault-tolerance, due to the symmetrical nature of peers and lack of centralized control, and (c) their robustness, because of self-organization in the face of peer and network failures.

Due to the above advantages, P2P systems have made inroads as scalable content distribution networks [24, 27, 25]. Most of these systems provide one main search functionality, namely, efficient location of data items based on a key value. In database terms, this corresponds to supporting simple equality selection queries on an attribute value.

In this paper, we argue for a much richer query semantics for P2P systems. We envision a future where users will use their local servers to publish data or services as semantically-rich XML documents. Users can then *query* this "P2P data warehouse" as if the data were stored in one huge centralized database system. Note that developing such a query infrastructure goes well beyond the capabilities of current distributed database systems [16] because of the scale, dynamics, and physical distribution of P2P systems [10].

As a first step towards the goal of supporting complex queries in a P2P system, we propose a new distributed fault-tolerant P2P index structure called the P-tree (for **P**2P-**tree**). The P-tree is the P2P equivalent of a B+-tree, and can efficiently support *range queries* in addition to equality queries. The P-tree is designed for application scenarios where there is one or a few data items per peer. One such application is resource discovery in a large grid. Here each participating grid node (peer) has a data item that describes its available resources. Users can then issue queries against these data items to find the grid nodes that satisfy their resource demands. Another related application is resource discovery for web services.

In a stable system without insertions or deletions, a P-tree of order d provides $O(log_d N)$ search cost for equality queries, where N is the number of data items in the system. For range queries, the search cost is $O(m + log_d N)$, where m is the number of data items in the selected range. The P-tree only requires $O(d \cdot log_d N)$ space at each peer, but is still very resilient to failures of even large parts of the network. In particular, our experimental results show that even in the presence of many insertions, deletions and failures, the performance of queries degrades only slightly. The paper makes the following specific contributions:

- We introduce P-trees, a new index for evaluating equality and range queries in a P2P system (Section 3).
- We describe algorithms for maintaining P-trees in a dynamic P2P environment, where data items/peers may inserted and deleted frequently (Section 4).
- We show the results of a simulation study of P-trees in a large-scale P2P network. Our results indicate that P-trees can handle frequent data item/peer insertions and deletions with low maintenance overhead and a small impact on search performance. We also present preliminary experimental results from a real (albeit small) distributed implementation (Section 5).

2. SYSTEM MODEL AND AN EXAMPLE

In this section we define our P2P system model, and illustrate it with an example.

2.1 Peers

A *peer* is a semi-autonomous processor that has two logical storage partitions. The first partition is used to store the data items that a peer wishes to publish to other peers; we call this the *data partition*. The second partition is "shared" space that is used to store the distributed data structure for speeding up the evaluation of user queries; we call this the *index partition*. The peers are semi-autonomous because they have full control over the data partition (i.e., they can arbitrarily insert, modify or delete data items in the data partition), while they have no control over the index partition. The content of the index partition is maintained by

the distributed indexing protocol, which we will describe in Section 4.

2.2 Peer-to-Peer Networks

A *peer-to-peer (P2P) network* is a collection of peers. We assume that there is some underlying network protocol that can be used to send messages from one peer to another (this protocol could be TCP for the Internet). One of the key features of a P2P network is that peers can arrive and leave at any time. A peer can join a P2P network by contacting some peer that is already part of the network. A peer can leave the network at any time without contacting any other peer; this models peer crashes and unpredictable network failure. We will talk in the remainder of the paper about peers maintaining "pointers" to other peers. Our pointers should be understood as network addresses, as we assume that the network is completely distributed.

2.3 Data Model

For ease of presentation, we assume that the data items conform to the relational data model. We also assume that all the relational tuples conform to the same global schema. Integrating different schemas is a hard and interesting problem, but is orthogonal to our indexing problem. We assume that in the cases where peers do in fact have data conforming to different schemas, some P2P schema mediation technique such as [7] is used to map them into a global schema.

We target applications that have a single data item per peer, such as resource discovery applications where each data item in a peer describes the nature of available resources in that peer. For most of this paper, we thus assume that there is a single tuple stored in the data partition of each peer. Our proposed techniques can be extended to the case where multiple tuples are stored in a peer by creating a "virtual peer" for each distinct tuple stored in a peer. Since each virtual peer has exactly one tuple (by definition), we can use the proposed algorithms by replacing "virtual peers" in place of regular peers. Some performance optimizations are also possible for virtual peers, and we return to this issue in Section 5 in the context of our real implementation.

Our proposed techniques do not scale to a large number of tuples per peer. Thus, to handle other applications that fall in this part of the design space, we need to devise new methods that will reduce the number of virtual peers in the system. We plan to explore this issue as part of future work.

2.4 System Evaluation Model

A P2P index structure needs to support the following operations: search, insertion of a data item/peer, deletion of a data item/peer. We consider an update to be a deletion followed by an insertion. We use the number of messages exchanged between peers as the primary performance metric for the above operations, since we expect the message cost to dominate other costs in a P2P system. This is consistent with other published work on P2P systems (e.g., [27, 25]). A secondary metric of interest is the amount of space required in the index partition of each peer.

2.5 An Illustrative Example

Consider a large-scale computing grid distributed all over the world. Each grid node has a record in its data partition that describes its available resources. The records are tuples that conform to the following relational schema: *Resources(IPAddress, OSType, MainMemory, CPUSpeed, Load, ...)* with *IPAddress* being the IP address of the grid node, *OSType* indicating the operating system (such as Linux, MACOs, ...), *MainMemory* showing how much main memory is available in MB, and *Load* showing the current Load on this grid node. Given this setup, a user may wish to issue a query to find suitable grid nodes for a main-memory intensive application: grid nodes with a "Linux" operating system with more than 4GB of main memory. This could be written as the following SQL query.

```
Select R.*
From   Resources R
Where  R.OSType = ''Linux'' and
       R.MainMemory >= 4096
```

A naive way to evaluate this query is to contact every peer (grid node) in the system, and select the relevant records from each peer. 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 have the relevant data.

In contrast, if we have a P-tree index built on the composite key (R.OSType, R.MainMemory), we can answer the above query efficiently. (Note that the distributed P-tree index structure will be stored in the index partitions of the peers.) In particular, only a logarithmic number of peers in addition to those that actually contain the desired data items will be contacted.

From the above example, it is also easy to see how P-tree are more efficient than P2P index structures that only support equality queries [27, 24, 25, 2]. In the above example, index structures that only support equality queries will have to contact *all* the grid nodes having "Linux" as the OSType, even though a large fraction of these may only have main memory less than 4GB. Another application of the P-tree is resource discovery for web services, where the capability of executing range queries efficiently can result in similar performance improvements.

3. THE P-TREE INDEX

We now propose the P-tree index structure. P-trees support the same class of queries as a centralized B+-tree index, but are highly distributed, fault-tolerant, and scale to a large number of peers. In a stable system without insertions or deletions, a P-tree of order d provides $O(log_d N)$ search cost for equality queries, where N is the number of data items in the system. For range queries, a stable P-tree provides a search cost of $O(m + log_d N)$, where m is the number of data items selected in the range. Our experimental results indicate that this search performance degrades only slightly even in the presence of peer insertions, deletions, and failures. The space requirements for a P-tree index structure is $O(d \cdot log_d N)$ at each peer.

For ease of exposition, we only consider single-attribute index keys (i.e., we do not consider composite keys). The extension to composite keys is similar to that for a B+-tree, and is easy to see once we introduce the P-tree. Also, like a B+-tree, a P-tree can also be used to support *prefix matches*, and *one-dimensional nearest neighbor queries*. We assume that the indexed attribute values are unique; duplicate index values can be made unique by appending the physical id of the peer where the duplicate value resides. We will refer to

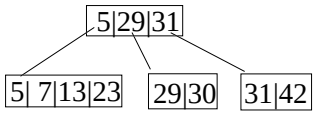

Figure 1: p_1's B+-tree

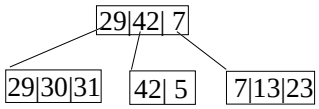

Figure 2: p_5's B+-tree

the value of the indexing attribute in a tuple stored at a peer as the *index value* of the peer. Finally, we assume that each query originates in one of the peers that are already part of the P2P network.

The rest of this section is organized as follows. We first describe B+-trees, and argue why prior work on distributed B+-trees cannot be used in a P2P environment. We then present the P-tree index structure.

3.1 B+-trees

The B+-tree index [5] is widely used for efficiently evaluating equality and range queries in centralized database systems. A B+-tree of order d is a balanced search tree in which the root node of the tree has between 2 and $2d$ entries. All non-root nodes of the tree have between d and $2d$ entries. This property ensures that the height of a B+-tree is at most $\lceil log_d N \rceil$, where N is the number of data items being indexed. Figure 1 shows an example B+-tree of order 2.

Equality search operations in a B+-tree proceed from the root to the leaf, by choosing the sub-tree that contains the desired value at each step of the search. The search cost is thus $O(log_d N)$, which is the same as the height of the tree. Range selections are evaluated by first determining the smallest value in the range (using equality selection), and then sequentially scanning the B+-tree leaf nodes until the end of the range. The search performance of range queries is thus $O(m + log_d N)$, where m is the number of data items selected in the query range.

Unfortunately, existing work on distributed B+-trees is not directly applicable in a P2P environment. To the best of our knowledge, all such index structures [14, 17] try to maintain a globally consistent B+-tree by *replicating* the nodes of the tree across different processors. The consistency of the replicated nodes is then maintained using *primary copy* replication. Relying on primary copy replication creates both scalability (load/resource requirements on primary copy) and availability (failure of primary copy) problems, and is clearly not a solution for a large-scale P2P systems with thousands of peers. We thus need to relax these stringent requirements of existing work, and P-trees are a first attempt at a specific relaxation.

3.2 P-trees: Overview

The key idea behind the P-tree is to give up the notion of maintaining a globally consistent B+-tree, and instead maintain *semi-independent* B+-trees at each peer. Maintaining semi-independent B+-trees allows for fully distributed index maintenance, without any need for inherently centralized and unscalable techniques such as primary copy replication.

To motivate the discussion of semi-independent B+-trees, we first introduce fully independent B+-trees in a P2P setting. Fully independent trees have excessive space cost and high maintenance overhead, but serve as a useful stepping stone in our discussion.

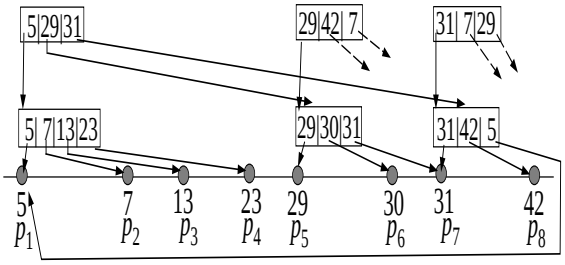

Figure 3: P-tree

3.2.1 Fully Independent B+-trees

Each peer maintains its own independent B+-tree, and each B+-tree is periodically updated as peers/data items are inserted/deleted from the system. As an illustration, let us assume that the data items with index values 5, 7, 13, 23, 29, 30, 31, 42 are stored in peers p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, respectively. An independent B+-tree maintained at p_1 is shown in Figure 1. In this tree, only the tuple corresponding to the left-most leaf value (5) is actually stored at p_1; the other leaf entries are "pointers" to the peers that have the corresponding data items.

As another illustration, the B+-tree stored in p_5 is shown in Figure 2. Here, p_5 views the index values as being organized on a ring, with the highest index value wrapping around to the lowest index value. In this ring organization, p_5 views the index value of its locally stored tuple (29) as the smallest value in the ring (note that in a ring, any value can be viewed as the smallest value). As before, only the tuple corresponding to the left-most leaf value is actually stored at p_5, and the other leaf entries are pointers to peers that have the corresponding data items. Note that the B+-trees stored at the peers p_1 and p_5 are completely independent, and have no relationship to each other except that they all index the same values.

Since peers have independent B+-trees, they can maintain their consistency in a fully distributed fashion. However, this approach suffers from the following drawbacks. First, since each peer indexes *all* data values, every peer has to be notified after every insertion/deletion - which is clearly not a scalable solution. Second, the space requirement at each node is large - $O(N)$, where N is the number of data items.

3.2.2 P-tree = Semi-Independent B+-trees

We now introduce the P-tree as a set of semi-independent B+-trees. Even though the B+-trees in a P-tree are only semi-independent (as opposed to fully independent), they allow the index to be maintained in a fully distributed fashion. They also avoid the problems associated with fully independent B+-tree.

The key idea is the following. At each peer, only the *left-most root-to-leaf path* of its corresponding independent B+-tree is stored. Each peer then relies on a selected sub-set of other peers to complete the remaining (non root-to-leaf parts) of its tree.

As an illustration, consider Figure 3. The peer p_1, which stores the tuple with index value 5, only stores the root-to-leaf path of its independent B+-tree (see Figure 1 for p_1's

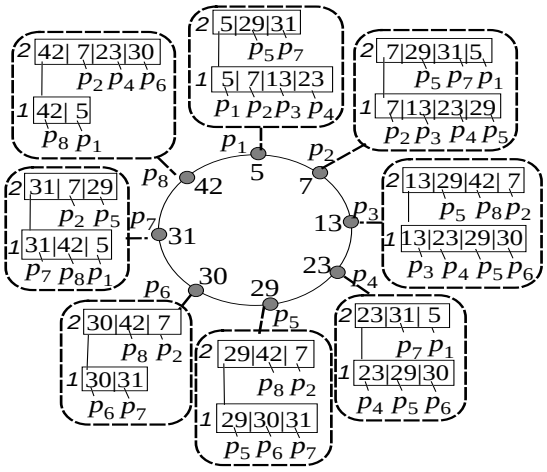

Figure 4: Full P-tree

full independent B+-tree). To complete the remaining parts of its tree - i.e., the sub-trees corresponding to the index values 29 and 31 at the root node - p_1 simply points to the corresponding B+-tree nodes in the peers p_5 and p_7 (which store the tuples corresponding to the index values 29 and 31, respectively). Note that p_5 and p_7 also store the root-to-leaf paths of their independent B+-trees (see Figure 2 for p_5's full independent B+-tree). Consequently, p_1 just points to the appropriate B+-tree nodes in p_5 and p_7 to complete its own B+-tree.

It is instructive to note the structure of P-trees in relation to regular B+-trees. Consider the semi-independent B+-tree at the peer p_1. The root node of this tree has three subtrees 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. Note that these sub-trees have *overlapping* ranges, and the same data values (31 and 5) are indexed by multiple sub-trees. This is in contrast to a regular B+-tree (see Figure 1), where the sub-trees have non-overlapping ranges. We allow for such overlap in a P-tree because this allows each peer to independently grow or shrink its tree in the face of insertions and deletions; this in turn eliminates the need for excessive coordination and communication between peers.

The above structure of P-trees has the following advantages. First, since each peer only stores tree nodes on the leftmost root-to-leaf path, it stores $O(log_d N)$ nodes, where N is the number of data items and d is the order of the P-tree. Since each node has at most $2d$ entries, the total storage requirement per node is $O(d \cdot log_d N)$ entries. Second, since each peer is solely responsible for maintaining the consistency of its leftmost root-to-leaf path nodes, it does not require global coordination among all the peers and does not need to be notified for every insertion/deletion.

For ease of exposition, in Figure 3, we have only shown (parts of) the semi-independent B+-trees in some of the peers. In a full P-tree, each peer has its own root-to-leaf path nodes, which in turn point to nodes in other peers. The full P-tree for our example is shown in Figure 4. Note that the values are organized as a ring because each peer views its locally stored value as the smallest value when maintaining its semi-independent B+-tree.

3.3 P-tree: Structure and Properties

We now formally define the structure of P-trees. We also outline the key properties that a P-tree needs to satisfy in order to ensure the correctness and logarithmic search performance of queries. We discuss algorithms for maintaining these properties in a fully decentralized fashion in Section 4.

3.3.1 P-tree: Structure

Consider the P-tree nodes stored in a peer p, which stores the index value $p.value$. p has possibly many index nodes corresponding to the path from the root to the leaf of its semi-independent B+-tree. We denote the height (number of levels) of p's tree by $p.numLevels$, and use $p.node[i]$ to refer to the index node at level i in p. Each node has possibly many entries. Each entry is the pair $(value, peer)$, which points to the peer $peer$ with index value $peer.value$. We use $p.node[i].numEntries$ to denote the number of entries in the node at level i in peer p, and we use $p.node[i][j]$ to refer to the jth entry of this node. For notational convenience, we define level 0 in a P-tree at peer p as having the d entries $(p.value, p)$.

As an illustration, consider the P-tree nodes of peer p_1 in Figure 4. Since p_1's tree has two nodes, the height of its tree is 2, and thus $p_1.numLevels = 2$. The node at level 1 (the lowest level) has 4 entries corresponding to the pairs $(5, p_1)$, $(7, p_2)$, $(13, p_3)$, $(23, p_4)$. Thus $p_1.node[1].numEntries = 4$, $p_1.node[1][0] = (5, p_1)$, $p_1.node[1][1] = (7, p_2)$, etc.

For notational convenience, we introduce the following notions. Given a peer p, we define $succ(p)$ to be the peer p' such that $p'.value$ appears right after $p.value$ in the P-tree ring. For example, in Figure 4, $succ(p_1) = p_2$, $succ(p_8) = p_1$, and so on. We similarly define $pred(p)$ to be the peer p' such that $p'.value$ appears right before $p.value$ in the P-tree ring. For example, in Figure 4, $pred(p_1) = p_8$, $pred(p_8) = p_7$, and so on.

In order to easily reason about the ordering of peers in the P-tree ring, we introduce the comparison operator $<_p$, where p is a peer. Intuitively, $<_p$ is a comparison operator that compares peers on the P-tree ring based on their index values, by treating $p.value$ as the smallest value in the ring. For example, for the comparison operator $<_{p_3}$, we treat $p_3.value$ as the smallest value in the ring in Figure 4. We thus have $p_6 <_{p_3} p_7$, $p_8 <_{p_3} p_1$, $p_1 <_{p_3} p_2$, $p_3 <_{p_3} p_2$, and so on. We define the operator $\leq_p$ similarly.

It is also useful to 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 reach by following the right-most path in the sub-tree rooted at $p.node[i]$. For example, in Figure 3, $reach(p1, 1) = p_4$ since the last entry of $p_1.node[1]$ points to p_4. Similarly, $reach(p_1, 2) = p_1$ since the last entry of $p_1.node[2]$ points to $p_7.node[1]$, whose last entry in turn points to p_1. We now give a formal (recursive) definition of $reach$. Let $lastEntry(p.node[i])$ denote the last entry of $p.node[i]$. Then $reach(p, 0) = p$, and $reach(p, i + 1) = reach(lastEntry(p.node[i + 1]).peer, i)$.

3.3.2 P-tree: Properties

We now define four key properties that characterize a consistent (distributed) P-tree index. If a P-tree satisfies all of the four properties at every peer, then it is called *consistent*; else it is called *inconsistent*. Consider a set of peers P, and a P-tree of order d.

Property 1 (Number of Entries Per Node) All non-

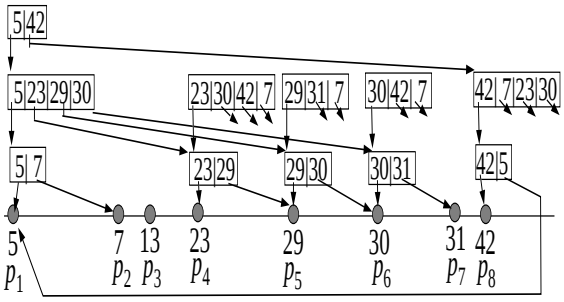

Figure 5: Inconsistent P-tree

root nodes have between d and $2d$ entries, while the root node has between 2 and $2d$ entries. Formally, for all peers $p \in P$, the following conditions hold:

$\forall i < p.numLevels\ (p.node[i].numEntries \in [d, 2d])$

$p.node[p.numLevels].numEntries \in [2, 2d]$

The motivation for these conditions is similar to that in B+-trees [5]. Allowing the number of entries to vary from d to $2d$ for non-root nodes makes them more resilient to insertions and deletions because 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 (Section 3.2.2). 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 $i \in [0, p.numLevels]$, the following condition holds:

$p.node[i][0] = (p.value, p)$

As discussed earlier (Section 3.2.2), this condition limits the storage requirements at each peer to be $O(log_d N)$, and 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 index values are indeed indexed by the P-tree; i.e., it ensures that no values are "missed" by the index structure. In a centralized B+-tree, this is usually not a problem. However, this becomes an issue in a distributed index structure, where different parts of the distributed index could evolve independently during insertions and deletions.

As an illustration of the type of problem that could occur if the coverage property is not satisfied, consider the example in Figure 5. Peer p_1 has three levels in its part of the P-tree. Consider the second level node in p_1 (with entries having index values 5, 23, 29, and 30). The sub-tree rooted at the first entry of this node (with index 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 index value 23) is stored in p_4 and indexes the range 23-29. However, neither of these sub-trees index the value 13. Therefore, if a search is issued for the value 13 in p_1, the index can be used only to reach up to p_2, which stores the value 7 (7 is the largest value less than 13 that is indexed). After reaching p_2, the search will have to do a sequential scan around the ring to reach p_3 that contains the value 13. Note that although p_3 is the immediate predecessor 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 above, "gaps" in between adjacent sub-trees (in the above example, the gap was the peer p_3 having index value 13) implies that search cost for certain queries can no longer be guaranteed to be logarithmic in the number of data items. The coverage property addresses this problem by ensuring that there are no gaps between adjacent sub-trees. A related issue is ensuring the the sub-tree rooted at the last entry of each root node indeed wraps all the way around the P-tree ring. These two properties together ensure that every index value is indeed 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 in level i of peer p. The coverage property is satisfied between these two pairs of entries iff the following condition holds.

$p_{j+1} \leq_{p_j} succ(reach(p_j, i-1))$

The coverage property is satisfied by the root node of a peer p if the following condition holds. In the definition, we let $lastPeer = lastEntry(p.node[p.numLevels]).peer$.

$p \leq_{lastPeer} succ(reach(p, p.numLevels))$

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.

Property 4 (Separation) The coverage property discussed above deals with the case when adjacent sub-trees are too far apart. A different concern arises when adjacent sub-trees overlap. As mentioned in Section 3.2.2, some overlap among the sub-trees is possible, and this is desirable because the sub-trees can then be independently maintained. However, excessive overlap can compromise logarithmic search performance. The separation property ensures the the overlap between successive sub-trees is not excessive.

As an illustration of the kinds of problems that could arise if the separation property is not enforced, consider again Figure 5. Consider the node at the second level in p_1's part of the tree (the index values are 5, 23, 29, and 30). The subtree rooted at the entries 23, 29, and 30 cover the ranges 23-29, 29-30, and 30-31, respectively. Note that each of these sub-trees only have one non-overlapping value with its adjacent sub-tree. Due to this excessive overlap, the height of the corresponding P-tree of order d ($d = 2$ in this case) can no longer be guaranteed to be $O(log_d N)$. (Note that our example is too small to illustrate this increase in height, so we have just illustrated this issue by showing the excessive overlap.) Consequently, if there is excess overlap, search cost can no longer be guaranteed to be logarithmic in the number of data items (although searches will still succeed).

The separation property avoids excessive overlap between adjacent sub-trees 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 $p.node[i][j] = (val_j, p_j)$ and $p.node[i][j+1] = (val_{j+1}, p_{j+1})$ be two adjacent entries in the node in level $i > 0$ in the index of peer p. The separation property is satisfied between these two pairs of entries iff the following condition holds.

$p_j.node[i-1][d-1].peer <_{p_j} p_{j+1}$

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.

Based on the above four properties, we can prove the following statements about P-trees.

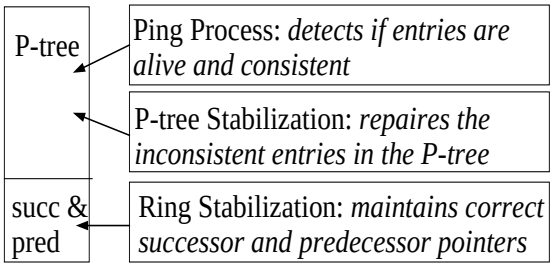

Figure 6: P-tree maintenance

Lemma 1: (Correctness of Search) In a consistent P-tree, a search for index value v succeeds iff a tuple with index value v is stored in a peer in the system.
Lemma 2: (Logarithmic Search Performance) In a consistent P-tree of order d with N data items, the search cost for a range query that returns m results is $O(m + log_d N)$.
Lemma 3: (Logarithmic Space Requirement) In a consistent P-tree of order d with N data items, the space requirements at each peer is $O(d \cdot log_d N)$.

4. P-TREE ALGORITHMS

We now describe algorithms for searching and updating P-trees. The main challenge is to ensure that a P-tree is consistent (i.e., satisfies Properties 1 through 4 in Section 3.3.2), even in the face of concurrent peer insertions, deletions, and failures. Recall that centralized concurrency control algorithms, or distributed algorithms based on primary copy replication, are not applicable in a P2P setting because peers may enter and leave the system frequently and unpredictably. Consequently, we need to devise fully distributed algorithms that can maintain the consistency of a P-tree in a large-scale P2P system.

The key idea is to allow nodes to be in a state of *local inconsistency*, where for a peer p the P-tree at p does 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, without any need for global communication or coordination.

4.1 High-Level System Architecture

Figure 6 depicts the high-level architecture of a P-tree component at a peer. The underlying ring structure of the P-tree 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 [27]. Thus, the P-tree ring leverages all of the fault-tolerant properties of Chord, as summarized in the following lemma [27].
Chord Lemma: (Fault-tolerance of Ring) If we use a successor list of length $O(logN)$ in the ring, where N is the number of peers, and then every peer fails with probability 1/2, then with high probability, find_successor (of a peer in the ring) returns the closest living successor.

[1]We study and quantify this degradation in Section 5.

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 Stabilization Process leads eventually to a globally consistent P-tree (see Section 4.6).

Since a P-tree can be locally inconsistent, we add a state variable to each node entry to indicate whether that entry is consistent or not. We use $p.node[i][j].state$ to refer to the state variable of the jth entry in the ith level node in peer p. The state variable $p.node[i][j].state$ can take on three values, `consistent`, `coverage`, or `separation`, indicating that $p.node[i][j]$ is either in a consistent state, violates the coverage property, or violates the separation property, respectively. The state variable is updated by the Ping Process and the Stabilization Process, and is also used by the latter.

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. When reading these sections, we ask the reader to note the beauty of the conceptual separation of *detecting changes*, and *repairing the P-tree data structure*. Specifically, during insertions, deletions, and failures, P-trees only detect changes and record them without repairing a possibly inconsistent P-tree data structure — this permits us to keep the insertion and deletion algorithms very simple (i.e., they only affect the ring level of the P-tree). Detection of changes is confined to the Ping Process which periodically runs at a peer and only *detects* entries where the local P-tree data structure is inconsistent. The Stabilization Process is the only process that actually *repairs* the P-tree data structure. The Stabilization Process investigates every entry that is not marked `consistent` and repairs the entry such that Properties 1 to 4 in Section 3.3.2 are again satisfied.

4.2 Search Algorithm

For search queries, we assume that each query originates at some peer p in the P2P network. The search takes as input the the lower-bound (lb) and the upper-bound (ub) of the range query, and the peer where the search was originated; the pseudo-code of the algorithm is shown in Algorithm 1. The search procedure at each peer is similar to B+-trees, except that traversal from one level to the next requires communication with another peer (lines 15–16). Once the search algorithm reaches the lowest level of the P-tree, the ring-level, it traverses the successor list until the value of a peer exceeds ub (lines 8–13). At the end of the range scan, a $SearchDoneMessage$ is sent to the peer that originated the search (line 12). Note that we ignore the state of the entries during search.

Example: Consider the range query $30 \leq value \leq 39$ that is issued at peer p_1 in Figure 4. The search algorithm first determines the largest P-tree level in p_1 that contains

Algorithm 1 : p.Search(int lb, int up, $originator$)

```
1: // Find maximum P-tree level that contains an
2: // entry that does not overshoot lb.
3: Find the maximum level l such that ∃ j > 0
   such that p.node[l][j].value ∈ (p.value, lb].
4: if no such level exists then
5:    if lb ≤ p.value ≤ ub then
6:       send p.data to originator
7:    end if
8:    if succ(p).value ∈ (p.value, ub] then
9:       // if successor satisfies search criterion
10:      send Search(lb,ub,originator) to succ(p)
11:   else
12:      send SearchDoneMessage to originator
13:   end if
14: else
15:   find maximum k such that
      p.node[l][k].value ∈ (p.value, lb]
16:   send Search(lb,ub, originator) message
      to p.node[l][k].peer
17: end if
```

an entry whose value is between 5 (value stored in p_1) and 30 (the lower bound of the range query). In the current example, this corresponds to the second entry at the second level of p_1's P-tree, which points to node p_5 with value 29. 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.

The search procedure will go down one level of the P-tree every time a search message is forwarded to a different peer. This is similar to the behavior of B+-trees, and guarantees that we need at most $log_d\ N$ steps, so long as all entries are consistent. If a P-tree is inconsistent, however, the search cost may be more than $log_d\ N$ because Properties 1 through 4 in Section 3.3 may be violated.

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 3.3.2 (note that the fault-tolerant ring is still operational even in the presence of failures). In Section 5, we experimentally show that the search performance of P-trees does not degrade much even when the tree is temporarily inconsistent.

It is important to note that every search query cannot always be guaranteed to terminate in a P2P system. For example, a peer n 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 [25, 27, 24]. We can prove the following property about search during concurrent insertions and deletions.

Lemma 4: (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.

Algorithm 2 : p.Ping()

```
1: for l = 1; l < p.numLevels; 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 ≥ p.node[l].numEntries
11: end for
```

4.3 Peer Insertions

We now consider the case where a new peer wants to join the system. As in many P2P systems, we assume that a new peer p indicates its desire to join the system by contacting an existing peer. p issues a regular range 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. Second, the P-tree nodes of p need to be initialized. Finally, some of the P-trees of existing peers may need to be updated to take into consideration the addition of p.

In order to add a new peer to the lowest level ring, we rely on the ring-level stabilization protocol. In order to initialize the P-tree of a new peer p, we simply copy the P-tree nodes from pred(p) and replace the first entry in each node with an entry corresponding to p. Although the P-tree nodes copied from pred(p) are likely to be a close approximation of p's own P-tree nodes, clearly some entries could violate the *coverage* or *separation* properties for p, even though they were satisfied for pred(p). The insertion algorithm adheres to our policy of strictly separating responsibilities and leaves marking of entries as inconsistent to the Ping Process.

Ensuring that the P-tree nodes of existing peers become aware of the newly inserted node requires no special action. Eventually, the Ping Process in the existing nodes will detect any inconsistencies due to the newly inserted node (if any), and will invalidate the appropriate entries. The Stabilization Process at these nodes will then fix these inconsistencies.

4.4 Peer Deletions and Failures

In a P2P system, 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 (which then get repaired using the Stabilization Process).

4.5 The Ping Process

The Ping Process runs periodically at each peer; its pseudocode is shown in Algorithm 2. The Ping Process checks whether the entries are consistent with respect to the *coverage* and *separation* properties (line 7) in function Check-

Algorithm 3 : p.Stabilize()

```
1: l = 1
2: repeat
3:    root=p.StabilizeLevel(l)
4:    l++
5: until (root)
6: p.numLevels = l − 1
```

Algorithm 4 : p.StabilizeLevel(int l)

```
1: j = 1;
2: while j < p.node[l].numEntries do
3:    if p.node[l][j].state ≠ 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 j++
16: end while
17: while ¬ COVERS(p.node[l][j − 1], p.value)
    ∧ j < d do
18:   prevPeer = p.node[l][j − 1].peer
19:   newPeer =
      succ(prevPeer.node[l − 1][d − 1].peer)
20:   INSERT(p.node[l],j,newPeer)
21:   j++
22: end while
23: if COVERS(p.node[l][j − 1], p.value) then
24:   return true
25: else
26:   return false
27: end if
```

CovSep(). If any node entry is inconsistent with respect to either of the above two properties, its state is set to either `coverage` or `separation`. The Ping Process also checks to see whether a peer has been deleted/failed (line 4), and if so, it removes the corresponding entry from the P-tree node (line 5); function Remove() removes the j^{th} entry and decrements $p.node[l].numEntries$. Note that the Ping Process does not repair any inconsistencies — it merely detects them. Detected inconsistencies are repaired by the Stabilization Process.

4.6 The Stabilization Process

The Stabilization Process is the key to maintaining the consistency of P-trees. A separate Stabilization Process runs independently at each peer, and it repairs any inconsistencies detected by the Ping Process. The actual algorithm for the Stabilization Process is remarkably simple, nevertheless it guarantees that the P-tree structure eventually becomes fully consistent after any pattern of concurrent insertions and deletions.

Let us give first a high-level overview of the Stabilization Process. At each peer p, the Stabilization Process wakes up periodically and repairs the tree level by level, from bottom to top, within each level starting at entry 0; the successor-maintenance algorithm from the literature ensuring that the successor-pointer at the lowest level will be corrected [27]. 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.

Let us now consider the outer loop of the algorithm shown in Algorithm 3. The algorithm loops from the lowest level to the top-most level of the P-tree until the root level is reached (as indicated by the boolean variable *root*). Since the height of the P-tree data structure could actually change, we update the height ($p.numLevels$) at the end of the function.

Algorithm 4 describes the Stabilization Process within each level of the P-tree data structure at a node. The first loop from lines 2 to 16 repairs existing entries in the P-tree. For each entry $p.node[l][j]$, it checks whether $p.node[l][j]$ is consistent. If not, then either coverage or separation with respect to the previous entry *prevPeer* (line 4) is violated, and we need to repair $p.node[l][j]$. We repair $p.node[l][j]$ 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 3.2, 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 message to $prevPeer.node[l-1][d-1].peer$. (We can also reduce this overhead to one message by caching relevant entries). Note that we could set *newPeer* to any peer which satisfies the *coverage* and *separation*. After the adjustments in lines 7 or 9, the current entry is now consistent.

After repairing the current entry $p.node[l][j]$, 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 in line 11. Line 13 contains a sanity check: If the current entry already wraps around the tree, i.e., its subtree covers the value $p.value$, then this level is the root level, and we can stop at the current entry (line 14).

The loop in lines 17 to 22 makes sure that $p.node[l]$ has at least d entries (unless again its subtree covers `pred`(p) and thus this level is the root level — this is checked by the call to COVERS in line 17) by filling $p.node[l]$ up to d entries. Lines 23 to 27 returns whether this level is the root of the tree.

The following lemma states that the Stabilization Process eventually returns a P-tree to a fully consistent state after an arbitrary sequence of concurrent insertions and deletions.

Lemma 5: (Eventual Consistency) Given that 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 data structure is consistent.

Let us sketch the main argument behind the proof. The key intuition is that the stabilization process works bottom-up from the lowest level to the highest level (see Algorithm 3), and within each level, it operates on entries in left to right order (see Algorithm 4). Also, when the stabi-

lization process operates on entry j at level l, it only depends on entries that (a) are at levels $l-1$ or lower, or (b) are at level l but with entry positions $i < j$ (line 4 in Algorithm 4). Thus, we can prove by induction first on the level, and then on the entry position that the P-tree will eventually become consistent.

4.7 Example of Peer Failure/Deletion

We now illustrate the working of the P-tree algorithms described in Section 4 by using an example where a peer fails (or is voluntarily deleted - these two cases are handled in the same way in P-trees). Peer insertions are handled in an analogous manner, but we do not include an example of insertions due to space limitations.

In the case of peer failure, we show how the Ping Process and the Stabilization Process cooperate to fix the inconsistencies resulting from the failure. The algorithms described in Section 4 are designed to work asynchronously at each peer, and can handle concurrent insertions (and deletions). However, for ease of exposition in the examples, we assume that the Ping Process and the Stabilization Process run synchronously at the different peers.

Consider the initial consistent P-tree shown in Figure 4. Let us now assume that p_4 (with index value 23) fails, and thus has to be removed from the system. When the Ping Process is run at level 1 at each peer, it detects the entries that point to p_4 and deletes them from the corresponding node (lines 4-5 in Algorithm 2). In Figure 4, the entry $(23, p_4)$ is deleted from the level 1 nodes of peers p_1, p_2, and p_3. All entries are still marked `consistent` because the coverage and separation properties are satisfied (in the figures, an entry is depicted as consistent if there is no $*$ next to the entry, and an entry is not consistent if there is a $*$ next to the entry). The resulting P-tree is shown in Figure 7.

When the Stabilization Process is run at the level 1 node of each peer, all entries are marked `consistent`, and hence no action needs to be taken. Now assume that the Ping Process is run at the level 2 node of each peer. The Ping Process does not detect any inconsistencies in most peers, except for the peers p_8 and p_3. In p_8, the Ping Process removes the entry $(23, p_4)$ because p_4 no longer exists in the system (lines 4-5 in Algorithm 2). The Ping Process then checks to see whether coverage or separation is violated for the entry $(30, p_6)$, which is next to the deleted entry (line 7). Since both properties are satisfied, the entry is marked as `consistent`. In p_3, the Ping Process marks the entry $(29, p_5)$ as `separation` because the sub-trees rooted at the entries $(13, p_3)$ and $(29, p_5)$ overlap too much. This state of the P-tree is shown in Figure 8.

When the Stabilization Process runs at the level 2 node at each peer, all entries are `consistent` except for $(29, p_5)$ in p_3. Since the state of the entry $(29, p_5)$ is set to `separation`, it is replaced with a new entry that satisfies separation (line 9 in Algorithm 4). Since the replacement of the entry does not cause the next entry $(42, p_8)$ to become inconsistent (line 11), the Stabilization Process terminates. The final consistent P-tree without p_4 is shown in figure 9.

4.8 Implementation Issues

Since the Ping Process and the Stabilization Process test for coverage and separation frequently, we briefly discuss some optimizations to make these checks more efficient. The separation property is easy to check by sending a single mes-

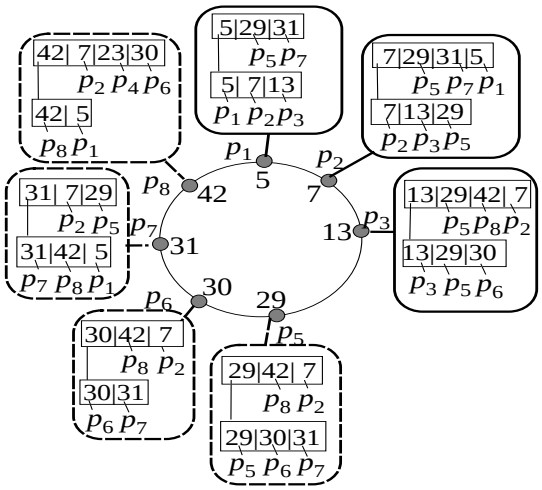

Figure 7: Peer Failure - Step 1

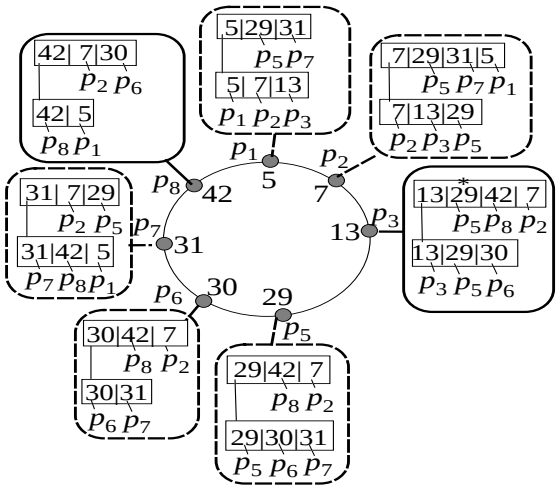

Figure 8: Peer Failure - Step 2

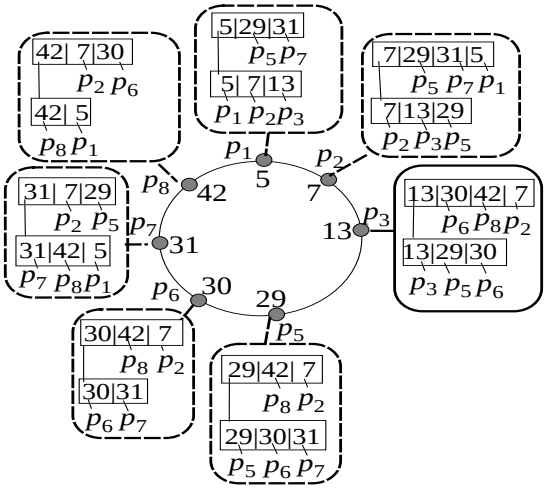

Figure 9: Peer Failure - Step 3

sage to $p' = p.node[l][j-1].peer$, and asking for $p'.node[l-1][d].value$. Checking the coverage property is more difficult, and requires one to compute the *reach* of a node (Section 3.3.2), which could require $O(log_d N)$ steps. To avoid this, we store an additional entry at the end of each node of a P-tree, called the *edge* entry. This entry is not used during search, but estimates the reach of the node, and can be efficiently maintained just like a regular entry.

5. EXPERIMENTAL EVALUATION

We now evaluate the performance of P-trees using both a simulation study and a real distributed implementation. In the simulation, our primary performance metric is the *message cost*, which is the total number of messages exchanged between peers for a given operation. A secondary metric is the *space requirement* for the index structure at each peer. In our preliminary experiments using a small distributed implementation, we use the elapsed time for an operation as the primary performance metric. For the simulation study, we assume that there is a single data item stored in each peer, since we focus on the number of messages exchanged between peers. In our real implementation, we handle multiple items per peer by mapping more than one "virtual peer" (Section 2.3) to the same physical peer.

5.1 Experimental Setup

We built a peer-to-peer simulator to evaluate the performance of P-tree over large-scale networks. The simulator was written in Java JDK 1.4, and all experiments were run on a cluster of workstations, each of which has 1GB of main memory and over 15GB of disk space.

In the simulator, each peer is associated with a 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 system will contact this peer. We simulate the functionality of the Ping Process by invalidating/deleting necessary entries before the Stabilization Process is called.

The parameters that we vary in our experiments are shown in Table 1. *NumPeers* is the number of peers in the system. *Order* is the order of the P-tree, and *FillFactor* is the fill factor of the P-tree, which is defined similar to the fill factor of a B+-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. *InsertionPattern* specifies the skew in the data values inserted into the system. If *InsertionPattern* is 0, values are inserted in descending order, and if it is 1, values are inserted uniformly at random. In general, if *InsertionPattern* is *ip*, it means that all insertions are localized within a fraction *ip* of the P2P network. In the systems we also have deletions, and we set the deletion pattern to be the same with the insertion pattern.

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 leaf nodes), 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.

Parameter	Range	Default
NumPeers	$1,000 - 250,000$	$100,000$
Order	$2 - 16$	4
FillFactor	$5 - 7$	$\lceil Order * 1.5 \rceil$
SPTimePeriod	$1 - 700$	25
IDRatio	$0.001 - 1000$	1
InsertionPattern	$0 - 1$	$1(random)$

Table 1: Parameters

5.2 Experimental Results

Varying Number of Peers Figure 10 shows the message cost for search and insertion/deletion operations, when the number of peers is varied. The right side of the y-axis contains the scale for search operations, and the left side contains the scale for insertion/deletion operations. The search message cost increases logarithmically with the number of peers (note the log scale on the x-axis). The logarithmic performance is to be expected because the height of the P-tree increases logarithmically with the number of peers. The figure also shows that the the message cost for insertions and deletions is a small fraction of the total number of peers in the system. This implies that the effects of insertions and deletions are highly localized, and do not spread to a large part of the P2P system. In particular, the message cost for insertions and deletions also increases roughly logarithmically with the number of peers.

Varying Order and Fill Factor Figure 11 shows the effect of varying the order of the P-tree and Figure 12 shows the effect of varying the fill factor of a node. In both cases, the search cost decreases with increasing order or increasing fill factor (as in B+-trees) 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. The cost increases because each peer has a larger number of entries (note that the number of entries per node 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 very high orders are not likely to be very practical. This is in contrast to B+-trees, where higher orders reduce both search and insertion/deletion cost.

Varying Stabilization Process Frequency Figure 13 shows the effect 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 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 14 shows the result of varying the ratio of insertion operations and deletion operations. We observe that 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

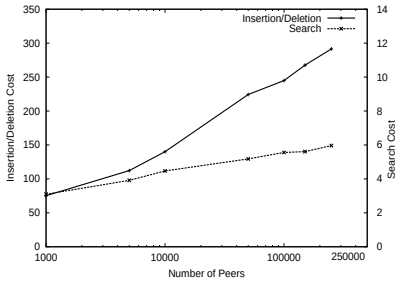

Figure 10: Number of peers plot

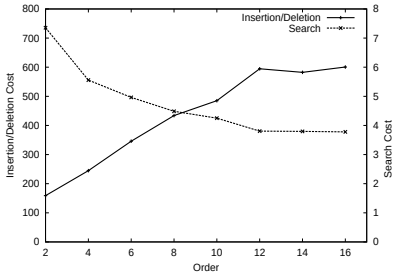

Figure 11: Order plot

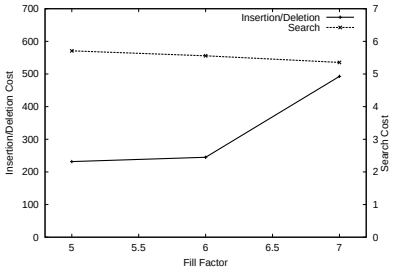

Figure 12: Fill factor plot

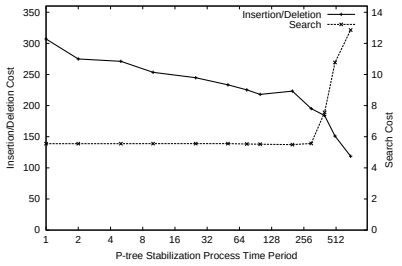

Figure 13: SP frequency plot

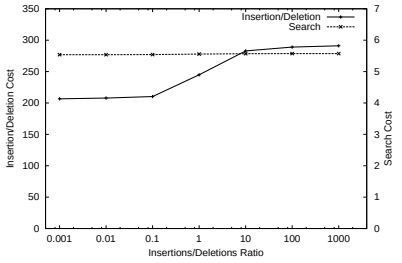

Figure 14: I/D ratio plot

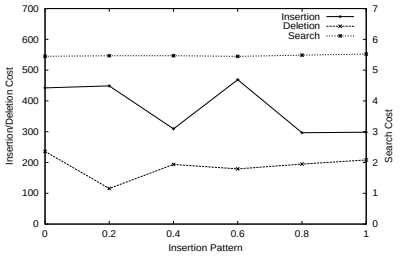

Figure 15: Insertion pattern plot

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.

Varying Insertion/Deletion Patterns Figure 15 shows the effect of varying the skew of the values inserted and deleted (recall that 0 corresponds to highly skewed distribution, while 1 corresponds to a uniform distribution). It is worth noting that even in a highly skewed distribution, the performance of P-trees degrades by less than a factor of two. This behavior is attributable to the fact that the P-tree dynamically balances its sub-trees, thereby spreading out any hotspots.

Space Requirements Our experiments showed that the number of entries stored at each peer is at most $2d \cdot log_d N$, where d is the order of the P-tree and N is the number of data items in the system.

5.3 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 Pentium IV PCs connected via a LAN; each PC had 1GB of RAM and 40GB of disk space. We varied the number of "virtual peers" from 20 to 30. We mapped 5 virtual peers to each of the 4-6 physical peers. Each virtual peer had a unique data value between 1 and 30. The virtual peers communicated using remote-procedure calls (RPCs), although the RPCs between two virtual peers mapped to the same physical peer was implemented as a local procedure call by the C# implementation. 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.

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

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 virtual peer failures varies from 13-19s. The search and stabilize times are of the order of seconds because we run the Ping and Stabilization process only once per second.

6. RELATED WORK

Chord [27], Pastry [25], Tapestry [31] and CAN [24] implement distributed hash tables to provide efficient lookup of a given key value. Since a hash function destroys the ordering in the key value space, these structures cannot process range queries efficiently. Approaches to the lookup problem based on prefix matching/tries [8, 2, 25] cannot be used to solve the range query problem for arbitrary numerical attributes such as floating point numbers. Other approaches to the lookup problem include [9, 30, 22]. Techniques for efficient keyword search are presented in [29, 6, 23]. However, none of these systems support range queries.

There has been recent work on P2P data management issues like schema mediation [4, 26, 15], query processing [28], and the evaluation of complex queries such as joins [12, 26].

However, none of these approaches address the issue of supporting range queries efficiently.

There has been some work on developing distributed index structures [19, 17, 13, 20]. The work on distributed B+-trees [17] is perhaps the closest to our work. However, most of these techniques maintain consistency among the distributed replicas by using a *primary copy*, which creates both scalability and availability problems when dealing with thousands of peers. In contrast, the P-tree data structure is designed to be resilient to extended failures of arbitrary peers (see also Section 3.2.1). The dPi-tree [21] and DRT [18] maintain replicas lazily. However, these schemes are not designed for peers that leave the system, which makes it inadequate in a P2P environment.

Gupta et al [11] present a technique for computing range queries in P2P systems using order-preserving hash functions. However, since the hash function scrambles the ordering in the value space, their system can only provide approximate answers to range queries (as opposed to the exact answers provided by P-trees). Aspnes et al. propose *Skip graphs* [3], a randomized structure based on skip lists, which supports range queries. However, unlike P-trees, they only provide probabilistic guarantees even when the index is fully consistent. Finally, Daskos et al. [1] present another scheme for answering range queries, but the performance of their system depends on certain heuristics for insertions. Their proposed heuristics do not offer any performance guarantees and thus, unlike P-trees, their search performance can be linear in the worst case even after their index becomes fully consistent.

7. CONCLUSION

We have proposed P-trees as a distributed and fault-tolerant index structure for P2P networks. The P-tree is well suited for applications such as resource disovery for web services and the grid because it extends the capability of existing P2P systems by supporting range queries in addition to equality queries. Results from our simulation study and real implementation show that P-trees support the basic operations of search, insertion and deletion efficiently with the average cost per operation being approximately logarithmic in the number of peers in the system. We believe that P-trees are just the first step towards building a fully functional P2P database system. There are many interesting open issues, including efficient support for multiple tuples per peer, complex queries (such as joins), approximate query results, data replication and exploiting physical proximity information.

8. REFERENCES

[1] S. G. A Daskos and X. An. Peper: A distributed range addressing space for p2p systems. In *Workshop on P2P Computing*, 2003.

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

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

[4] P. Bernstein and et al. Data management for peer-to-peer computing: A vision. In *WebDB*, 2002.

[5] D. Comer. The ubiquitous b-tree. In *Computing Surveys, 11(2)*, pages 121–137, 1979.

[6] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer networks. In *ICDCS*, 2002.

[7] A. H. et al. Schema mediation in peer data management systems. In *ICDE*, 2003.

[8] M. J. Freedman and R. Vingralek. Efficient peer-to-peer lookup based on a distributed trie. In *IPTPS*, 2002.

[9] P. Ganesan, Q. Sun, and H. Garcia-Molina. Yappers: A peer-to-peer lookup service over arbitrary topology. In *INFOCOM*, 2003.

[10] S. Gribble and et al. What can databases do for peer-to-peer? In *WebDB*, 2001.

[11] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *CIDR*, 2003.

[12] M. Harren and et al. Complex queries in dht-based peer-to-peer networks. In *IPTPS*, 2002.

[13] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the b-link tree. In *IPPS*, 1992.

[14] T. Johnson and P. Krishna. Lazy updates for distributed search structure. In *SIGMOD*, 1993.

[15] V. Josifovski, P. Schwarz, L. Haas, and E. Lin. Garlic: a new flavor of federated query processing for db2. In *SIGMOD*, pages 524–532. ACM Press, 2002.

[16] D. Kossmann. The state of the art in distributed query processing. In *ACM Computing Surveys*, Sep 2000.

[17] P. A. Krishna and T. Johnson. Index replication in a distributed b-tree. In *COMAD*, 1994.

[18] B. Krll and P. Widmayer. Distributing a search tree among a growing number of processors. In *SIGMOD*, 1994.

[19] W. Litwin, M.-A. Neimat, and D. A. Schneider. Lh* - linear hashing for distributed files. In *SIGMOD*, 1993.

[20] W. Litwin, M.-A. Neimat, and D. A. Schneider. Rp*: A family of order preserving scalable distributed data structures. In *VLDB*, 1994.

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

[22] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *PODC*, 2002.

[23] Neurogrid - http://www.neurogrid.net.

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

[25] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[26] W. Siong Ng and et al. Peerdb: A p2p-based system for distributed data sharing. In *ICDE*, 2003.

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

[28] D. M. Vassilis Papadimos and K. Tufte. Distributed query processing and catalogs for peer-to-peer systems. In *CIDR*, 2003.

[29] B. Yang and H. Garcia-Molina. Efficient search in peer-to-peer networks. In *ICDCS*, 2002.

[30] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *ICDE*, 2003.

[31] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: an infrastructure for fault-taulerant wide-area location and routing. In *Technical Report, U.C.Berkeley*, 2001.