P-Tree: A P2P Index for Resource Discovery Applications

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 for resource discovery applications called the **P-tree**. P-trees can efficiently support *range queries* in addition to equality queries.

1. INTRODUCTION

Peer-to-peer (P2P) systems are emerging as a new paradigm for structuring large-scale distributed systems. The key advantages of P2P systems are their scalability, due to resource-sharing among cooperating peers, their fault-tolerance, due to the symmetrical nature of peers, and their robustness, due to self-organization in the face of peer and network failures. Due to the above advantages, P2P systems have made inroads for content distribution and service discovery applications [10, 8, 9, 1]. However, most existing systems only support location of services based on their name (i.e., they only support simple equality lookups).

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 offer services described by semantically-rich XML documents. Users can then *query* this "P2P service directory" as if all the services were registered 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* attribute that specifies the IP address of the grid node, an *OSType* attribute indicating the operating system (such as Linux, MAC-OS, etc.), and a *MainMemory* attribute indicating how much main memory is available at the 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 at least 4GB of main memory:

```
for $node in //node
where $node/@OSType = 'Linux' and
      $node/@MainMemory >= 4096
return $node/@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 grid nodes 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 because it supports both equality *and* range queries. In a stable system (no insertions or deletions), a P-tree of order d provides $O(log_d N)$ search cost for equality queries and $O(m+log_d N)$ 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 show that P-trees can handle frequent item/peer insertions and deletions with low maintenance overhead and small impact on search performance.

For the rest of this paper, we use the following terminology and make the following assumptions. We target applications that offer a single 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 [5] 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, however, 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 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.

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

WWW2004, May 17–20, 2004, New York, New York, USA.
.

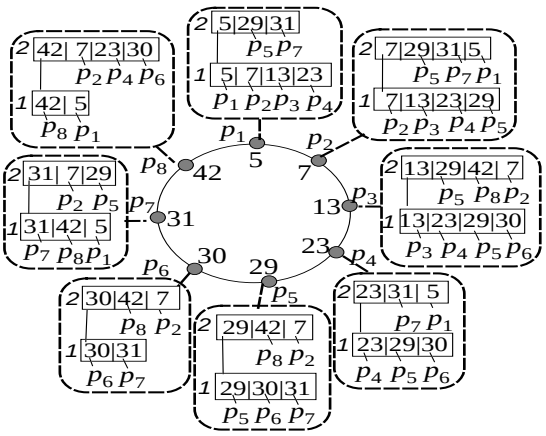

Figure 1: Full P-tree

value as being the smallest value in the ring (note that on a ring, any value can be viewed as the smallest value). 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 the remaining (non root-to-leaf) parts of its tree.

As an illustration, consider Figure 1. The peer p_1, which stores the item with value 5, only stores the root-to-leaf path of its 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 nodes in the peers p_5 and p_7 (which store the data items corresponding to 29 and 31, respectively). Note that p_5 and p_7 also store the root-to-leaf paths of their independent B+-trees. Consequently, 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. A full formalization of the P-tree properties, such as the extent of the allowed overlap among sub-trees, can be found in [4].

The above structure of P-trees has the following advantages. First, since the P-tree maintains the B+-tree-like hierarchical structure, it can provide $O(log_d N)$ search performance for equality queries in a consistent state. Second, since the order of the ring corresponds to the order of 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 solely 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 leftmost root-to-leaf path, and each node has at most $2d$ entries, the total storage requirement per peer is $O(d \cdot log_d N)$.

2.2 P-tree: Algorithms

This section presents a high-level description of the P-tree algorithms, which are fully decentralized. For full details and experimental results, we refer the reader to [4].

The main idea is to allows the P-tree to be in a state of *local inconsistency*, where some of the consistency requirements for semi-independent B+-trees are not satisfied (such as the extent of overlap between successive tree branches). Local inconsistency allows searches to proceed correctly, with perhaps a slight degradation in performance, even if peers are continually being inserted and deleted from the system.

The local inconsistency of the P-tree is repaired by two co-operating periodic processes, the *Ping Process* and the *Stabilization Process* that run at each peer. These processes transform an inconsistent P-tree to a consistent one without any need for global coordination. The Ping Process detects changes due to insertions, deletions and peer failures, and marks the appropriate index entries as inconsistent. The Stabilization Process repairs these inconsistent entries so the P-tree properties are again satisfied. 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.

3. RELATED WORK

Existing systems [10, 6] that use hash functions and implement distributed schemes for efficient lookup cannot efficiently process range queries to provide exact answers due to order scrambling in the value space. Distributed database index structures (e.g. [7]) are inadequate in a P2P framework as they do not allow peers to leave the system at will. Skip graphs [2] and Peper [5] support range queries but one provides only probabilistic guarantees while the other provides no search guarantees, even with a consistent index.

4. CONCLUSION

We have proposed the P-tree index, which is 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 cost per operation being approximately logarithmic in the number of peers.

5. 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. Crainiceanu et al. Querying peer-to-peer networks using p-trees. In *Cornell University TR 2004-1926*, 2004.

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

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

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

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

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

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