Brief Announcement: Hashed Predecessor Patricia Trie - A Data Structure for Efficient Predecessor Queries in Peer-to-Peer Systems

Sebastian Kniesburges and Christian Scheideler

Department of Computer Science
University of Paderborn
D-33102 Paderborn
Germany
seppel@upb.de, scheideler@upb.de

Abstract. The design of efficient search structures for peer-to-peer systems has attracted a lot of attention in recent years. In this announcement we address the problem of finding the predecessor in a key set and present an efficient data structure called hashed Predecessor Patricia trie. Our hashed Predecessor Patricia trie supports \(\text{PredecessorSearch}(x)\) and \(\text{Insert}(x)\) and \(\text{Delete}(x)\) in \(O(\log \log u)\) hash table accesses when \(u\) is the size of the universe of the keys. That is the costs only depend on \(u\) and not the size of the data structure. One feature of our approach is that it only uses the lookup interface of the hash table and therefore hash table accesses may be realized by any distributed hash table (DHT).

1 Introduction

In this brief announcement we consider the predecessor problem in peer-to-peer systems. We present a data structure that efficiently supports the predecessor problem with the help of any common DHT, e.g. Chord or Pastry. We define the predecessor in the following way: Given a key set \(S\) with a total order and a search key \(x\), find \(\max\{y \in S \mid y \leq x\}\). We interpret \(z \leq x\) as \(z\) is lexicographically smaller than \(x\). In the following we only consider binary strings. The predecessor problem has many applications ranging from string matching problems, IP lookup in Internet routers and computational geometry to range queries in distributed file-sharing applications. Our data structure supports the following operations: \(\text{Insert}(x)\): this adds the key \(x\) to the set \(S\). If \(x\) already exists in \(S\), it will not be inserted a second time. \(\text{Delete}(x)\): this removes the key from the set \(S\), i.e., \(S := S - \{x\}\). \(\text{PredecessorSearch}(x)\): this returns a key \(y \in S\) that is the predecessor of \(x\). Related data structures include trie hashing [2] and the popular x- and y-fast tries [3] other related work is mentioned in our
previous paper [1]. We think that our solution using only a single Patricia trie is intuitive and simple to understand. Furthermore it is applicable to any hash table and thus also DHTs.

2 Our Results

The hashed Predecessor Patricia trie is based on the hashed Patricia we introduced in [1]. This is constructed by adding some additional nodes to the Patricia trie to allow a binary search on the prefix lengths. For details of this construction see [1]. In our extended approach, the hashed Predecessor Patricia trie, we assume that all inserted keys have the same length $\log u$ and modify the hashed Patricia trie by adding some further pointers to enable an efficient predecessor search. All leaves form a sorted doubly-linked list. Differing from the hashed Patricia trie we store for each node $v$ a pointer to the largest key $l_{\text{max}}(v)$ in its left subtrie instead of an arbitrary key in its subtries. To ensure efficient updates all the pointers are undirected, i.e. each leaf stores the start nodes of the pointers pointing to it. The basic idea to find the predecessor for a search key $x$ is to use two consecutive binary searches according to [1]. The first binary search finds the node $u$ such that $u$’s identifier is the largest prefix of $x$ among all node identifiers. The second binary search then looks for the ancestor $w$ of $u$ such that $l_{\text{max}}(w)$ is the predecessor of $x$. Each binary search needs $\mathcal{O}(\log \log u)$ hashtable lookup (HT-Lookup) operations. By this construction it follows that each inner node stores at most $\mathcal{O}(1)$ pointers, and at most $\mathcal{O}(1)$ pointers point to the same leaf. Then the following theorem holds.

**Theorem 1.** An execution of $\text{PredecessorSearch}(x)$ needs $\mathcal{O}(\log \log u)$ hashtable lookup (HT-Lookup) operations, an execution of $\text{Insert}(x)$ needs $\mathcal{O}(\log \log u)$ HT-Lookup and $\mathcal{O}(1)$ HT-Write operations and an execution of and $\text{Delete}(x)$ needs $\mathcal{O}(1)$ HT-Lookup and $\mathcal{O}(1)$ HT-Write operations. The hashed Predecessor Patricia trie needs $\Theta(\sum_{k \in S} \log u)$ memory space, where $\sum_{k \in S} \log u$ is the sum of the bit lengths of the stored keys.

References