Department of Informatics, University of Zürich

MSc Basismodul

The Adaptive Radix Tree

Rafael Kallis

Matrikelnummer: 14-708-887

Email: rk@rafaelkallis.com

September 1, 2018

supervised by Prof. Dr. Michael Böhlen and Kevin Wellenzohn

University of Zurich[UZH]

Department of Informatics

1 Introduction

Main-Memory Databases increasingly become a viable option for many applications. Whilst main memory is a considerably faster medium than a secondary disk, utilizing caches more efficiently would lead to even better access times.

Leis et al. [2] propose the Adaptive Radix Tree (ART), an in-memory data structure which efficiently stores and retrieves data, even outperforming red-black trees. As we will see later, ART achieves its exceptional performance, and space efficiency, by compressing the tree both vertically and horizontally. Higher space effiency allows ART to utilize caches more optimal.

The goal of this project is to implement ART, as proposed by [2] in C++ and compare its transactional throughput against other in-memory data structures, e.g. red-black trees and hash tables. In Section 2 we describe how ART is constructed by applying vertical and horizontal compression to a trie. Next, we describe the point query procedure, as well as value insertion and removal in Section 3. Finally, a benchmark of ART, a red-black tree and a hashtable is presented in **??**.

2 Adaptive Radix Tree (ART)

The *Adaptive Radix Tree* (Tree) is a space efficient trie, which achieves its low memory consumption using *vertical* and *horizontal* compression. Using vertical compression, we reduce the tree height significantly by merging parent nodes with child nodes under certain circumstances. Horizontal compression reduces the amount of space required by each node depending on the number of child nodes. We will talk about both concepts in-depth in Sections 2.2 and 2.3.

2.1 Trie

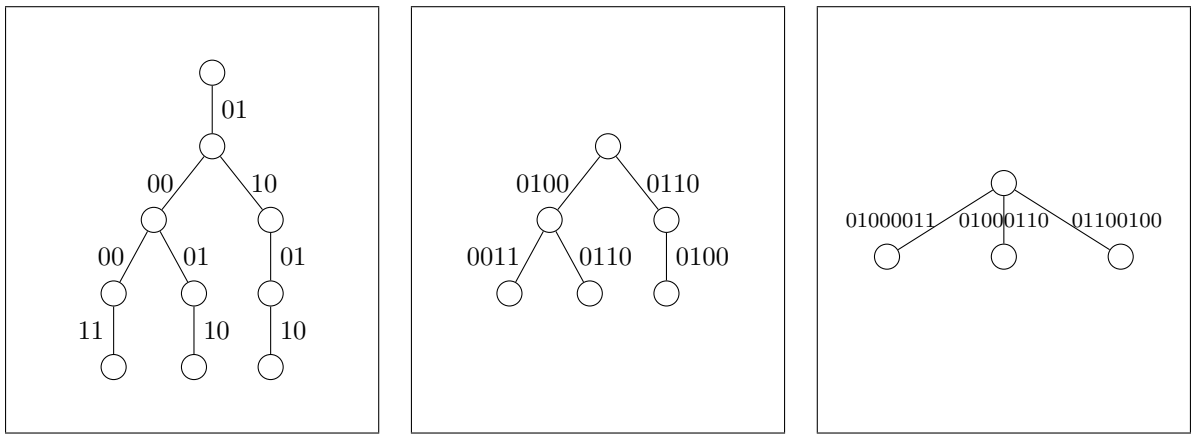
The trie [1] is a hierarchical data structure which stores key-value pairs. The trie can answer both point and range queries efficiently since keys are stored sorted according to their lexicographic order. A node's path represents the node's key. This is done by splitting a key into chunks of s bits, where s is called *span*.

Each inner node has 2^s child nodes, one for each possible s-bit sequence. During tree traversal, we propagate down to the child node identified by the d-th s-bit chunk of the key, where d is the depth of the current node. Using an array of 2^s pointers, this lookup can be done without any additional comparison.

Figure 1 depicts tries storing the 8-bit keys "01000011", "01000110" and "01100100" with $s = 2, 4, 8$. Span s is critical for the performance of the trie because s determines the height of the trie. We observe that by increasing the span, we decrease the tree height. A trie storing k bit keys has $\lceil \frac{k}{s} \rceil$ levels of inner nodes. As a consequence, point queries, insertions and deletions have $O(k)$ complexity. Having a data structure with time complexity not dependend on n, makes it very attractive for large data sets.

Span s also determines the space consumption of the tree. A node with span s requires 2^s pointers. The tries in Figure 1 require a total of $240, 224, 384$ and 2048 bytes respectively, for the bookeeping of child nodes, assuming 64-bit pointers. An apparent trade off exists between tree height versus space efficiency that depends on s.

When storing key-value pairs, we might insert a tuple whose key is a prefix of an existing key. In order to support this operation, each node is given a pointer called "value" which defaults to `null`. With this addition, inner nodes can now hold values, i.e. key-value pairs can be added where the key is a prefix of an existing key.



$s = 2$ $s = 4$ $s = 8$

Figure 1: Tries with span $s = 2, 4, 8$ storing keys "01000011", "01000110" and "01100100".

2.2 Vertical (Prefix) Compression

When storing long keys, chains start to form where each node only has a single child. As a consequence, we waste a lot of space on structural information. Morrison introduced *Patricia* [3]. Patricia is a space-optimized trie in which each node with no siblings is merged with its parent, i.e. inner nodes are only created if they are required to distinguish at least two leaf nodes. Doing so, we eliminate chains caused by long keys which make tries space-inefficient. Although Morrison's Patricia tree is a bitwise trie, i.e. has a span $s = 1$, the principle can be applied to tries with any span. We refer to this heuristic as *vertical compression*.

Vertical compression is implemented by storing an additional variable, called *prefix*, inside each node. This variable stores the concatenation of partial keys of descendants that were eliminated because they had no siblings. Figure 2 depicts two tries, one with and one without vertical compression. We observe that nodes with no siblings, color coded red, are eliminated and their partial key is appended to the parent's prefix. With even longer keys, the results of vertical compression are even more astonishing.

Tall trees may become very compact, and so not only do we save space that was otherwise wasted on structural inforation, we also are now able to traverse the data structure even faster since the height decreased.

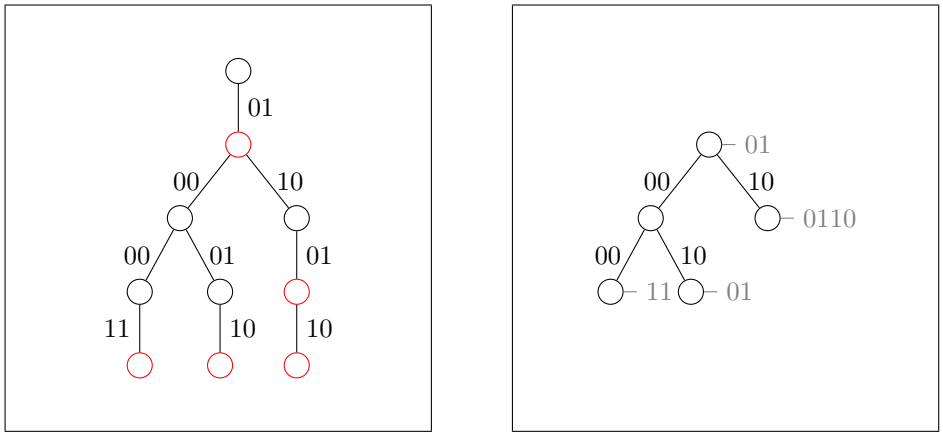


Figure 2: Tries with span $s = 2$ storing keys "01000011", "01000110" and "01100100". The trie on the right incorporates vertical compression. Red nodes indicate nodes which get eliminated under vertical compression. Gray strings represent the value of the "prefix" property.

2.3 Horizontal Compression (Adaptive Nodes)

With large values of span s, we sacrifice an excessive amount of space for a smaller tree height. Space is allocated for pointers which keep references of child nodes.

In order to reduce the space needed to keep such references, Leis et al. propose *Adaptive Nodes* [2], which make use of dynamic data structures instead of static arrays for child node bookkeeping. Doing so, we allocate a minimal amount of space when the number of children is small and add more space if required, i.e. more children are added. We refer to this heuristic as *horizontal compression*. Leis et al. also fix the span $s = 8$, i.e. partial keys are 1 byte long and therefore each node can have up to $2^8 = 256$ children.

When applying horizontal compression, a node is in one of four configurations, depending on the number of child nodes. Each of the four configurations is optimized for a different amount of children. The most compact configuration is called *Node4* which can carry up to four children. In the same manner, we also have *Node16*, *Node48* and *Node256*.

We now describe the structure of each of the four configurations. The node types are also illustrated in Figure 3. We store the partial keys 65, 82, 84 and their corresponding child nodes α, β, γ in an instance of each node type. Note that $\varnothing$ represents a `null` pointer.

A node of type *Node4* contains two 4-element arrays, one called "partial keys" and one called "children". The "partial keys" array, holds partial keys which identify children of that node. The "children" array, holds pointers to the child nodes. Partial keys and pointers are stored at corresponding positions and the partial keys are sorted. *Node4* cannot be used with more than 4 child nodes.

A node of type *Node16* is structured similarly to *Node4*, the only difference being the lengths of the two static arrays, which are 16 each. *Node16* shall be used when the number of child nodes is at least 5 and not greater than 16.

An instance of *Node48* contains a 256-element array named "indexes" and a 48-element array called "children". Partial keys are stored implicitly in "indexes", i.e. can be indexed with partial key bytes directly. As the name suggests, "indexes" stores the index of a child node inside the "children" array. This node can be compared to virtual memory, since the address space (256 addresses) is wider than the actual available storage space (48 slots). *Node48* shall be used when a node has at least 17 child nodes and not more than 48. In Figure 3, some entries in the "indexes" array contain the value 48, which is used to indicate that no child node is identified by the corresponding partial key.

Finally, a node of type *Node256* contains an array of 256 pointers which can be indexed with partial key bytes directly. Child nodes can be found with a single lookup. A node shall be an instance of *Node256* if it has at least 48 child nodes.

In addition to the state required to bookkeep child nodes, the nodes also have to store the node type.

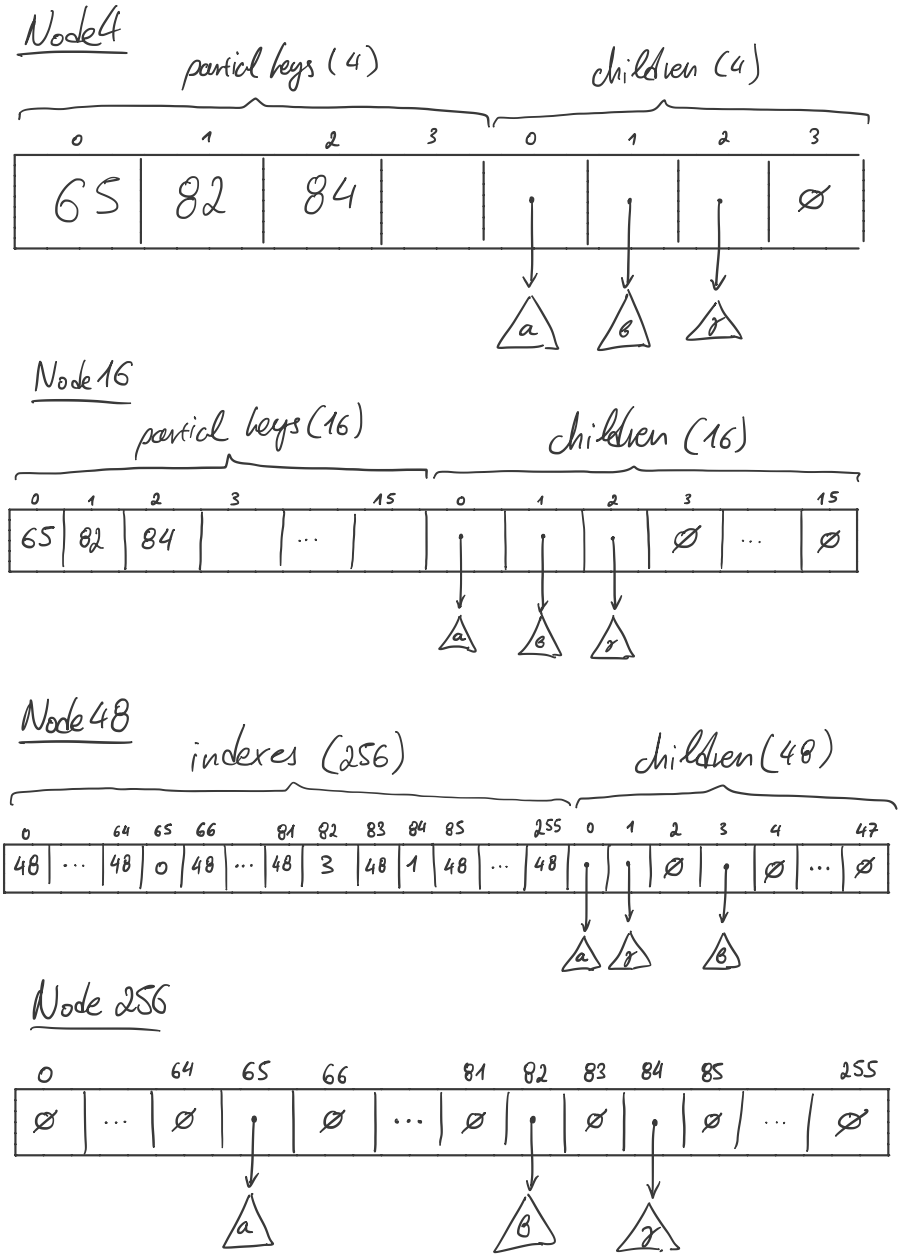


Figure 3: When horizontal compression is applied, a node is in one of four configurations, namely *Node4*, *Node16*, *Node48* and *Node256*. Each of the four configurations is optimized for a different number of child nodes. We store the partial keys 65, 82, 84 and their corresponding child nodes α, β, γ in an instance of each node type. Note that $\varnothing$ represents a `null` pointer.

3 Algorithms

We will now describe how four fundamental operations, namely point query, insert and delete are implemented in ART. The algorithms mentioned below are based on the implementations presented by Leis et al. [2]. However, their implementation does not allow for two key-value pairs to be stored simultaneously where one key is a prefix of the other. We therefore adjusted the algorithms accordingly. In addition to prefix key insertion support, we also use iteration instead of recursion in order to avoid expensive context switches. We present our implementations below.

3.1 Point Query

The code fragment shows the implementation of a point query on ART in C++. Method `get` accepts a key as an argument and returns a pointer the the value associated with the given key, or `null` otherwise.

In lines 7-8 we declare and initialize a pointer, `cur`, which references the current node during tree traversal, and an integer, `depth` which holds the depth of the current node.

We now enter a loop in which we check if `cur` references `null` during the beginning of each iteration, and if so, we return `null`.

In lines 13-19 we check if a prefix mismatch occurs. This step is required because of vertical compression (c.f. Section 2.2). Method `check_prefix` is a member of the `node` class which determines the number of matching bytes between `cur`'s prefix and `key` w.r.t. the current depth. If a byte mismatch is discovered, `null` is returned.

Lines 20-24 check for an exact match of the key at the current node. If so, we return the value of the current node.

Finally, we traverse to the next child node. `depth` is increased to account for the nodes merged due to vertical compression. We lookup the next child node, which is assigned to `cur`.

```
template <class T> class art {
public:
  /**
   * Finds the value associated with the given key.
   */
  T *get(const key_type &key) const {
    node<T> *cur = root_;
    int depth = 0;
    while (true) {
      if (cur == nullptr) {
        return nullptr;
      }
      const int prefix_len =
        cur->get_prefix().length();
      const bool is_prefix_match =
        cur->check_prefix(key, depth) == prefix_len;
      if (!is_prefix_match) {
        return nullptr;
      }
      const bool is_exact_match =
        prefix_len == key.length() - depth;
      if (is_exact_match) {
        return cur->get_value();
      }
      depth += prefix_len;
      node<T> **next = cur->find_child(key[depth]);
      cur = next != nullptr ? *next : nullptr;
      ++depth;
    }
  }

  /* ... */
private:
  node<T> *root_ = nullptr;
}
```

3.2 Insertion

3.3 Deletion

References

[1] E. Fredkin. Trie memory. Communications of the ACM, 3(9):490–499, 1960.

[2] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In 2013 IEEE 29th International Conference on Data Engineering (ICDE), pages 38–49. IEEE, 2013.

[3] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM (JACM), 15(4):514–534, 1968.