EXAMINATION OF CACHE SENSITIVE B+-TREES FOR STRING INDEXING

C. A. CSAR

ABSTRACT. There are a variety of data structures designed for use as indexes in main memory that are suited for a variety of different tasks. Past work has demonstrated the importance of accounting for the existence of the processor cache in designing and implementing such data structures. Cache sensitivity refers to making good use of the space in the processor cache and minimizing the number of misses any operation experiences. By way of example the cache line on the Zoo machines is 64 bytes, which is capable of holding only 4 integers keys if there is a pointer to each child. The Cache Sensitive B+-Tree (CSB+-Tree) was one of those developed for this task. Although there exist variants of B+-Trees specialized for indexing strings, there do not seem to have been attempts to make them cache sensitive. The purpose of this project was to provide a second implementation of Cache Sensitive B+-Trees that would be better suited to general use than the currently extant one, and to examine extending its functionality to include the use of strings as keys, without sacrificing cache sensitivity and maintaining performance, and while preserving a joint code base with fixed length keys, such as integers. It is expected that performance will be maintained and that benchmarks will demonstrate the competitive speed of modified Cache Sensitive B+-Trees for use with variable length strings as keys in an index. Of the several variants of Cache Sensitive B+-Trees implemented, the full Cache Sensitive B+-Tree was chosen on account of its demonstrated superior performance due to allocating all of an internal node’s children in a single block.

1 INTRODUCTION

There now exist computers with rather large main memories which make it possible to have indexes of significant size contained entirely within memory. Combined with the increase in memory size has been
a growing gap in processor speed in comparison with cache miss cost. On account of this previous work has examined the cache sensitivity of some data structures and worked on improving it [6, 7].

2 Design

This project’s original intent was to develop a main memory index system, and as part of that evaluate various data structure to determine which would lead to the fastest index system. For this a Cache Sensitive B+-Tree was chosen as the first candidate data structure to evaluate; however, at that time it was discovered that there was only the one implementation of a CSB+-Tree available, which was produced as part of [7], and this implementation was not suited to the task as it used only 32 bit integers for keys and payloads. Given the desire to also use strings as keys, and the unpleasantness of extending code implemented primarily with gotos, which admittedly may be for speed, it was decided to reimplement a CSB+-Tree in a form that would be better suited for current and future use. The original proposal for the index system specified the capability of using 32 bit integers, 64 bit integers, and strings as keys; consequently the current implementation was designed to use any of these three as keys with minimal automatic reconfiguration that would allow all three variants to be used in the code at the same time. The layout of a CSB+-Tree is documented mainly in [7] and is an extension of [2], but there are a few key points. The main cache sensitivity of the CSB+-Tree comes from eliminating pointers in the internal nodes of the tree in order to make better use of the space used by the node, which in [7] is one cache line (64 bytes...
on the Zoo), but as shown in [5] the optimal size is substantially larger than this on account of greater fanout decreasing the depth of the tree. To reduce the number of pointers in a tree node only one pointer is used for all of the child nodes so that the position of any given child node may be directly calculated from that pointer, this requires that all of the children of any node to be allocated in a block and kept together. In order to reduce copying costs a full CSB+-Tree allocates all children, even the empty ones, at the beginning rather than reallocating as necessary; however this has the disadvantage that memory is used even if the children are never created, but since B+-Trees have the property that all nodes should be at least fifty percent full this simply increases memory usage by approximately a factor of two. The small sizes of the blocks used for the nodes leads to an interesting question when using strings as keys, namely that the strings were assumed to be of variable length from around ten characters to one hundred and twenty-eight characters, which means that the expected string length (assuming a random distribution) will be about the size of a cache line. Consequently any individual string could be a significant portion of the preferred size of a node and this would lead to reduced fanout and also to slower search within a node. To avoid this, and to take advantage of the code base for integers, a hash based solution was examined. Since one of the properties of B+ trees is that the entries are stored in sorted order to enable sequential scans, an order preserving hash function was needed. A similar method is used for String B-Trees [3] which use a of each string prefix. The order preserving function used is relatively
simple; since strings have a maximum length, the strings may be enumerated and then all possible strings may be divided into $2^{64}$ buckets with each bucket getting one integer. The function used should work equally well for restricted alphabets as if $N$ is the number of characters in the alphabet, $S$ is the string of length $y$, and $S_1 \ldots S_y$ are the individual characters, and $x$ is the maximum length of the string, then for mapping to a 64 bit integer the following is used.

$$\frac{\sum_{i=1}^{y} (S_i \times N^{x-i}) \times 2^{64}}{N^x}$$

It seems fairly likely that this function is roughly equivalent to the prefix of strings, but there are some differences, in the prefix method some space is wasted as with null terminated strings as found in C the number of characters in the alphabet is only 255, but the space for all 256 is used.

### 3 Implementation

As it was desired to have the various options for key and payload type to share the same code base and to be usable together it was necessary to change the names of all functions and other structures in the name space automatically. To do this everything whose name must change is referenced with a macro that adds the mnemonic for the particular type of key and payload combination. This system also enables additional key and payload types to be added without significant difficulty provided that they share similar properties to those already implemented. Strings and indeed any other sort of structure represent
a special case because it is generally impractical to store copies of them for use as keys due to the storage or comparison requirements. Instead they are to be encoded as a fixed width integer as described above for strings. This integer representation will be used in the internal nodes of the tree to determine which leaf is correct for either insertion or searching. Once the leaf correct leaf node is found the proper insertion location among those entries with the same integer value may be found by using a sequential search. This use of a sequential search for lookup is potentially slow and undesirable, but if a random distribution of keys is assumed then the number of collisions should be relatively small, as with 30 million entries which on average should consume approximately 2 GB of memory for the keys alone and with this many keys only around one trillionth of the hash space. Because the keys need to be kept sorted there exists a relatively bad case that can occur for insertion. In the event that there is a collision an entry might need to be inserted into a leaf different from the one that the tree traversal resulted in, ordinarily this will not be a problem, but in the event that the correct leaf is full and needs to split there is no way in the current implementation to split this later node. To solve this it is necessary to move nodes into the leaf node our traversal found in order to make space in the correct leaf. This process may be somewhat expensive as it will involve copying a fair amount of data, but this is a rare case of a rare case. Integers do not have this problem even among entries with the same key and different payloads because the leaf that the traversal reaches is guaranteed to be correct as even if there are some number of
leaves all full of the same key, any one of them will do because there are no guarantees about the relative order of entries with the same keys.

4 Related Work

The Trie [4] and variants thereof are the primary structures for indexing strings and it is against these that this extension of the CSB+-Tree will be compared. It is expected that this comparison should be favorable due to the generally poor cache sensitivity of Tries. There has been some recent work [1] in making Tries cache conscious and a variant known as a HAT-Trie that [1] claims to be the fastest for maintaining strings in sorted order in main memory. Future work will be required to compare these data structures under a variety of loads. Regrettably it will not be possible to evaluate the claims of [1] and examine whether a CSB+-Tree modified for use with strings is faster due to the lack of an available implementation to run benchmarks against.

References