Just another bit faster

Speed comparison of T-Tree and B-Tree in a database alike benchmark.

A Theoretical and Practical consideration

Sascha Kuhl,
London, Marburg

Abstract

We observe the real time saving between a B-Tree, which is used in many common DBMS implementations and a T-Tree, which is up to the moment almost not used. We find the T-Tree a bit faster and thus less energy consuming. We argue the finding holds for other circumstances than the ones observed.

Introduction

Database systems were invented to have an organised and common way to store and access data, up to large amounts. Having evolved from simple file stores, they come along in various forms, graph databases, relational databases, object oriented ones or key value stores, just to name the most common approaches.

More than the other types, relational databases rely on indexes in forms of trees and tries to not keep stuck with poor "heap" or list alike performance on data access. Most of the commercial packages, such as SQL Server or Oracle implement a B or B+-Tree as a mandatory clustered primary key for each table having such a key.

However, when looking up a given key, B-Tree implementations (we omit the between nodes link of the B+-Tree) are a mixture of binary tree (between the nodes) and linked list (within the node). A later development, the T-Tree still is of this mixture, but when it comes to a look up, a single list iteration is needed, within the last node. We argue for this reason, that the T-Tree outperforms a B-Tree implementation on a self-developed look up benchmark with 1024^2 Keys (medium to large size table).

T-Tree

Being a modification of a standard binary tree, a T-Tree has b memory cells in each tree node. Unlike a B-Tree, the b cells don't have a child node each. It is just that each node has a left and a right child with values smaller or higher than the node's min and max, respectively. Upon look up, the searched value is compared with this min or max and the node is either considered or simply left aside as in the binary tree. We argue that for n data rows, ie. n keys, a tree depth of $d = \log(2; n/b)$ is required. A look up will therefore take $d = \log(2; n/b)$ descending steps as well as b/2 steps to iterate through the list with in the node, on average.

In Total: $b/2 + \log(2; n/b) \Rightarrow O(n) = \log(n)$

From this formula, we can see, that b must be chosen wisely. In case, b is too small, let say 1, the T-tree becomes a binary AVL-Tree, having only the descending logarithmic component. In case, b is too large the tree transforms to a linear list, with n/2 steps by iteration on average.

B-Tree

Being also a modification of a standard binary tree, a B-Tree also has b memory cells in each tree node. Here, each value points to a child having smaller or larger values than the value, itself. Upon look up, both the tree is descended, as well as the list of values must be iterated in each node, that is passed by. We therefore obtain for n data rows with a depth of $d = \log(b; n/b)$, $d = \log(b; n/b)$

descending steps, with a factor of b/2 on average for the iteration.

In Total: b/2*log(b;n/b) => O(n) = log(n)

Theoretical Comparison

As already described, the T-Tree has an advantage when descending to the leaf, because the algorithm can omit all nodes that are smaller or bigger than the given value. Only the last node, which contains the key, needs to be iterated through. In the B-Tree implementation, the iteration takes place in every node. We therefore propose that T-Tree will be faster.

Benchmark

Both, B-Tree and T-Tree are tested in a DOS environment. Pascal has been the Programming language at hand. The reason is simple: Under DOS, there is no page file swapping of the Operating System that could confound the benchmark.

The benchmark is executed for 1024^2 keys with different node sizes for both B-Tree and T-Tree. The execution time is measured.

The different node sizes will give a hint on a possible optimal node size (Please note, that I was unable to calculate this minimum from the formulas above).

Results

The following table shows the average look up time, for different node sizes.

Mittelwert von Time		TreeType	
Command	Size	Ttree	Btree
lookup	2	91,5	142,25
	4	81,3125	105,625
	6	68,3125	93,8125
	8	68,875	88,8125
	10	62,1875	84,625
	12	63,5	87,5625
	14	59,875	84,8125
	16	62,1875	83,6875
	20	61,5625	88,1875
	32	60,5	97,1875
	36	61,4375	104,75
	48	63,5625	102,9375
	64	69,3125	114,5
	96	79,0625	143,75
	128	89,6875	183,6875
	256	133,125	257,5625
lookup Ergebnis		**73,5**	**116,484375**

We clearly see that T-Tree is faster for all node sizes.

Conclusion

In this document, we were able to show that a T-Tree implementation outperforms a B-Tree for 1024^2 keys in a DOS environment. We further assume that the time savings are around 20% and that the T-Tree Node Size to Time-Relation has a minimum around 12 values per node. We assume the findings hold for other operating systems.

Bibliography

Lehman, Tobin J.; Carey, Michael J. (1986). *A Study of Index Structures for Main Memory Database Management Systems*. Twelfth International Conference on Very Large Databases