Cost Analysis of B+-Tree and CSB+-Tree in Main Memory Database

Lan Wang[1, a], Fengdong Sun[1, b]

[1] Department Of Computer Science and Technology , DaLian Neusoft University of Information Dalian,116023,China

[a]email: wanglan@neusoft.edu.cn, [b]email:sunfengdong@neusoft.edu.cn

Keywords: Main Memory Databases ; B+-Tree; CSB+-Tree; Cost Analysis;

Abstract. Index is indispensable in database system to speeding up data access. Commonly used indexes in main memory databases are B+-Tree, T-Tree and their variants. Cost model of main memory database are more complex than disk-oriental database, and relatively little work has been done on this area. This paper establishes a cost model for B+-Tree and CSB+-Tree, analyzes their main performance factors. We perform several experiment evaluations on the cost model.

Introduction

Index is an essential part of relational database to improve data access performance. For main memory database, the primary goal of index is to reduce overall computation time while using as little memory as possible [1].

With the development of computer hardware, to minimize the impact of speed gap of CPU and memory multiple caches are added between them. Correspondingly, cache sensitive index has been developed, two of which are CSB+-Tree and CST-Tree [2,3]. CSB+Tree is a variant of B+-Tree, whose node size is set to cache line size and pointer usage is restricted. Compared to T-Tree, nodes in CST-Tree are organized in group, and nodes of same group are stored contiguously.

Some researchers focus on cost model, which is the foundation of query optimization. In main memory database all data is kept in memory, so access bottleneck have moved up to RAM and CPU. Although multi-level caches are introduced to speed up memory access, it makes data cost model more complex in main memory database. At present, execution time and cache miss are usually used to evaluate main memory database query [4].

B+-Tree/CSB+-Tree's Cost Analysis

Suppose B+-Tree and CSB+-Tree indexes are created on relation R. We define the parameter symbol in Table 1. Subscript i and l represent inner node and leaf node.

TABLE I. PARAMETER SYMBOL OVERVIEW

Symbol	Description	Symbol	Description	Symbol	Description
n_i	inner node key number	$\|c\|$	cache line size	$\|k\|$	key size
n_l	leaf node key number	$\|m\|$	metadata size in node	s	range query selectivity
$\|n_i\|$	size of inner node	$\|d\|$	rid size in leaf node	$\|p\|$	pointer size
$\|n_l\|$	size of leaf node	t	total record numbers	u	fill factor

According to parameters above, key number of inner node and leaf node are:

$$n_{i_{B+}} = \frac{(|n_i| - |m_i| - |p|)}{|k| + |p|} \qquad n_{i_{CSB+}} = \frac{(|n_i| - |m_i| - |p|)}{|k|} \qquad n_l = \frac{(|n_l| - |m_l| - 2|p|)}{|k| + |d|}$$

Tree's fan-out is:

$$f = u * n_i + 1 \qquad f_{B+} = u* \frac{(|n_i| - |m_i| - |p|)}{|k| + |p|} + 1 \qquad f_{CSB+} = u* \frac{(|n_i| - |m_i| - |p|)}{|k|} + 1$$

Number of leaf node and depth of tree are:

$$L = \frac{t}{u*n_1} \qquad h = \log_f L + 1 = \log_f \frac{t*\left(|k| + |d|\right)}{u*(|n_1| - |m_1| - 2|p|)} + 1$$

In B+-Tree and CSB+-Tree, point query is processed like this: to compare in inner node to find the right sub-tree, and then move to the sub-tree until we reach the leaf node. Whether required key exists or not, the path traversed is no difference. The cost of point query consists of two parts: the cost of comparing in inner node, and the cost of movement between different tree hierarchies. The comparisons number in inner node is Log2(u*ni), so the total number of comparisons and branch miss-prediction is:

$$NUM_{compare} = \sum_{i=1}^{h} Log_2(u*n_i) \qquad NUM_{Branch\_Misp} = \frac{1}{4} * \sum_{i=1}^{h} Log_2(u*n_i)$$

Moving between different tree levels invokes cache miss and TLB miss. For cache miss, one moving invokes one cache miss if node size is smaller than cache line size. The value is if node size is bigger than cache line size. Because we always need to access the first cache block of the node, we add cache miss $\frac{|c|}{|n|}$ for each access.

$$NUM_{L1\_miss_{(|n|<=|c|)}} = h \qquad NUM_{L1\_miss_{(|n|>|c|)}} = h*(Log_2 \frac{|n|}{|c|} + \frac{|c|}{|n|})$$

When moving between tree levels, each level invokes one TLB miss.

$$NUM_{TLB\_miss} = h$$

And L2 cache miss can be presented as:

$$NUM_{L2\_miss} = \frac{NUM_{L1\_iss} * SIZE_{L1}}{SIZE_{L2}}$$

The cost of range query is divided into two parts: the time to locate the first leaf node and the time to traverse leaf node links. The cost to find the first leaf node is the same as point query. The leaf node traversal cost is:

$$NUM_{compare} = t*s \qquad NUM_{L1\_miss_{(|n1|<=|c|)}} = \frac{t*s}{u*n_1} \qquad NUM_{L1\_miss_{(|n1|>|c|)}} = \frac{t*s}{u*n_1} * \frac{|n_1|}{|c|}$$

$$NUM_{TLB\_miss_{B+}} = \frac{t*s}{u*n_1} \qquad NUM_{TLB\_miss_{CSB+}} = \frac{t*s}{f_{CSB+} * u*n_1}$$

Now let's consider of insert operation. The cost of insert can be divided into three parts: the time to find insert position, the time of data movement within the node or node split, and the time of data insertion. Insert position finding is a point query operation. The cost of data movement within a node is about:

$$NUM_{move} = \frac{u*|n|}{2}$$

As for the node splitting cost, B+-Tree need to move half of the data of a node, while CSB+-Tree have to move half of the data of a node group.

$$NUM_{split_{B+}} = \frac{|n|}{2} \qquad NUM_{split_{CSB+}} = \frac{f_{CSB+} * |n|}{2}$$

Experiment

We ran our experiments on an intel® Xeon® machine. The hardware detail is shown in Table Ⅰ.

We choose SUSE Linux Enterprise Server 10 SP2 (x86_64)-Kernel 2.6.16.60-0.21 as the operating system.

TABLE II. EXPERIMENTAL MACHINE DETAIL

Device	Description	Device	Description
CPU	Intel® Xeon® CPU L5408@2.13GHz	Cycles Per Instruction	0.7
CPU Type	Intel Core 2 45nm processor	L1 cache cycles	3
CPU Cores	2 sockets * 4 cores/socket	L2 cache cycles	40
Cache Line Size	64 Byte	TLB miss cycles	40
L1 Cache Size	32K	Branch misp cycles	10
L2 Cache Size	6M	Memory latency cycles	300

We implemented a B+-Tree and a Full CSB+-Tree. Both of them support bulk load, point query, range query and insert operation. Experiments conducted are:

(1)To compare the overall performance of two trees on point query, range query and insertion including execution time, cache misses, branch miss-prediction etc. The difference analyzing is also need.

(2)To adjust the parameters of Section Parameters Effects to verify B+-Tree and CSB+-Tree performance impact factors: number of records, key size, pointer size, inner node size, leaf node size, record orderliness, range size and fill factor.

Unless otherwise stated, the experiment is based on a data of 10,000,000 random keys. We repeatedly perform our experiment to get an average result. Intel® VTuneTM Amplifier 2013 [20] is used to acquire performance events in Table II. As our code size is small, performance events concern instruction miss is not considered.

TABLE III. INTEL VTUNE COUNT EVENTS

VTune Events	Description	VTune Events	Description
CPU_CLK_UNHALTED.CORE	CPU cycles of the execution	INST_RETIRED.ANY	Total CPU instructions
BR_MISSP_EXEC	Branch miss-predictions	DTLB_MISSES.ANY	DTLB miss
L1D.REPL	L1D cache miss	L2_LINES_IN.SELF.ANY	L2 cache miss

We also record the run time besides above events.

In the experiment, we set node size the same as cache line size (64Byte). Other parameters are shown in Table II.

Point query: To perform 100,000 times random point queries on 10,000,000 records.

Range query: To perform 10 times range queries on 10,000,000 records. The condition keys are randomly generated with query selectivity 10%.

Insert:To insert 100,000 records to 10,000,000 records. The insert keys are randomly generated.

TABLE IV. PARAMETERS OF EXPERIMENT

Parameter Name	Value	Parameter Name	Value
Leaf node size	64	Inner node size	64
Key size	4	Rid size	4
Pointer size		Fill factor	0.7
Inner node key num of B+-Tree	7	Inner node key num of CSB+-Tree	14
Leaf node key num	6		

Test result is shown in Figure.1. The main vertical axis shows the number of CPU cycles That VTune has collected (in Million). The second vertical axis shows the execution time (s).

It can be seen that the main performance affecter is L2 cache miss. For point query, CSB+-Tree outperforms B+-Tree. That is because CSB+-Tree's inner node spend less space to store pointers, which leads to less L2 cache miss. For range query, time is mainly consumed in leaf node traversal, so the time difference of finding the first key can be neglected. One interest thing is the insert operation. Despite the CSB+-Tree nodes need to move more data in node splitting. But the insert performance of B+-Tree is not better than CSB+-Tree. That is because of the 0.7 fill factor, which means that inserting %1 records does not cause too much node splitting. Thus insert performance is mainly determined by the key search time, where CSB+-Tree outperforms.

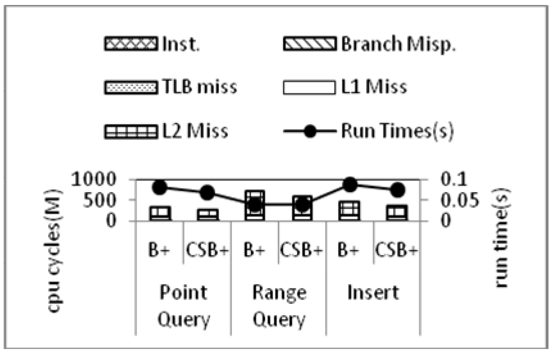

Figure 1 Overall performance of B+-Tree and CSB+-Tree

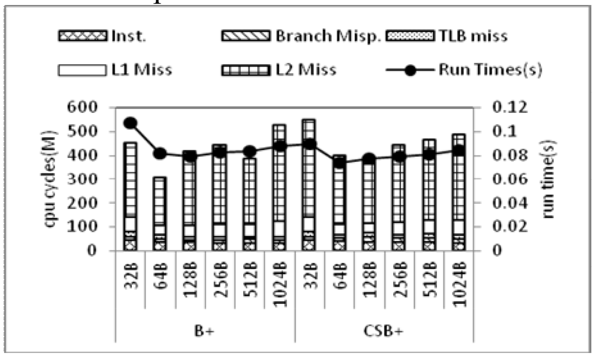

Figure 2 Inner node size impact

Conclusions

This paper analyzes the B+-Tree and CSB+-Tree access cost, and established a model to evaluate their performance. From the cost model, we find several parameters that affect B+-Tree and CSB+-Tree's performance. We verified B+-Tree and CSB+-Tree performance while varying different parameters. The conclusions are:

(1)Performance is significantly higher when node size>= cache line size. When node size is two times larger than cache line size, both B+-Tree and CSB+-Tree's performance will slowly decrease. Inner node size and leaf node size have similar impact on query performance. There is no reason to have them keep different values.

(2)For range query, B+-Tree and CSB+-Tree's performance has no big different. Both performances decrease linearly according with selectivity.

(3)No matter for insert or query, fill factor increasing improves the performance of B+-Tree. CSB+-Tree has the best insert performance when fill factor is 0.7.

References

[1]Lehman Tobin J et al. A study of index structures for main memory database management systems[C]. Proceedings of the 12th VLDB Conference, Kyoto, Japan, 1986: 294-303.

[2]Rao Jun, Ross Kenneth A. Making B+-Trees cache conscious in main memory[C]. Proceedings of the 2000 ACM SIGMOD international Conference on Management of Data, Dallas, Texas, USA, 2000: 475-486.

[3]Lee lg-Hoon, Shim Junho et al. CST-trees: Cache sensitive T-trees[C]. Proceedings of the 12th International Conference on Database Systems for Advanced Applications, Bangkok, Thailand. 2007: 398-409.

[4]Shan Wang, Yanqin Xiao . Research of main memory database[J]，Computer Application Oct. 2007, vol 27, No. 10 2353-2357, In Chinese.

[5]Lee lg-Hoon, Jae-won Lee. Cache Conscious Trees on Mordern Microprocessors[C], Proceedings of the 4th International Conference on Ubiquitous Information Management and Communication, ICUIMC 2010, Suwon, Republic of Korea, January 14-15, 2010.