Introducing Preparatory Operations into Concurrency Control in Parallel B-trees

Damdinsuren AMARMEND, Masayoshi ARITSUGI and Yoshinari KANAMORI

Department of Computer Science, Faculty of Engineering, Gunma University

1-5-1 Tenjin-cho, Kiryu, Gunma 376-8515, Japan

E-mail: {amarmend, aritsugi, kanamori}@dbms.cs.gunma-u.ac.jp

Abstract Parallel and distributed B-tree structures and their concurrency control protocols for shared-nothing parallel database systems have been proposed and are being developed in connection with the related technological advancements such as clustered computers. We focus specially on the performance of the concurrency control protocols for parallel B-trees and consider the possibility and suitability of introduction of preemptive split or merge method to them. We propose a protocol, called PO-P, in which preparatory operations are used during traversals of updater processes, and we expect this protocol can serve as a good alternative to the present protocols for parallel B-trees.

Keyword parallel shared nothing computers, B-tree, concurrency control protocol, parallel and distributed database systems.

1. Introduction

Shared-nothing parallel database systems are being an interesting subject of study in terms of scalability and availability to tackle large-scale data processing problems. Shared-nothing parallel database systems comprise independent processing elements (called PE or node) each of which has own memory, disk and CPU, and are interconnected via high-speed network.

In shared-nothing parallel database systems, each PE is able to serve a request to read, update or search a tuple. If the serving PE has no further information to perform the request, it can transfer the request to the appropriate PE to continue the service.

Parallel B-trees are being considered to be well suited for fast and sequential retrievals and management of data partitioned across PEs by value ranges. And their concurrency control protocols are being proposed, for instance, Fat-Btree [1] and its concurrency control protocol INC-OPT [2]. We focus our study on concurrency control protocols for parallel B-trees.

2. Concurrency Control in Parallel B-trees

There are processes of two types run on B-trees: retrieval (e.g., read or search a tuple) and update (e.g., insert or delete a tuple). In general, concurrency control protocol in B-tree is for how to run processes on the tree so that the throughput of the system is the maximum. The throughput is the number of operations done per unit of time, i.e., degree of concurrency. During the concurrent operations, the processes on the tree must keep the physical consistency of the tree. While one process changes an index node, another process is not allowed to read the node. To avoid such inconsistency, concurrency control protocols use locks and latches.

2.1 Locks and Latches

Both lock and latch are the mechanisms to regulate the access of the processes into tree's nodes. In general, lock is for logical consistency of a data object while latch is for physical consistency of a data object, and the former is more complex and more expensive than the latter one. Thus a concurrency control protocol that uses latch is expected to be better, although it must be designed as deadlock free because latch mechanism does not have deadlock detection. We will assume in this paper latches. Retrieval and update processes use different kind of latches. Granule latches and their compatibilities are illustrated in Fig.1. 'Y' means the corresponding latches are compatible. Retrieval processes use IS and S latches while update processes use IX, X and SIX latches. IS means intended shared latch, IX means intended exclusive and SIX means intended shared-exclusive latch on the descendant nodes. From the Fig.1, if a process holds a IS latch on a node, another process can get IX latch on the node.

Mode	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	
IX	Y	Y			
S	Y		Y		
SIX	Y				
X					

Fig.1 Latch compatibility matrix.

Processes use latch-coupling technique during its traversal from the root node. The latch-coupling comprises of the following steps:

1. Determine the next child from the node on which the process already has a latch.
2. Acquire latch on the child node.
3. Release the parent node's latch.

Retrieval processes can use S or IS latch-coupling whereas update processes are able to use X, IX and SIX latch-coupling.

2.2 Related Work

In [2], they described that conventional concurrency control protocols for B-tree such as Bayer-Schkolnick's B-X, B-SIX, B-OPT [6], OPT-DLOCK [9], ARIES/IM [7] and B-link [8] are not suitable for parallel B-trees, and revealed that a special protocol is necessary. Since the case for read process is easy to solve, we mostly will describe about on update process.

The INC-OPT is an optimistic protocol, which differs from the B-OPT in the second phase when a structure modification operation (SMO) such as split or merge happens during an update. During SMO, INC-OPT releases all the previously obtained latches and traverses again from the root by IX latch-coupling until the top update scope node whereupon it gets X latches on the whole update scope. The update scope could be the path from the root to the desired leaf, in the worst case.

As an alternative, we study the suitability of introduction of preparatory operations (PO) into parallel B-tree's concurrency control. If a node has full or insufficient entries then it will be referred to as an unsafe node, otherwise as a safe node. If an entry is inserted into or deleted from unsafe node then that will cause split the node or merge with another node, respectively. If an unsafe node is found on the way of update process's traversal from B-tree's root to the desired leaf, the process does split or merge operations which are called preparatory operations (PO) on the unsafe node for making it a safe node. There are some top-down concurrency protocols named TD-X, TD-SIX and TD-OPT [3][4] in which POs are used, however, they are designed for conventional B-tree.

According to these protocols, a process traverses using lock coupling from the root to the target leaf, and during the traversal, it checks the child node to know whether it is safe or unsafe. If any unsafe node occurs it locks both parent and child nodes, then does PO and continue its traversal. A read process lock traverses using IS lock coupling in the all three protocols. In TD-X, update process uses X tree lock coupling during traversal thus blocks other processes on the locked node(s). In TD-SIX, update process uses SIX lock coupling, thus allows other read process to read the locked node but not other update processes. If the process faces an unsafe node, it tries to convert SIX locks into X. In contrast, an update process in TD-OPT has two phases when it encounters an unsafe node. Until the unsafe node occurs it lock couples using IX then releases the lock and traverses again from the root using SIX lock coupling.

As for the performance, TD-OPT outperforms other two protocols but in the second phase, it still blocks much other update processes. Making propagated updates into some small operations and also its simplicity are the interesting aspects of this approach for the parallel B-tree.

3. PO-P Protocol

We propose optimistic PO-P (preparatory operations-parallel) concurrency control as an alternative protocol for parallel B-tree structure. If a node to be read is not available on the PE the request will be sent along with request type, search key and remote starting node address to the appropriate PE. The intended latches such as IS or IX are only used on local nodes. A read process will start as getting IS latch on the root as in other optimistic protocols. Further traversal will use IS latch coupling until the desired leaf is found. The leaf will be latched with S latch. If the leaf is on the PE then all the nodes along the traversal reside on the same PE.

For an update process, it will start from root node by IX latch coupling as TD-OPT does. If the updater encounters no unsafe node along the path until to the leaf, the leaf will be X latched and updated, and the process is complete. However, during the traversal if the update process finds unsafe node (that necessitates PO), it releases parent's IX latch, then restart its traversal from

the root again. In the second phase, the update process traverses the tree down by IX latch coupling to obtain X latches on the parent and unsafe node of the previous traversal. The process tries to get X latch on the parent and its copies on other PE-s, after that tries to get also X latch on the child and its copy nodes. After all the necessary latches are obtained the process starts PO on the child and reflects the change in the parent. POs on the copy nodes will be done in parallel. In Fig.2, PO-P protocol is shown in detail.

```
1.  height := H; unsafe_node := No;
2.  Parent := NULL; Child := ROOT; h := 1;
3.  while h < height do begin
4.     latch IX on Child; unlatch Parent;
5.     if Child is unsafe then begin
6.           unlatch Child; height := h-1;
7.           unsafe_node := Yes; goto 2; end;
8.        else begin
9.             Parent := Child; Determine nextChild;
10.            Child := nextChild; h := h+1;   end;
11.    end;
12. end;
13. if unsafe_node = Yes then begin
14.    latch X on Child and its copies;
15.    unlatch Parent; Parent := Child;
16.    Determine nextChild; Child := nextChild;
17.    latch X on Child and its copies;
18.    perform appropriate POs on Child;
19.    unlatch Parent's copies, Child and its copies;
20.    Determine the appropriate Child;
21.    unsafe_node := No; height := H;
22.    h:=h+1; goto 3;
23. end;
24. latch X on Child; unlatch Parent;
25. if Child is unsafe then goto 6;
26. else update Child; unlatch Child;
```

Fig.2 PO-P protocol.

In shared-nothing parallel B-tree structure, some nodes are replicated across the PE-s and their information is reachable from the corresponding parent nodes. We assume distributed latch manager and latching on the nodes which have remote copies on another PE-s will be obtained by requesting to the remote latch managers in sequential order as same as INC-OPT. Only X latch will be requested remotely if it is necessary. PO-P uses latches so must be deadlock free, and must persist B-tree's physical consistency.

Theorem 2.1: The PO-P protocol is deadlock free.

Proof. For the remote copy nodes across many PE-s, latching process will use sequential order of the related PE's order. Suppose a node A is replicated as A_1 on PE1 and A_2 on PE2 and A_n on PEn. Then let's assume 2 processes on different PEi and PEj (i, j $\subseteq$1…n) try to get latch on the copies starting from PE1 to PEn. Since local latch manager on PE1 gives only one of them, the remaining one will wait until the succeeded process releases the latch on PE0. From this, no deadlock will happen in this case. Next, we consider deadlocks of different nodes. Let's take a process p and a process q, which precedes p along its traversal path from the root to the target leaf. If a deadlock occurs between these processes, p holds a node on which q wants to have a latch. It means that the node is child node for p and parent node for q due to PO-P's traversal. It is impossible because of B-tree descendant order and latch-coupling technique. When a re-traversal occurs, the p releases its latch on child node due to PO-P. Thus, this situation will not happen. Hence, it is proved that PO-P to be a deadlock free protocol. □

Theorem 2.2: The PO-P keeps the physical consistency of B-tree structure.

Proof. Read processes will not cause any change to the physical structure of B-tree. During IS latch coupling, once a node is IS latched, any POs will not be made on the children nodes. If the child node is determined to be located on some other PEs, the process transfers the request to one of them on random basis. The sending process will unlatch the parent node after the sending process is complete. The receiving process is supposed to continue the traversal from the subsequent level, however if it is not able to get IS latch on the node, it must restart from the root. Otherwise, an update effect on the node may lead the process's traversal into wrong direction. Therefore, the retrieval process will navigate correctly and will see the tree consistently.

Despite using IX latches, update processes traverse down the tree in a similar way with retrieval processes until they find unsafe children nodes. Before an update all the necessary latches are obtained including copy nodes. A deadlock free protocol will be used to latch copy nodes. Other

processes will not be able to get latch on the nodes of the scope. Updating the leaf will not cause any SMO as well any propagated update up to the higher levels because there will not be any remained unsafe node on the path of the process from the root to the leaf. The leaf is also checked before the update whether it is safe or not. Only after the updates are done on all the nodes, the latches are released. Multiple concurrent POs and leaf updates can occur simultaneously, however, their update scopes don't go crossed. Therefore, every process will keep B-tree's portion as consistently. Thus, PO-P will always keep the whole B-tree physically consistent.□

4. Expected Performance Results

PO-P is designed to work on shared-nothing parallel B-tree such as Fat-Btree. Hence, low latch synchronization cost, high throughput performance and short response time must be taken care of on it. For read processes no difference from INC-OPT. If there is no unsafe node during an update process, it will also be the same as INC-OPT. But, in case of some unsafe node on the path of the update process, it will need to make same number of restarts from the root in order to do the respective POs to the unsafe nodes. We could compare PO-P with TD-OP because they both use the same PO approach. But in case of unsafe node occurrence, TD-OPT starts traversing SIX and thus blocks other update processes, it means lowering throughput performance. PO-P looks similar with INC-OPT, but the difference is PO-P is using POs in parallel situation. In Fig.3 an example of a parallel B-tree, which has 3 levels and is distributed on 3 PEs. Assume a process on PE1 recognize during its traversal that lower two nodes are unsafe. In case of using INC-OPT, the process in its second traversal will get latches on the scope and on root nodes of PE2 and PE3. The processes on PE2 and PE3 will wait until PE1 completes the update and releases the latches. In this case, if the number of PEs increases the number of processes waiting will increase. And also the height of the scope increase, the waiting time can increase as well.

In case of using PO-P, first, the process latches only the root and the intermediate node. Then as soon as it completes PO on the node, the root nodes are released. This makes processes on PE2 and PE3 able to traverse the root while the process on PE1 performs PO on the intermediate and leaf nodes locally. Thus, PO-P is expected to outperform in case of such highly propagated updates. The probability of upper unsafe node occurrence in PO-P, and the probability of highly cascaded update occurrence in INC-OPT are intuitively close. Not necessary preparatory operations may seems to cause some overheads, but it will be balanced by fastening next processes not letting them to make any update propagation up to the upper nodes. And mostly, B-tree height is low, thus many simultaneous PO occurrence along any update process's path is rare. PO-P does remote latch synchronization when only if the parent and/or child nodes have copies on other PEs and the child is unsafe. Thus, it is expected PO-P's performance is not worse than INC-OPT and to be able to serve well as an alternative concurrency protocol for this kind of B-tree.

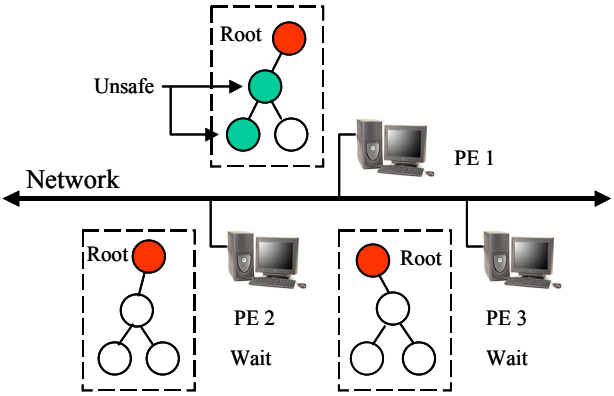

Fig.3 An example of parallel B-tree on a parallel machine.

5. Experiments and Results

In order to see PO-P's performance practically, we implemented it and INC-OPT protocol on a small Fat-Btree as a base parallel on a shared-nothing parallel machine. According to Fat-Btree structure, each copy node has its appropriate information of own copy and remote children on separate pages, which is accessible from the node. We use 4KB for page, and 208B for tuple. Therefore, an index node can have ⌊4096-32 (header)-1⌋ = 507 entries at max. And a leaf page can hold ⌊(4096-32)/208⌋ = 19 tuples at max.

5.1 Communications between PEs

We use LAM/MPI 6.5.8 implementation for the point-to-point communications between PEs. MPI provides easy to program primitives such as

MPI_Send and MPI_Recv. Both sending and receiving operations can be either blocking or non-blocking. If threads are used in request processing they need to communicate with other remote threads by messages, therefore MPI implementation used in the experiment must be thread-safe and support multi-thread communication.

5.2 System Structure

The general system structure is shown in Fig.4. We assume three workstations connected through network as the underlying shared-nothing parallel machine. The PEs numbered from 1 through 3 have same specifications which shown in Table 1. All the implementations are done in main memory of the machines because we focused on to measure the performance of concurrency control protocols. We use PE0 for the tree initialization, and request generator to the parallel machine. PE0 does the following:

1. Initialize the global B-tree on it.
2. Distribute the tree to PE1-PE3 according to the pre-determined value ranges.
3. After completing step 2, send requests.

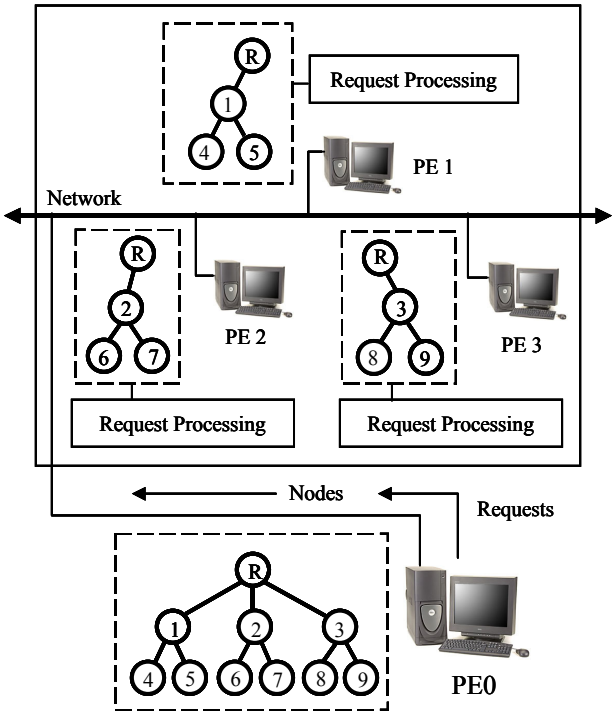

Fig.4 System structure.

In Table 2, the initial B-tree configurations, which are used in the experiments, are listed. After step 2, PE1 through PE3 are becomes ready to receive requests.

Table 1. Specifications of PEs.

Processing elements	CPU speed	Memory size	OS and version
PE0	296MHz	128MB	SunOS 5.6
PE1~PE3	248MHz	-	-

First column is sizes of the initially inserted tuples ranging from 50K to 130K, next is the corresponding tree height, and next 3 columns are number of nodes in the levels. In the fourth column, the intermediate nodes occupancies are also showed.

Table 2. B-tree configurations.

Tuples	H	$N_n(1)$	$N_n(2)$	$N_n(3)$
50,000	3	1	8 (92%)	3728
80,000	3	1	16 (74%)	6021
90,000	3	1	16 (77%)	6713
100,000	3	1	16 (82%)	7472
130,000	3	1	32 (59%)	9723

A request consists of:

1. Type (retrieval | update)
2. Key (attribute value).

We assume here only insert operations for update and read operations for retrieval. In each experiment, PE0 sends 6000 requests. In Table 3, system's initial performance data is given. Remote local latch overhead is rather higher than the local latch because of messaging cost and also waiting time for other process to release latch.

Table 3. System performance.

Communication throughput of the network	18Mbps
Message setup time	93ms
Access time for a page in memory	1.5ms
Latch overhead in a local PE	2.5ms
Latch overhead to a remote PE	419ms

5.3 Request Processing on a PE

Each PE has request processing logical module, which is illustrated in Fig 5. Request reception module receives requests, which can be originated from PE0 or transferred from another PE. Depending on the type, the reception will give the request to one of the modules: retrieval request processing and update request processing. Those two modules traverse the tree and interact with

latch manager in order to obtain appropriate latches on the nodes. Each PE has local latch manager, which is responsible for latch and unlatching of local nodes. During update operations on copy nodes, update process may make remote latch requests to remote latch managers for copy nodes.

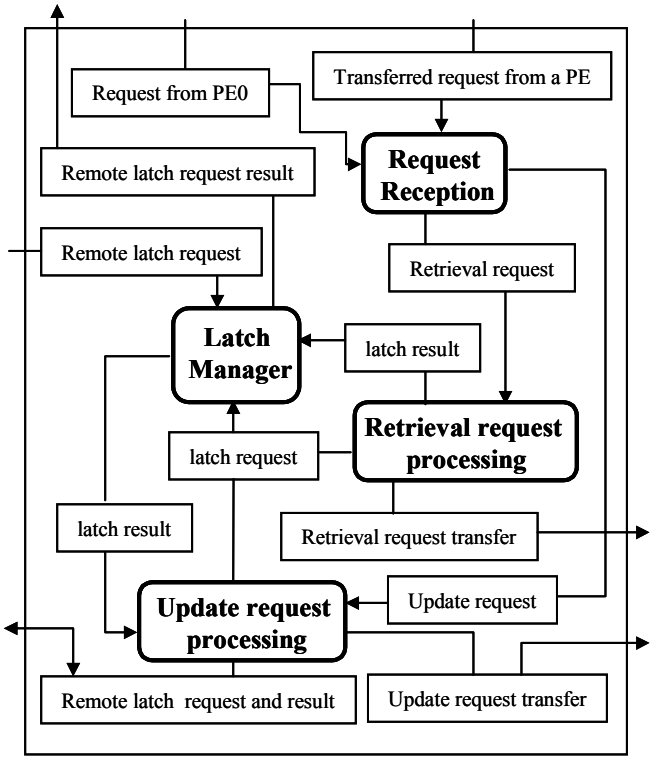

Fig.5 Request processing part in a PE.

The difference of the two protocols lies in update request processing part, especially in case of node splits. Only split SMO is considered in this paper because merge SMO is similar. If an SMO happens on copy nodes, the initiating PE must send to other PEs the whole parent scope information, while PO-P sends only one level up information.

5.4 The Performance Results of the Two Protocols

Four experiments results will be discussed in this section. First three experiments have only difference that is the number of PEs, which ranges from 1 through 3. The results of the four experiments are shown in Figs 6 to 8. The vertical axis of all the graphs denote throughput i.e., main performance degree. The horizontal axis of the first three graphs shows the changes of update ratios in the number of requests. Update ratio ranges from 0 to 100% by 20% increase, because the difference of the two protocols will be observed in case of high update rates. In the fourth experiment, we intended to see the performance when the number of tuples increases.

In the first experiment, only one PE manages the tree nodes and the operations on the tree. The data is located on one place and, all the requests are done sequentially.

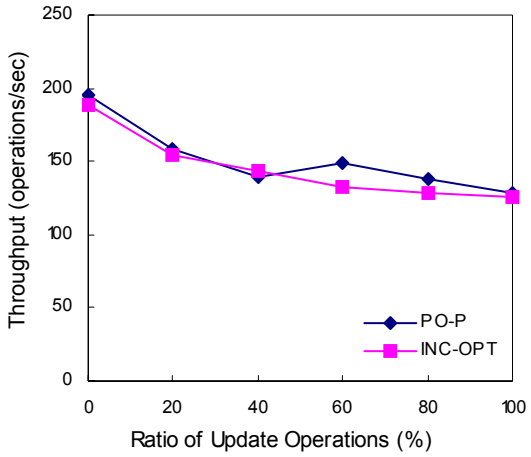

Fig.6 Throughput of the two protocols under 1 PE. (Number of tuples: 54K, Number of requests: 6000)

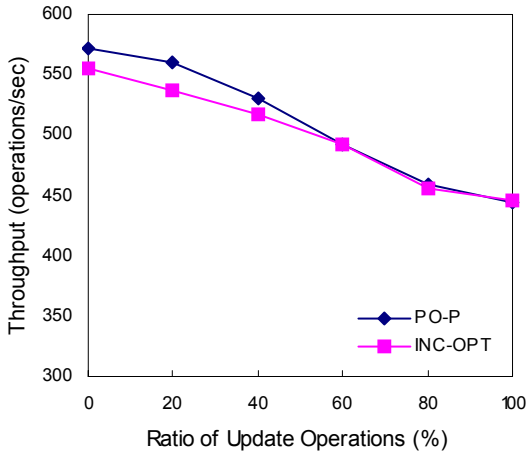

Fig.7 Throughput of the two protocols under 2 PEs. (Number of tuples: 54K, Number of requests: 6000)

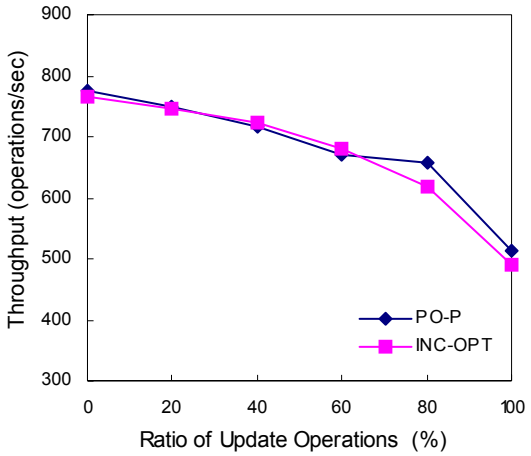

Fig.8 Throughput of the two protocols under 3 PEs. (Number of tuples: 54K, Number of requests: 6000)

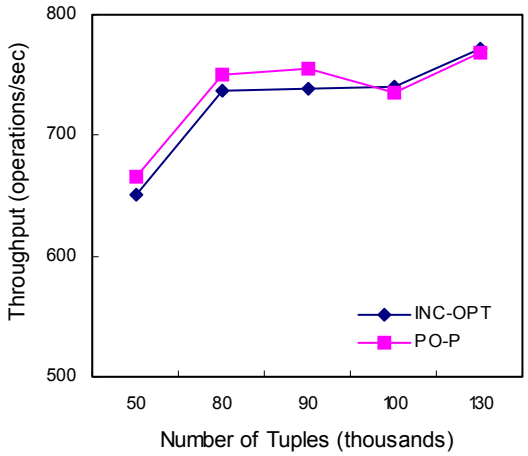

Fig.9 Throughput of the two protocols under 3 PEs. (Number of tuples: 54K, Number of requests: 6000)

When number of update ratio increase around 60% the PO-P shows slight improvements. The throughput is very low because no parallel operations are done. In Fig.7, two of PEs involves. The throughput is improved than previous one for both protocols however much performance difference between is not observed.

In Fig.8, three of PEs participates in the experiment. The throughput is increased than previous two experiments because of parallel data processing. We could see slight difference between them when update ratio grows from 80% up to 100%. We choose the B-tree initial configuration so that the intermediate nodes have 70%-100% occupancy, i.e., intermediate node split will often occur. When the intermediate splits often occur, INC-OPT often latches all the root nodes and will not release them until it finishes the update. Instead, PO-P releases the root nodes as soon as the process completes split of the intermediate node. When the number of PE and the tree height increases the difference will be bigger.

In Fig 9, the result of the last experiment is shown when the number of data increases. As the number of tuples increases the performance increases for both protocol. All the data is stored in main memory that is cache hit rate is 100%. Therefore, throughput is not depending on the data size because access time is the same. The main influence is the occurrence of the intermediate nodes.

From the Table 2, we can see the initial node occupancy and number of the intermediate nodes. When the size of the tuples 50K, 80K and 90K, the occupancy is higher and the number of intermediate nodes is fewer than that of 100K and 130K. However, if the data is stored initially on disk, the throughput will decrease when the data size increases because cache hit rate becomes lower.

From these experiments, we conclude that the performance of the PO-P is not worse anytime than the INC-OPT and may outperform INC-OPT in some cases.

6. Conclusions

We have proposed a protocol, named PO-P for shared-nothing parallel B-trees such as Fat-Btree, as an alternative. In our protocol, we adopted preparatory operations (PO) for update processes. Using PO enables highly propagated updates performed in separate small operations while other optimistic protocols perform it once on the whole scope. PO-P deals with only two levels latching all the time. This feature of the protocol makes the copy latch waiting time shorter for processes in the distributed environment. We expect PO-P's performance will be better than the current methods when the number of PE increases and the update ratio is high and in other situations the protocol will show at least the same performance.

Our further work will include the improvements on the protocol, e.g., perform POs in case of they are really necessary.

References

[1] H. Yokota, Y. Kanemasa and J. Miyazaki, “Fat-Btrees: An update-conscious parallel directory structure,” Proc. IEEE ICDE Conf., 1999.

[2] J. Miyazaki and H. Yokota, “Concurrency control and performance evaluation of parallel B-tree structures,” IEICE Trans. Inf. & Syst., vol.E-85, no.8, pp.1269-1283, Aug. 2002.

[3] Y. Mond and Y. Raz, “Concurrency control in B+-trees databases using preparatory operations,” Proc. VLDB, 1985.

[4] J. R. Haritsa and S. Seshadri, “Real-time index concurrency control,” IEEE Trans. Knowledge and Data Eng., vol.12, no.3, pp.429-447, May/June 2000.

[5] J. Gray and A. Reuter, “Transaction Processing: Concepts and Techniques,” Morgan Kaufmann, San Francisco, 1993.

[6] R. Bayer and M. Schkolnick, “Concurrency of operations on B-trees,” Acta Informatica, vol.9, no.1, 1977.

[7] C. Mohan and F. Levine, “ARIES/IM: An efficient and high concurrency index management method using write-ahead logging,” Proc. ACM SIGMOD Conf., 1992.

[8] P. Lehman and S. Yao, “Efficient locking for concurrent operations on B-trees,” ACM Trans. Database Systems, 1981.

[9] V. Srinivasan and M. J. Carey, “Performance of B-tree concurrency control algorithms,” Proc. VLDB, 1993.