A Concurrency Control Protocol for Parallel B-tree Structures without Latch-Coupling for Explosively Growing Digital Content

Tomohiro Yoshihara[1] * Dai Kobayashi[1] †
[1]Department of Computer Science,
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo 152-8552, Japan
{yoshihara,daik}@de.cs.titech.ac.jp

Haruo Yokota[2,1]
[2]Global Scientific Information and Computing Center, Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo 152-8550, Japan
yokota@cs.titech.ac.jp

ABSTRACT

While shared-nothing parallel infrastructures provide fast processing of explosively growing digital content, managing data efficiently across multiple nodes is important. The value–range partitioning method with parallel B-tree structures in a shared-nothing environment is an efficient approach for handling large amounts of data. To handle large amounts of data, it is also important to provide an efficient concurrency control protocol for the parallel B-tree. Many studies have proposed concurrency control protocols for B-trees, which use latch-coupling. None of these studies has considered that latch-coupling contains a performance bottleneck of sending of messages between processing elements (PEs) in distributed environments because latch-coupling is efficient for a B-tree on a single machine. The only protocol without latch-coupling is the B-link algorithm, but it is difficult to use the B-link algorithm directly on an entire parallel B-tree structure because it is necessary to guarantee the consistency of the side pointers. We propose a new concurrency control protocol named LCFB that requires no latch-coupling in optimistic processes. LCFB reduces the amount of communication between PEs during a B-tree traversal. To detect access path errors in the LCFB protocol caused by removal of latch-coupling, we assign boundary values to each index page. Because a page split may cause page deletion in a Fat-Btree, we also propose an effective method for handling page deletions without latch-coupling. We then combine LCFB with the B-link algorithm within each PE to reduce the cost of Structure Modification Operations (SMOs) in a PE, as a solution to the difficulty of consistency management for the side pointers in a parallel B-tree structure. To compare the performance of the proposed protocol with conventional protocols MARK-OPT, INC-OPT, and ARIES/IM, we implemented them on an autonomous disk system with a Fat-Btree structure. Experimental results in various environments indicate that the system throughput of the proposed protocols is always superior to those of the other protocols, especially in large-scale configurations, and LCFB with the B-link algorithm is effective at higher update ratios.

*This work was done while the author was with Tokyo Institute of Technology. The author is now working at Systems Development Laboratory, Hitachi, Ltd.

†The author is a JSPS Research Fellow (DC).

EDBT'08, March 25–30, 2008, Nantes, France.

1. INTRODUCTION

Recently, digital data has grown explosively in real life. On the Internet, large amounts of data is daily transferred between enterprises, and a great volume of content such as web information, audio and video streams, is provided to end customers. This growth means that computer systems must handle large storage spaces capable of supporting high throughput to enable users to access the data in parallel with high bandwidth, in order to realize such systems as high-performance B2B systems, web servers, and video-on-demand systems. To meet these demands, distributed storage systems in shared-nothing parallel environments have become increasingly widespread. However, accesses from many users in a distributed environment exhibit high skew, and huge distributed storage systems complicate extremely the management of storage, even though they provide high throughput. Distributed directory structures, including parallel B-tree structures, provide data placement that facilitates dynamic management and reduces complications. Storage systems with these structures can handle skew efficiently and can provide efficient accesses. Therefore, these structures provide not only simpler management to administrators but also higher throughput to users. They can support the high demands of recent years and are implemented in many systems.

The technique of virtualization helps to reduce the complication of storage systems. Virtualization is the pooling of physical storage from multiple network storage devices into what appears to be a single storage device that is managed from a central console. Storage virtualization is commonly used in Storage Area Networks (SANs) [24], for example, as the IBM Storage Tank [19]. The management of storage devices can be tedious and time consuming. Storage virtual-

ization helps the storage administrator perform the tasks of backup, archiving, and recovery more easily, and in less time, by disguising the actual complexity of the SAN. In storage systems that use SANs, metadata servers that manage data placement and accesses from users are separated from the nodes that store data, and the metadata servers implement virtualization. Distributed file systems such as the Google File System [7] and Cluster File System (Lustre) [4] also use metadata servers to manage data placement.

The simplest approach is to manage data placement with a centralized metadata server. However, a highly redundant centralized server needs to be implemented with high reliability parts in order not to create a single point of failure. Moreover, this approach leads to performance bottlenecks so it is important to implement a highly scalable metadata server. Therefore, in practice, distributed metadata servers are used. To manage data placement efficiently in distributed servers, distributed indexes are implemented (e.g., Autonomous Disk [29], Boxwood [18], and Bigtable [3]). In particular, parallel B-tree structures simplify the management of storage data [29, 18]. On the other hand, in some systems, the nodes that store the data manage the data. As an efficient virtualization in these systems, methods in which nodes cooperate by using distributed directories are proposed. Parallel B-tree structures also function efficiently as distributed directories.

To support the performance demands of users in rapidly growing storage systems, it is important to provide not only an efficient update-conscious parallel B-tree structure like the Fat-Btree [30] but also an efficient concurrency control protocol for the parallel B-tree or Fat-Btree. One example is the INC-OPT protocol, which is suited to parallel B-trees on shared-nothing parallel machines [21]. The INC-OPT protocol outperforms conventional B-tree concurrency control protocols such as the B-OPT protocol [1] and the ARIES/IM protocol [22]. However, the cost of spreading an X latch in the protocol is still high when structure modification operations (SMOs) occur frequently, degrading total performance.

A concurrency control protocol that is an improvement over the INC-OPT protocol, the MARK-OPT protocol [31] has been developed. It reduces the frequency of restarts traversing from the root node compared with the INC-OPT protocol. Experimental results on an autonomous disk system [29] using Fat-Btree as the distributed directory structure indicated that the MARK-OPT protocol outperformed the INC-OPT protocol. However, transferring access requests to another PE requires three network messages per transfer, because the traversal uses latch-coupling. As we have indicated, the cost of latch-coupling is high in a distributed environment. Moreover, the frequency of request transfers in a large-scale configuration is high. Therefore, the performance of MARK-OPT does not scale well with system size.

In this paper, we propose a new concurrency control protocol called the *latch-coupling-free parallel B-tree* (LCFB) *concurrency control protocol*, which is suited to Fat-Btrees. It reduces the cost of request transfers compared with MARK-OPT. In addition, we combine LCFB with the B-link algorithm, which reduces the cost of SMOs. The B-link can achieve excellent concurrency control, however it was difficult to effectively apply the B-link to a parallel Btree structure because it was necessary to guarantee the consistency of the side pointers. The combination is a solution to the difficulty of consistency management for the side pointers in a parallel B-tree structure. We implemented the proposed protocols and MARK-OPT, INC-OPT, and ARIES/IM[1] on an autonomous disk system [29] using Fat-Btree as the distributed directory structure, and we measured the system throughput as a function of system size. Experimental results indicate that the proposed protocols are effective and scalable, and LCFB with B-link is especially effective for higher update ratios. The only protocol existing to date without latch-coupling is the B-link algorithm, but it is difficult to directly use the B-link algorithm on an entire parallel B-tree structure. We focus on the cost of latch-coupling in distributed environments, and propose a protocol without latch-coupling and B-link.

The remainder of this paper is organized as follows. First, the methods for partitioning data among PEs, parallel B-tree structures and the concept of the Fat-Btree structure are reviewed in Section 2. Section 3 then describes existing concurrency controls for parallel B-trees. Our new concurrency control protocols for parallel B-tree structures are explained in Section 4. Experimental results are reported in Section 5. Section 6 presents a discussion of recovery in parallel B-trees. We review related work in Section 7. The final section presents the conclusions of this paper.

2. BACKGROUND

There are several ways to partition a large amount of data among PEs: value–range, round robin, and hashing [6]. Value–range partitioning can determine which PE should contain object data for strict match queries. It can also treat range queries, and cluster I/O operations to reduce the number of I/O operations for near values. However, it can degrade load distribution because it can potentially skew the distribution of data. Even though the initial data allocation has no skew, repeated updates may destroy the balance because partitioning criteria are statically fixed. On the other hand, round-robin partitioning can produce no skew. All PEs, however, must participate in every query for strict or range matches, because there can be no information about value location.

2.1 Hash Partitioning

Hash partitioning can provide fast access methods as well as partitioning strategies for parallel database environments. Hashing is not only a partitioning strategy, but is also an access method for fast retrieval. There are several papers on using hashing as a parallel index mechanism, such as [16, 5]. However, as described above, hashing cannot handle range queries, I/O clustering. In hashing, dynamic data migration handling access skews among PEs also has a high cost.

2.2 Parallel B-tree Structures

To exploit the virtues of value–range partitioning and to provide a fast access method for each PE, parallel B-trees have been proposed as parallel directory structures [27]. Parallel B-trees are useful because they can manage strict match and range queries and clustering I/O operations, and they can balance the amount of data in each PE via index nodes

[1]We implemented ARIES/IM based on [22], but we did not include its recovery mechanism because we do not focus on recovery in this paper.

of a B-tree. They can also locally perform dynamic data migration whose cost is low.

2.2.1 SMOs

Structure Modification Operations (SMOs) are page splits and page merges. Page splits are performed when pages become full, whereas page merges are performed when pages become empty. B-trees in real database systems usually perform page merges only when pages become empty; nodes are not required to be at least half full, because in practical workloads this is not found to decrease occupancy by much [11]. Because the consistency of the B-tree must be guaranteed when SMOs occur, concurrency control for the B-tree is necessary. Moreover, concurrency control is very important because it largely influences the system throughput when SMOs occur.

2.2.2 Request Transfers

Sometimes, an access request cannot be served from the current PE, as the destination leaf node is stored elsewhere. Such requests must be transferred to the PE that has the appropriate child node. To enable smooth and quick access transfers, such a parent node must have pointers that indicate remote child nodes. Moreover, each interaction between PEs for transferring access requests to another PE requires communicating across a network, with the number of the interactions influencing system performance. It is important to reduce the number of interactions for high scalability because the number of interactions increases as the number of PEs increases.

2.3 Partitioned B-tree Structures

A partitioned B-tree consists of a two-tier index structure. The first tier directs the search to the PE where the data is stored, and the data is range partitioned. The second tier is a collection of B^+-trees, one at each PE. Each B^+-tree independently indexes the data at its PE. However, the partitioned B-tree structure is not height balanced because the number of records on each PE can be different, and differing heights on the PEs degrade the total performance.

An aB^+-tree [14] is another two-tier index structure designed to maintain the global height-balanced property of indexes on all the PEs. The first tier of an aB^+-tree is similar to that of a partitioned B-tree, but each PE has a variation of a B^+-tree in the second tier, and the root node can be a fat node. Furthermore, all the B^+-trees across all PEs are of the same height. However, balancing the heights has a cost.

2.4 The Fat-Btree Structure

A Fat-Btree [30] is a form of parallel B-tree in which the leaf pages of the B^+-tree are distributed among the PEs. Each PE has a subtree of the whole B-tree containing the root node and intermediate index nodes between the root node and leaf nodes allocated to that PE. Fat-Btrees have the advantage of parallel B-trees (which hashing does not have) and Fat-Btrees are naturally height balanced. Figure 1 shows an example of a Fat-Btree using four PEs.

Although the number of copies of index nodes increases with proximity to the root node of the Fat-Btree, the update frequency of these nodes is relatively low. On the other hand, leaf nodes have a relatively high update frequency, but are not duplicated. Consequently, nodes with a higher update frequency have a lower synchronization overhead.

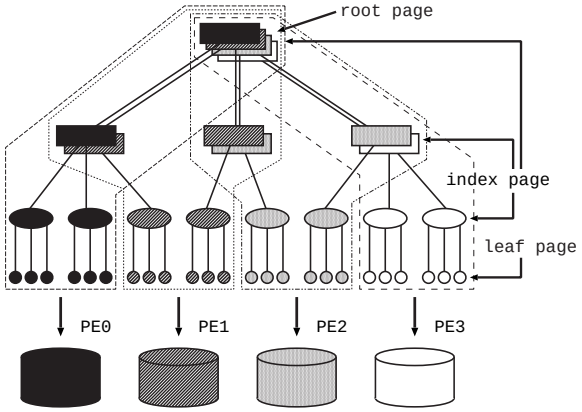


Figure 1: Fat-Btree.

Table 1: Latch matrix.

Mode	IS	IX	S	SIX	X
IS	◯	◯	◯	◯	
IX	◯	◯			
S	◯		◯		
SIX	◯				
X					

Moreover, with Fat-Btrees, index pages are only required for locating the leaf pages stored in each PE. Therefore, Fat-Btrees can have a high cache hit rate if the index pages are cached in each PE. Because of this high cache hit rate, update and search processes can be processed quickly, compared with a conventional parallel B-tree structure.

3. CONCURRENCY CONTROL METHODS

Some kind of concurrency control method for the B-tree is necessary to guarantee consistency. Instead of locks, fast and simple latches are usually used for concurrency control during a traversal of index nodes in a B-tree [9]. A latch is a form of semaphore, and the latch manager does not have a deadlock detection mechanism. Therefore, concurrency control for a B-tree node should be deadlock free.

3.1 Latch Modes

In this paper, a latch is assumed to have five modes: IS, IX, S, SIX, and X, as shown in Table 1 [9]. The symbol "◯" means that the two modes are compatible, i.e., two or more transactions can hold a latch at the same time.

Because parallel B-tree structures, including the Fat-Btree, have duplicated nodes, a special protocol for the distributed latch manager is required to satisfy latch semantics. Requested IS and IX mode latches can be processed only on a local PE, whereas the other modes must be granted on all the PEs storing the duplicated nodes to be latched. That is, the IS and IX modes have much smaller synchronization costs than the S, SIX, and X modes, which require communication between the PEs. The S, SIX, and X mode latches on remote copies are acquired by using their pointers. In addition, such latches must be set in linear order to avoid a deadlock (e.g., the order of logical PE number). This means the synchronization cost grows in proportion to the number

of PEs storing a copy of the node to be latched.

3.2 Concurrency Control for a Fat-Btree

As described above, concurrency control of the access path should be deadlock free. Moreover, on index nodes at upper levels of the Fat-Btree, which are replicated in many PEs, the use of S, SIX, and X mode latches that require synchronization should be avoided as much as possible.

An alternative concurrency control protocol, suggested by Mohan et al. [22], acquires an X-tree latch to protect the entire B-tree when SMOs occur. This protocol in a parallel B-tree in a distributed environment is simplified by allocating one PE to manage the tree latch. However, this PE becomes a bottleneck as the number of PEs increases. Moreover, the synchronization overhead is large, as latches on the entire tree are acquired.

In light of the preceding discussion, a good concurrency control protocol for parallel B-trees should satisfy the following conditions [21].

Condition 1. A concurrency control method for parallel B-trees should satisfy the following conditions.

(a) No concurrency control protocol method for index nodes causing deadlocks should be used.

(b) Use of S, SIX, and X mode latches on index nodes at upper levels of the B-tree should be avoided as much as possible.

(c) The entire tree should never be latched, even for short periods.

B-OPT [1], OPT-DLOCK [28], and ARIES/IM [22] are excellent concurrency control methods for a B-tree on a single machine. However, they do not satisfy Condition 1: B-OPT does not satisfy Condition 1–(b), OPT-DLOCK does not satisfy Condition 1–(a), and ARIES/IM does not satisfy Condition 1–(c). Therefore, these concurrency controls are unsuitable for parallel B-trees such as Fat-Btree.

3.3 The INC-OPT Protocol

The INC-OPT protocol satisfies Condition 1 [21].

With INC-OPT, searching for a key is simple. An IS mode latch is held on the root node initially, and then the following steps are performed during traversal of the parallel B-tree.

1. Derive a pointer to a child node by comparing the key in the parent node.
2. Acquire an IS mode latch on the child, and release the latch on the parent.
3. Repeat the above steps until the traversal reaches a leaf node.

The above procedure is usually called latch-coupling[2]. When the traversal arrives at a leaf node, it acquires an S latch on the leaf and reads data from it.

The INC-OPT protocol for an update consists of two phases.

The first phase: The B-tree is traversed using latch-coupling with IX latches. At the leaf node, an X latch is acquired. If the leaf node is not full, the updater updates it. Otherwise, if the leaf node is full, the leaf is split, the latch is released immediately, and the request shifts to the second phase.

The second phase: INC-OPT tries to acquire X mode latches on the lower two nodes, i.e., the leaf node and its parent. If the parent node must also be split, INC-OPT releases all latches and tries to acquire X mode latches on the lower three nodes. This process continues until all the nodes involved in the SMO are protected by X latches.

The INC-OPT protocol is defined precisely in [21].

When an SMO occurs, INC-OPT may require multiple restarts. When the SMO involves the root node, INC-OPT requires as many phases as the height of the B-tree. This increases the response time of update operations. In addition, it decreases overall system throughput because of the multiple X latches used on upper-level index nodes.

3.4 The MARK-OPT Protocol

The MARK-OPT protocol [31] is suitable for a parallel B-tree. It marks the lowest SMO occurrence point during latch-coupling operations. MARK-OPT improves response time by reducing the frequency of restarts. In addition, MARK-OPT produces higher system throughputs by reducing the intermediate phases of distributing X latches and removing unrequired X latches.

The procedure that MARK-OPT uses to search for a key is identical to that of INC-OPT, whereas its update consists of the following two phases.

The first phase: The traversal reaches a leaf with latch-coupling using IX latches. If an index node is not full, MARK-OPT marks the height of the node from the root node. If subsequent nodes are not full, the marked height is carried forward to the subsequent nodes. At the leaf, an X latch is acquired. If the leaf node is not full, the update occurs. If the leaf node is full, a split occurs in the leaf, this latch is released immediately, and the procedure shifts to the second phase.

The second phase: The height of the tree is marked as in the first phase. MARK-OPT tries to acquire the X mode latches on the leaf node and the index nodes below the height marked in the previous phase. If any node involved in the SMO is not protected by an X latch, it releases all latches and restarts. This process continues until all the nodes involved in the SMO are protected by X latches.

The MARK-OPT protocol is precisely defined in [31].

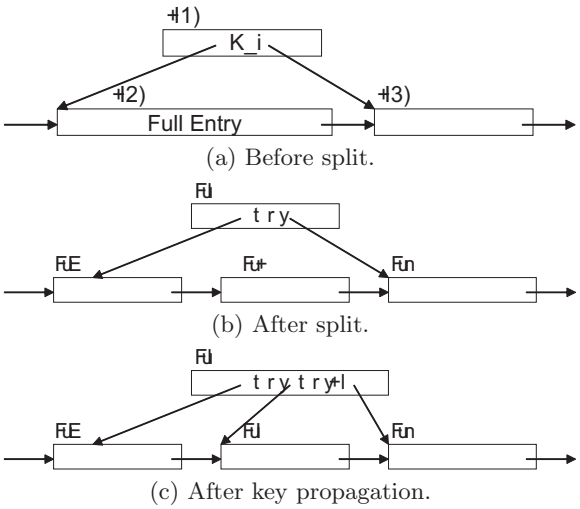
Because MARK-OPT decides the range of the X latch based on the state of the previous phase obtained by marking, it may require multiple restarts when SMOs have spread. However, the maximum number of phases in MARK-OPT is the height of the tree, as is the case for INC-OPT. MARK-OPT often requires only one restart because SMOs rarely spread. MARK-OPT does not require as many restarts as INC-OPT, even when SMOs occur on nodes at upper levels.

3.5 B^{link}-tree

A B^{link}-tree [15, 13] links all nodes at each level together. A node contains a side pointer connecting a sibling index node and its key (high key). Figure 2 shows an example

[2]Latch-coupling is called *crabbing* in [9].

Figure 2: B^{link}-tree node.



(a) Before split.

(b) After split.

(c) After key propagation.

Figure 3: A B^{link}-tree page split.

B^{link}-tree node, and Figure 3 shows an example page split. A process with a search key higher than the "high key" uses the side pointer to find the appropriate page. Moreover, in the B^{link}-tree, neither processes for searching nor processes for updating latch-couple on their way down to a leaf node.

In the B^{link}-tree, a process that is searching holds an IS mode latch on the root node initially, and then the following steps are performed during traversal of a B-tree.

1. Derive a pointer to a child node or a sibling node by comparing the key in the parent node.
2. Release the latch on the parent, and acquire an IS mode latch on the child or the sibling.
3. Repeat the above steps until the traversal reaches a leaf node.

When the traversal arrives at a leaf node, acquire an S latch on the leaf and read data from it. A process for updating traverses to a leaf node in the same way as the process for searching. However, an X latch is held in the leaf node. If the leaf is not full, the updater updates it. If the leaf is full, a split occurs in the leaf, and the updater performs the split in the leaf. After the latch on the leaf is released, SMOs are propagated to the parent. To perform SMOs, X mode latches for each page are held. At this time, the updater does not latch-couple. Therefore, readers and updaters acquire the latch only on one node at a time.

The B-link algorithm is very effective for a B-tree in a single machine. However, there is a high cost in using the B-link algorithm on an entire parallel B-tree structure. Details are given in Section 4.5.1

In [10], the balance of of the B-link tree is maintained at all times, so that a logarithmic time bound for a search or

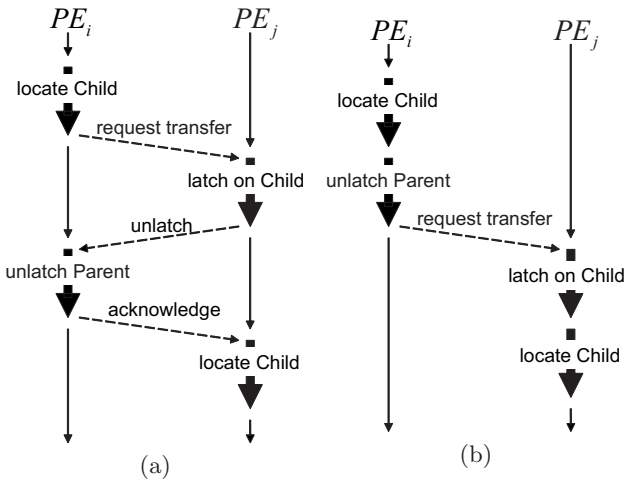


Figure 4: Request transfer protocols.

an update operation and deletions. This algorithm also does not consider parallel B-tree structures.

4. LCFB

Our new concurrency control protocol, which improves the performance of parallel B-trees, is called the *latch-coupling-free parallel B-tree* (LCFB) *concurrency control protocol*.

In traditional INC-OPT and MARK-OPT, a traversal reaches a leaf using latch-coupling with IS or IX latches. Therefore, transferring access requests to another PE requires three sequential messages per transfer (see Figure 4–(a)). First, a message ("request transfer") is sent to a destination PE to acquire a latch on the child. Next, a message ("unlatch") is sent to the source PE to release the latch on the parent. Finally, a message ("acknowledge") is sent to the destination PE to process on the child in the destination PE. On the other hand, if a traversal reaches a leaf without latch-coupling, as in LCFB, only one message per transfer is required (see Figure 4–(b)). Therefore, LCFB improves response time by reducing the frequency of network communication. Because the cost of network communication is large in traversal, LCFB can have some dramatic effects with small changes.

4.1 Access Path Error Detection

Traversal without latch-coupling may follow a pointer to an incorrect page. If this happens, the traversal restarts on the root page after the acquired latch is released.

An example of an access path error because of a split is shown in Figure 5. In Figure 5–(a), before index page (I2) splits, let process A acquire a latch on (I1). Then process A finds the next page (I2) for the key *"Key_i"* from (I1) and releases the latch on (I1) without acquiring the latch on (I2). Therefore, other processes may modify the index pages in Figure 5–(a) into the index pages in Figure 5–(b). Process A then acquires a latch on (I2) and locates (I2), so it does not follow a link to the correct page (I3), but to an incorrect page (I2).

In this case, the traversal restarts on the root page in LCFB. However, a process cannot detect an access path error from the index pages shown in Figure 5. From the information contained in (I2), it cannot determine that key

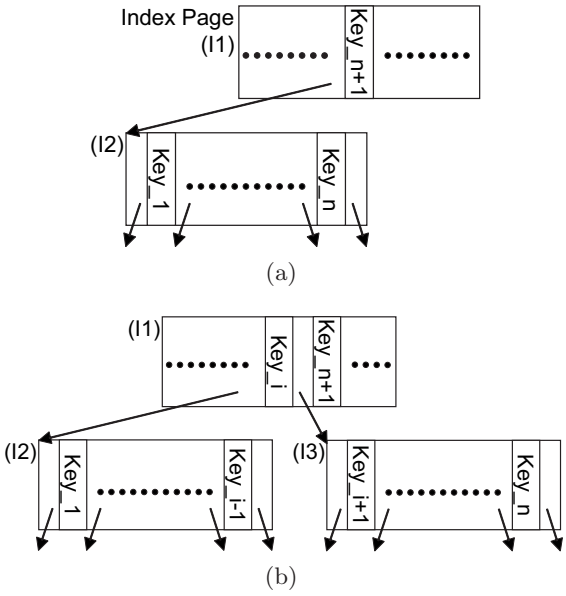


Figure 5: Splitting an index page.

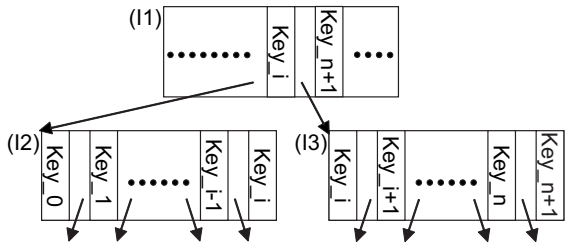


Figure 6: Data structure of index pages with boundary values.

"Key_i" is not contained in (I2), so the following problem occurs when a search key is greater than *"Key_i – 1"*. If a process determines that a search key is not contained in (I2) and restarts on the root page, it cannot traverse to a child page of (I2). On the other hand, if it determines that the key is contained in (I2) and it follows the access path, process A follows an incorrect access path.

To detect the access path error, we use index pages as shown in Figure 6. The pages have boundary values at both ends. By using the boundary values contained on the page, a process can determine whether the page is correct. In the above example, process A can find that *"Key_i"* is greater than the boundary value *"Key_i"* and can determine that page (I2) is incorrect.

A step with access path error detection gives a higher execution time in a node than traditional protocols. However, this step requires only two comparisons with the boundary values. Therefore, the additional execution time is minimal.

In the B-link algorithm [15, 13] and the ARIES/IM [22] algorithm, access path errors can also occur. In the B-link algorithm, such errors occur because neither readers nor updaters acquire the latch on only one node at a time. In this case, links chaining together all nodes at each level lead to correct access paths. In ARIES/IM, the traversal reaches a leaf with latch-coupling. However, readers or updaters can access an updated child node ((I2) in Figure 5–(b)) even before a parent node is updated because updaters acquire the X latch on only one updated node at a time. Therefore, access path errors can still occur. In this case, returning to a page in which the value of the Log Sequence Number (LSN) is not updated leads to a correct access path.

In these methods, updaters acquire X latches bottom-up for SMOs. Therefore, many access path errors occur in top-down traversals. On the other hand, in LCFB, updaters acquire X latches top-down for SMOs. Therefore, the frequency of access path errors is low compared with the B-link algorithm or ARIES/IM[3].

In the proposed method, retraversing from the root is necessary when an access path error is detected. However, a process can retraverse from a middle node by the technique using LSNs as well as ARIES/IM and that of [25]. If a request transfer happens during a traversal, this technique may bring the process back to a former PE. This is not as efficient as retraversing from the root. The technique using LSNs should therefore be used only within the same PE.

4.2 Search

The LCFB process for searching does not use latch-coupling. An IS mode latch is held on the root node initially, and then the following steps are performed during traversal of a parallel B-tree.

1. If an access path error is detected, release the latch on the parent and restart at the root node.
2. Derive a pointer to a child node by comparing keys in the parent node.
3. Release the latch on the parent, and acquire an IS mode latch on the child.
4. Repeat the above steps until the traversal reaches a leaf node.

When the traversal arrives at the leaf node, acquire an S latch on the leaf and read data from it.

4.3 Update

The LCFB process for updating consists of the following two phases.

The first phase: The traversal reaches a leaf without latch-coupling using IX latches. If an access path error is detected, release the latch on the parent and restart at the root node. At this time, the mark made on the last traversal is not used. If an index node is not full, LCFB marks the height of the node from the root node. If subsequent nodes are not full, the marked height is carried forward to the subsequent nodes. At the leaf, an X latch is acquired. If the leaf node is not full, the update occurs. If the leaf node is full, a split occurs in the leaf, this latch is released immediately, and the procedure shifts to the second phase.

The first phase: The height of the tree is marked as in the first phase. LCFB tries to acquire the X mode latches

[3]In LCFB, the frequency of access path errors is very low. Therefore, the difference in approaches to return to the correct access path will hardly affect performance. In the experiments in Section 5, the maximum frequency of access path errors was less than 0.00002.

```
1   l := H;
2   P := null; C := ROOT; h := 1; m := 1;
3   while h < l do begin
4       Unlatch P, IX latch on C;
5       if Access path error is detected then
6           Unlatch C; goto 2;
7       if C is safe then
8           m := h;              /* Marking */
9       Determine NewC;
10      P := C; C := NewChild; h := h + 1;
11  end;
12  if h < H then
13      X latch on C and its copies, Unlatch P;
14  else begin
15      Unlatch P, X latch on C;
16      if Access path error is detected then
17          Unlatch C; goto 2;
18  end;
19  while h < H do begin
20      Determine NewChild;
21      P := C; C := NewChild; h := h + 1;
22      X latch on C and its copies;
23  end;
24  if X latches are not sufficient for SMOs then
25      Release all granted latches; l := m; goto 2;
26  else begin
27      Update including SMOs;
28      Release all granted latches;
29  end;
```

Figure 7: The LCFB protocol.

on the leaf node and the index nodes below the height marked in the previous phase. If any node involved in the SMO is not protected by an X latch, it releases all latches and restarts. This process continues until all the nodes involved in the SMO are protected by X latches.

More precisely, the level of node (h) is one for the root node, and the height of the tree H for the leaf node. Let l denote the level of the B-tree at which LCFB must start using X latches. The variable l is initially set to H. The height marked during a traversal is denoted by m. The parent is denoted by P and the child is denoted by C. LCFB is shown in Figure 7.

The LCFB protocol satisfies Condition 1. The reasons are:

1. It is deadlock free because it acquires latches top-down.
2. It does not latch the index nodes with the S, SIX modes, and does not acquire needless X mode latches on nodes not relating to SMOs.
3. It never uses a tree latch.

Therefore, LCFB is a concurrency control protocol suited to parallel B-trees.

4.3.1 Delete

A page with no entries is deleted. When latch-coupling guarantees a pointer to a correct page, a pointer to a deleted page is not acquired. However, as latch-coupling is not used in LCFB, a process may attempt to access deleted pages.

To prevent accesses to deleted pages, we combine the *drain technique* [12] and a *delete_bit*. The delete_bit is usually set to "0". A page scheduled for deletion is not deleted immediately, but has its delete_bit set. With the drain technique, the actual deletion of the page is delayed until the termination of all processes that began traversal while pointers to the page still existed. Such processes can then determine an access path error based on a value of the delete_bit.

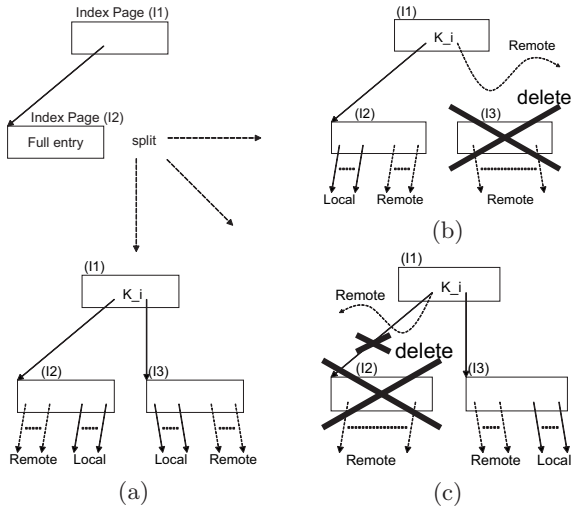


Figure 8: Splitting in the Fat-Btree.

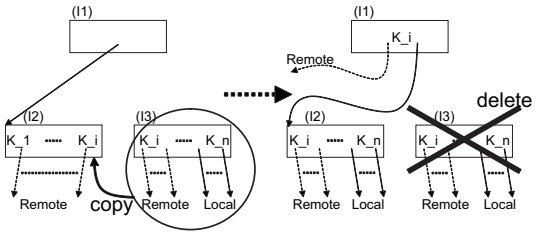


Figure 9: Swapping a deleted page and a nondeleted page.

4.3.2 Insert

With Fat-Btrees, a page may be deleted even when an index node is split by an insert operation. If an index page only has pointers to remote pages, the index page is deleted to satisfy the properties of the Fat-Btree.

Splitting in Fat-Btrees falls into three patterns (see Figure 8). In Figure 8–(a), no page is deleted. In Figure 8–(b), (I3) is deleted. However, no pointer to (I3) exists. Therefore, no process can access the deleted page. On the other hand, in Figure 8–(c), (I2) is deleted, but a pointer to (I2) remains. In this case, a process may access the deleted page.

To prevent access to pages deleted by splitting, we use the following procedure. Entries in (I3) are copied to (I2), and (I3) is deleted instead of (I2). There is no pointer to (I3) (see Figure 9). Therefore, a process may not access the deleted page.

In this case, using the delete_bit can also block an access to the deleted page. However, the method shown in Figure 9 is more effective than using the delete_bit because the method restarts only those processes that access pages not containing the key (the frequency of restarts is halved).

4.4 Correctness

Concurrency control of B-trees guarantees that all processes correctly terminate within a finite length of time. We show that LCFB guarantees this.

When an updater realizes that it did not acquire all required X latches for an SMO, the updater releases all the latches without modifying any data. Thus, LCFB essen-

tially follows the two-phase locking/latching (2PL) protocol, which ensures physical consistency of the B-tree structure for each update [2].

Traversals without latch-coupling are correctly executed when SMOs do not occur, but traversals without latch-coupling may access incorrect pages when SMOs occur. The SMOs occur due to insert requests or delete requests. Accessing an incorrect page due to an SMO for an insert request is detected by comparing a key whose process accesses an incorrect page with the boundary values stored in each page. Accessing an incorrect page due to an SMO for a delete request is detected by checking delete_bit in each page. Therefore, LCFB can detect accessing incorrect pages due to all SMOs. Moreover, because consistency of the B-tree is guaranteed, boundary values in each page show correctly the range of values in each page (the delete_bit in each page show correctly whether each page is valid) and it restarts at the root page when a process accesses an incorrect page. Therefore, LCFB does not read incorrect data in nondestination leaf pages, it does not update incorrect pages, and it does not incorrectly terminate.

LCFB is deadlock free and certainly terminates because it acquires latches top-down. Moreover, it restarts at the root page when a process accesses an incorrect page, but shifting to traversal with latch-coupling after a number of restarts can avoid the situation of livelock, or an infinite number of restarts. Denote the level of node by h, and the value that limits restarts by r. At worst, accessing $h*(r+1)$ pages can provide the acquisition of latches on the above destination pages. Because LCFB is deadlock free, it accesses $h*(r+1)$ pages within a finite length of time, and LCFB terminates within a finite length of time.

Because LCFB does not incorrectly terminate and it terminates within a finite length of time, LCFB guarantees that all processes correctly terminate within finite time.

4.5 LCFB with B-link

We combined LCFB with the B-link algorithm within each PE to reduce the cost of SMOs in a PE.

LCFB can process searches quickly. This is because LCFB reduces the cost of request transfers by not latch-coupling. However, LCFB must acquire X latches on all nodes involved in an SMO, and the cost of updates is large. B-links can reduce the frequency of X latches and the cost of SMOs. However, it is difficult to use the B-link algorithm on an entire parallel B-tree structure. By combining LCFB with B-link, the extension can utilize the advantages of the B-link, which can process both update processes and search processes effectively.

4.5.1 Application of B-link to the Fat-Btree

In the B^{link}-tree, an index node links to a sibling node. Moreover, in the Fat-Btree, an index page is deleted if the index page only has pointers to remote pages. Therefore, a link from an index page (1, 2) to an index page (3, 3) must be deleted if index pages (1, 3), (2, 3), and (3, 3) split, as in Figure 10. This deletion requires an X latch on the index page (1, 2). The index page (1, 2) does not exist on the path from the root to the leaf. The advantage is lost because the acquisition is an inefficient process.

To avoid losing the advantage of the B-link, we linked between index pages within each PE with side pointers, and we did not link between index pages in different PEs

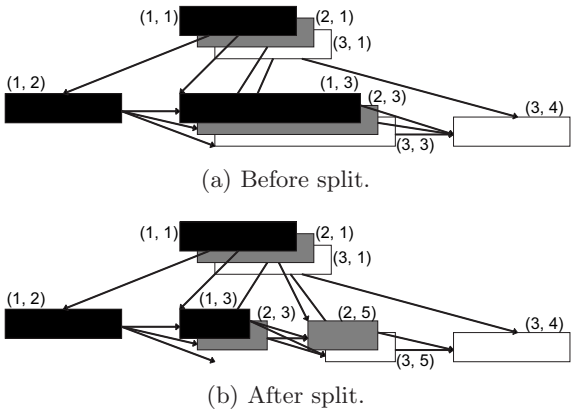


Figure 10: A page split on the Fat-Btree using B-link.

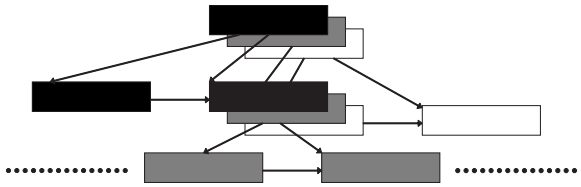


Figure 11: Fat-B^{link}-tree.

(see Figure 11). The side pointer is deleted when splitting links between index pages in different PEs (see Figure 10). Therefore, the B-link algorithm operates effectively if the side pointers link between index pages within each PE.

In the B^{link}-tree combined with LCFB, an index page has a lower boundary value (see Figure 12).

4.5.2 Switching Processing Protocols

There may be copies of index pages that do not have side pointers. The B-link algorithm cannot guarantee that all processes behave correctly if index nodes are not all linked together at each level. We therefore combine the B-link algorithm with the LCFB algorithm. LCFB performs SMOs when the nodes involved in the SMOs are protected by X latches, and it guarantees that all processes behave correctly on index pages having copies.

The cost of updates in LCFB is higher than with the B-link algorithm. However, few index pages have copies[4]. The B-link algorithm performs most SMOs because the nodes having multiple pages exist at upper levels of the B-tree[5]. Therefore, LCFB with B-link performs SMOs effectively.

4.5.3 Search

LCFB with the B-link process for searching for a key is basically identical to LCFB, as it does not use latch-coupling. IS mode latches are acquired on index pages and the S mode latches are acquired on leaf pages.

In LCFB, a traversal restarts on a root page if access path errors occur. In LCFB with the B-link algorithm, the process follows the side pointer if the side pointer can lead

[4]In the Fat-Btree for the experiments in Section 5, only about 7.0% of all index nodes had multiple copies.

[5]In the experiments in Section 5, SMOs involving nodes with multiple pages were about 0.9% of all update requests.

Figure 12: B^{link}-tree node with LCFB.

to a correct access path.

4.5.4 Update

The LCFB with the B-link algorithm process for updating consists of the following two phases.

The first phase: The traversal to a leaf is basically identical to search. However, IX mode latches are acquired on index pages and X mode latches are acquired on leaf pages. If the leaf node is not full, the updater updates it. If the leaf node is full, the updater performs bottom-up updates according to the B-link algorithm. If an SMO involves an index node having multiple pages, the procedure shifts to the second phase.

The second phase: According to LCFB, all the nodes involved in other SMOs are protected by X latches. The process then propagates to an appropriate parent and performs SMOs at levels above the parent. If an X latch is acquired on an index node having a single page, the latches on levels above the node are released immediately, and the procedure shifts to the first phase to propagate the appropriate parent instead of updating a leaf.

More precisely, the level of node (h) is one for the root node, and the height of the tree H for the leaf node. Let l denote the level of the B-tree at which LCFB must start using X latches. The variable l is initially set to H. The height marked during a traversal is denoted by m. Finally, the level of the parent into which an SMO propagates is denoted by r. The parent is denoted by P and the child is denoted by C. LCFB with the B-link algorithm is shown in Figure 13.

```
1   l := H; r := H;
2   P := null; C := ROOT; h := 1; m := 1;
3   while h < min(l, r) do begin
4       Unlatch P, IX latch on C;
5       if Access path error is detected then
6           Unlatch C; goto 2;
7       Determine NewChild;
8       P := C; C := NewChild;
9       if C is not sibling of P then begin
10          if P is safe then
11              m := h;              /* Marking */
12          push(P); h := h + 1;
13      end;
14  end;
15  if C has no copy then begin
16      Unlatch P, X latch on C;
17      while C is not target do begin
18          Determine NewChild;
19          P := C; C := NewChild;
20          Unlatch P, X latch on C;
21      end;
22      while C is unsafe do begin
23          Update including SMO on C;
24          Unlatch C;
25          C := pop(); h := h - 1;
26          X latch on C;
27          if C has its copies then
28              Unlatch C; l := m; r := h; goto 2;
29      end;
30      Update; Unlatch C;
31  end;
32  else begin
33      X latch on C and its copies, Unlatch P;
34      while h < r do begin
35          Determine NewChild;
36          P := C; C := NewChild;
37          if P is safe then
38              m := h;              /* Marking */
39          push(P); h := h + 1;
40          if C has no copy then
41              Release all granted latches; l := H; goto 3;
42          X latch on C and its copies
43      end;
44      if X latches are not sufficient for SMOs then
45          Release all granted latches; l := m; goto 2;
46      else begin
47          Update including all SMOs;
48          Release all granted latches;
49      end;
50  end;
```

Figure 13: The LCFB with B-link protocol.

4.5.5 Correctness

Because LCFB with the B-link algorithm proceeds according to just the B-link algorithm when an SMO involving an index node having only single pages occurs, the B-link algorithm processes correctly. Therefore, we need to show that the B-link algorithm processes correctly when an SMO involving an index node having multiple pages occurs.

When an SMO involving an index node having multiple pages occurs, a switch between LCFB and the B-link algorithm occurs. We show that LCFB with the B-link algorithm guarantees that all processes behave correctly, even when a switch between LCFB and the B-link algorithm occurs.

When the procedure shifts from LCFB to the B-link algorithm, all latches acquired in LCFB are released. Therefore, the shift does not cause deadlock. Because SMOs are performed until all the nodes involved by them are protected by X latches, the shift does not cause an incomplete B-tree. When the procedure shifts from the B-link algorithm to LCFB, SMOs break, and the shift may cause an incomplete B-tree. However, all processes will behave correctly because the side pointers link sibling nodes. Moreover, no index nodes are processed by both LCFB and B-link algorithms, because the procedure checks whether index pages have copies whenever latches are acquired.

Therefore, LCFB with the B-link algorithm guarantees that all processes behave correctly even when switches between LCFB and the B-link algorithm occur.

5. EXPERIMENTS

To show that the proposed protocols are effective, we used an implementation of an autonomous disk system [29] using blade systems. We used a Fat-Btree, and evaluated the performance under a number of conditions.

5.1 Experimental Environment

We used an experimental system of an autonomous disk distributed storage technology. The experimental system was implemented on a 160-node blade system using the Java programming language under Linux. We used 128 nodes for storing data and 32 nodes as clients sending requests. A preliminary experiment showed that the backbone network switch had adequate performance. The experimental envi-

Table 2: Experimental environment.

# Nodes:	4–128 (Storage), 32 (Clients)
CPU:	AMD Athlon XP-M 1800+ (1.53 GHz)
Memory:	PC2100 DDR SDRAM 1 GB
Network	1000BASE-T
Hard Drives:	TOSHIBA MK3019GAX (30 GB, 5400 rpm, 2.5 inch)
OS:	Linux 2.4.20
Java VM:	Sun J2SE SDK 1.5.0_03 Server VM

Table 3: Parameters used for the experiments.

Page size:	4 KB
Tuple size:	350 B
Max no. of entries in an index node (fanout):	64
Max no. of tuples in a leaf node:	8

ronment is summarized in Table 2.

Initial Fat-Btree Construction

We prepared a leaf node on each PE, to seed the Fat-Btree. The key of the initial leaf node was used as a lower bound on the key values of nodes stored in that PE. The initial leaf node keys were set in ascending order of PE number. Therefore, the leaf node stored in each PE by follow-on insertion was divided statically. We then repeatedly inserted random elements in the initial Fat-Btree. Table 3 shows the basic parameters we set for the experiments. These parameters were chosen to distinguish clearly the differences between the protocols. Traditional methods store only a list of keys in a node of the B-tree. On the other hand, the proposed method also stores the upper and lower boundary values in a node. Therefore, the traditional methods and the proposed methods should have different fanout. Because the difference is very small, we ignored it in this paper.

The experiments for this paper were organized as follows. First, we evaluated the performance as a function of the number of PEs storing data by counting the frequency of request transfers. Next, we evaluated the performance as a function of the update ratio and counted the frequency of X latches.

5.2 Comparison with Differing Numbers of PEs

Thirty-two clients (32 threads in parallel per blade) sent requests to the PEs containing the Fat-Btree with 5120 tuples per PE, for 10 seconds. The access frequencies were uniform, and the update ratio was fixed at 20%.

Figure 14 shows the performance of the five concurrency control protocols and the frequency of request transfers, and Figure 15 shows the frequency of sent messages per operation when the number of PEs varied from four to 128.

The throughputs of INC-OPT and MARK-OPT increased more slowly as the number of PEs increased. This is because the frequency of request transfers increases (see Figure 14). INC-OPT and MARK-OPT with latch-coupling require three network messages per request transfer. Therefore, the high frequency of request transfer increases the network communication overhead. This is also shown in Figure 15. On the other hand, the proposed protocols can transfer a request with just one network message. Therefore, the throughput of the proposed protocols increased more rapidly with the number of PEs. Moreover, the frequencies

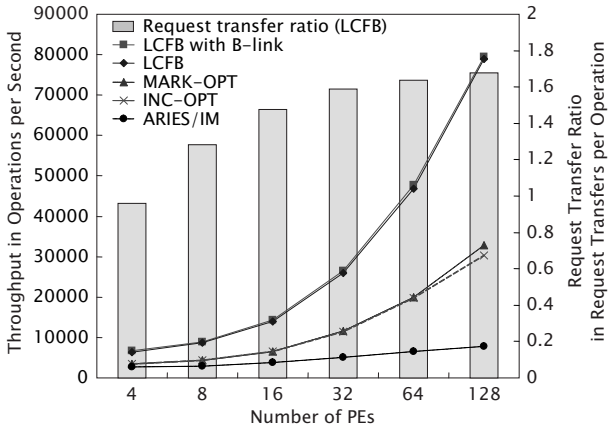


Figure 14: Comparison with differing numbers of PEs.

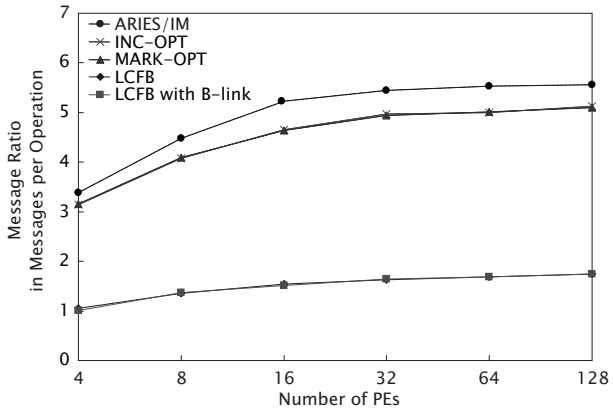


Figure 15: Comparison of message ratio with differing numbers of PEs.

of request transfers in each protocol are the same. These experimental results indicate that the proposed protocols have the highest scalability in those protocols.

5.3 Comparison with Changing Update Ratio

Thirty-two clients (32 threads in parallel per blade) sent requests to the PEs containing the Fat-Btree with 5120 tuples per PE, for 10 seconds. The access frequencies were uniform, and the number of PEs was fixed at 64.

Figure 16 shows the performance of the five concurrency control protocols, Figure 17 shows the frequency of acquisitions of X latches per operation and the frequency of restarts with access path error in LCFB as the update ratio changes from 0% to 100%.

When the update ratio is low, the proposed protocols have much better performance than INC-OPT and MARK-OPT, but the efficacy of LCFB drops as the update ratio increases. This is because LCFB and MARK-OPT are basically the same when processing the SMOs that occupy the entire large processing time. The decline in the throughput of LCFB with the B-link algorithm is much slower than that of LCFB alone. This is because LCFB with B-link reduces the frequency of acquisition of X latches (see Figure 17).

The frequency of request transfers is update independent,

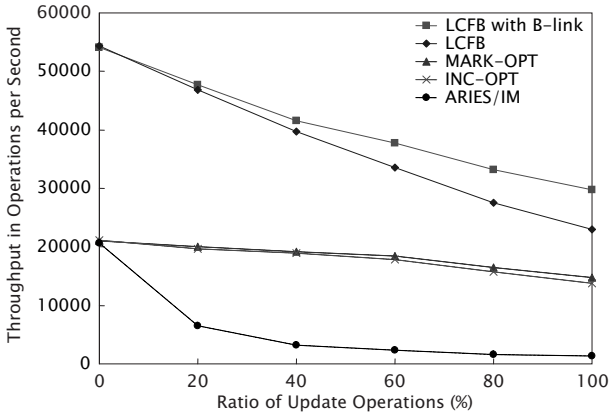


Figure 16: Comparison of concurrency control protocols with changing update ratio.

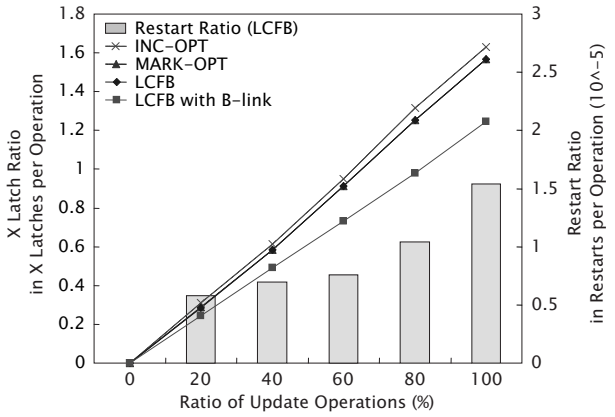


Figure 17: Comparison of X latch ratio and restart ratio with changing update ratio.

and the frequency of access path errors is highest when the update ratio is 100%. Therefore, LCFB has the greatest disadvantage when the update ratio is 100%. Experimental results also indicate that the frequency of restarts with access path errors in LCFB increases as the update ratio increases. However, the throughput of LCFB is higher than that of conventional protocols and it is low enough not to affect system performance even when the update ratio is 100%. Experiments on all update ratios includes situations in practice, and the disadvantage of access path errors never exceeds the advantage of efficient request transfers in all situations.

6. RECOVERY

Before our protocol can be incorporated into real database systems or storage systems, recovery strategies are needed. The recovery strategies must treat three types of failures: transaction failures, system failures, and media failures. For recovery from each failure, it is important to treat all transactions as atomic actions [17], to log all transactions, and to keep the logs of noncommitted transactions.

Atomic actions in distributed environments require distributed commit protocols such as the two-phase commit protocol. Moreover, we have also proposed BA-1.5PC [23], which can efficiently do logging and handle transaction failures in a distributed environment. BA-1.5PC adopts Fat-Btree and our proposed protocol as the distributed index and the concurrency control, respectively. Moreover, it is able to recover from system failures and media failures if the logs are kept. It significantly outperforms several well-known commit protocols in terms of transaction throughput.

It would be possible to adapt the techniques used in the ARIES/IM [22] for recovery from system failures. Viable recovery strategies for the B-link algorithms have also been proposed [17]. To improve data availability in the face of media failures, we can choose systems with primary-backup declustering like autonomous disk [29]. Recovery from media failures in the Fat-Btree has been discussed in [20].

While a discussion of recovery issues is beyond the scope of this paper, the point to be noted is that viable approaches do exist.

7. RELATED WORK

Graefe uses fence keys in leaf pages that are similar to our boundary keys [8]. In [8], adding techniques of log-structured file systems to traditional B-trees without adding a layer of indirection for locating B-tree nodes on disk improves write performance. By using this technique, B-tree pages migrate to new locations when they are updated. However, in the traditional B^+-tree, there exist three pointers to a leaf page (parent and two siblings). Therefore, those pages must also migrate to a new location. To solve this problem, they retain in each page a lower and upper fence key that define the range of keys that may be inserted in the future into those pages. This change decreases the number of required updates of pointers when a leaf page migrates to a new location, and decreases update costs. An important use of the fence keys is consistency checking that the correctness of a commercial database has not been corrupted by hardware or software errors. Moreover, the fence keys affect key range locking. Therefore, the fence keys are similar to our boundary keys, but their uses are different.

8. CONCLUSION

We propose a new concurrency control for Fat-Btrees suitable for shared-nothing parallel machines. To reduce the cost of request transfers in Fat-Btrees, LCFB does not use latch-coupling during optimistic operations. In the distributed environment of a shared-nothing parallel machine, the overhead of latch-coupling is high. LCFB gains a significant amount of system throughput by avoiding latch-coupling. To detect access path errors in LCFB, index pages have boundary values at both ends. Moreover, by combining the drain technique with a delete_bit, access to a deleted page by processes is blocked. In a Fat-Btree, a page split may cause page deletion. Swapping deleted pages with non-deleted pages in the protocol is more effective than normal page deletion.

To reduce the cost of SMOs in LCFB, we also combine the LCFB protocol with the B-link algorithm within each PE. The combination of LCFB and the inside B-link is an effective approach to attack the problem of difficulty of the consistency management for the side pointers in parallel B-tree structures. The B-link algorithm reduces the frequency and the range of the X latches for a single B-tree. The experimental results with changing system size indicate that the

proposed protocols are always effective, especially in large-scale configurations. The experimental results with changing update ratio indicate that LCFB with the B-link algorithm reduces the frequency of X latches and is effective at higher update ratios. Any prootocol has not existed like LCFB with high scalability.

In future years, parallel B-tree structures will become more important in contents servers such as the SAS Scalable Performance Data Server [26]. Therefore, we have found that there are significant advantages to providing efficient right concurrency control for parallel B-tree structures. We have achieved a substantial amount of scalability for the content servers by designing LCFB.

9. ACKNOWLEDGMENTS

We thank Dr. Jun Miyazaki of NAIST for his advice on concurrency control for Fat-Btrees and B-link. This work is partially supported by CREST of JST (Japan Science and Technology Agency), SRC (Storage Research Consortium), a Grant-in-Aid for Scientific Research from MEXT Japan (#16016232, #18049026), the Tokyo Institute of Technology 21COE Program "Framework for Systematization and Application of Large-Scale Knowledge Resources", and the NHK science and technical research laboratories.

10. REFERENCES

[1] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Inf.*, 9(1):1–21, 1977.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of OSDI 2006*, pages 205–218, 2006.

[4] Cluster File Systems, Inc. Lustre: A scalable, high-performance file system. `http://www.lustre.org/docs/whitepaper.pdf`.

[5] R. Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Proc of FODO '93*, pages 101–114, 1993.

[6] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of SOSP 2003*, pages 29–43, 2003.

[8] G. Graefe. Write-optimized B-trees. In *Proc. of VLDB 2004*, pages 672–683, 2004.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[10] I. Jaluta, S. Sippu, and E. Soisalon-Soininen. Concurrency control and recovery for balanced B-link trees. 14(2):257–277, 2005.

[11] T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *Proc. of PODS '89*, pages 235–246, 1989.

[12] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.

[13] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *Proc. of FJCC '86*, pages 380–389, 1986.

[14] M. Lee, Mong-Liand Kitsuregawa, B. C. Ooi, K.-L. Tan, and A. Mondal. Towards self-tuning data placement in parallel database systems. In *Proc. of the SIGMOD '00*, pages 225–236, 2000.

[15] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[16] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* — linear hashing for distributed files. In *Proc. of the SIGMOD '93*, pages 327–336, 1993.

[17] D. Lomet and B. Salzberg. Concurrency and recovery for index trees. *VLDB Journal*, 6(3):224–240, 1997.

[18] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of OSDI 2004*, pages 105–120, 2004.

[19] J. Menon, D. A. Pease, R. Rees, L. D. ich, and B. Hillsberg. IBM storage tank — a heterogeneous scalable SAN file system. *IBM Syst. J.*, 42(2):250–267, 2003.

[20] J. Miyazaki, Y. Abe, and H. Yokota. Availabilities and costs of reliable Fat-Btrees. In *Proc. of PRDC 2004*, pages 163–172, 2004.

[21] J. Miyazaki and H. Yokota. Concurrency control and performance evaluation of parallel B-tree structures. *IEICE Transactions on Information and Systems*, E85-D(8):1269–1283, 2002.

[22] C. Mohan and F. Levine. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In *Proc. of the SIGMOD '92*, pages 371–381, 1992.

[23] X. Ouyang, T. Yoshihara, and H. Yokota. An efficient commit protocol exploiting primary-backup placement in a distributed storage system. In *Proc. of PRDC 2006*, pages 238–247, 2006.

[24] B. Phillips. Have storage area networks come of age? *IEEE Computer*, 31(7):10–12, 1998.

[25] N. Ponnekanti and H. Kodavalla. Online index rebuild. In *Proc. of the SIGMOD '00*, pages 529–538, 2000.

[26] SAS. SAS scalable performance data server. `http://www.sas.com/technologies/dw/storage/spds/index.html`.

[27] B. Seeger and P.-Å. Larson. Multi-disk B-trees. In *Proc. of the SIGMOD '91*, pages 436–445, 1991.

[28] V. Srinivasan and M. J. Carey. Performance of B-tree concurrency control algorithms. In *Proc. of the SIGMOD '91*, pages 416–425, 1991.

[29] H. Yokota. Autonomous disks for advanced database applications. In *Proc. of DANTE '99*, pages 435–442, 1999.

[30] H. Yokota, Y. Kanemasa, and J. Miyazaki. Fat-Btree: An update-conscious parallel directory structure. In *Proc. of ICDE '99*, pages 448–457, 1999.

[31] T. Yoshihara, D. Kobayashi, and H. Yokota. MARK-OPT: A concurrency control protocol for parallel B-tree structures to reduce the cost of SMOs. *IEICE Transactions on Information and Systems*, E90-D(8):1213–1224, 2007.