Concurrence Control Protocol for Parallel B-Tree Structures That Improves the Efficiency of Request Transfers and SMOs within a Node

Tomohiro YOSHIHARA†,‡‡†, Dai KOBAYASHI†,‡, Nonmembers, and Haruo YOKOTA†††, Fellow

SUMMARY Many concurrency control protocols for B-trees use latch-coupling because its execution is efficient on a single machine. Some studies have indicated that latch-coupling may involve a performance bottleneck when using multicore processors in a shared-everything environment, but no studies have considered the possible performance bottleneck caused by sending messages between processing elements (PEs) in shared-nothing environments. We propose two new concurrency control protocols, “LCFB” and “LCFB-link”, which require no latch-coupling in optimistic processes. The LCFB-link also innovates B-link approach within each PE to reduce the cost of modifications in the PE, as a solution to the difficulty of consistency management for the side pointers in a parallel B-tree. The B-link algorithm is well known as a protocol without latch-coupling, but B-link has the difficulty of guaranteeing the consistency of the side pointers in a parallel B-tree. Experimental results in various environments indicated that the system throughput of the proposed protocols was always superior to those of the conventional protocols, particularly in large-scale configurations, and using LCFB-link was effective for higher update ratios. In addition, to mitigate access skew, data should migrate between PEs. We have demonstrated that our protocols always improve the system throughput and are effective as concurrency controls for data migration.

key words: concurrency, distributed databases, transaction processing, directory structures, trees

1. Introduction

Many industries and research groups aim to utilize the results of massive data computation for critical business decisions. Analysis of the results is crucial in many ways, to improve service quality and to support novel features. Because the scale of the data volumes for these purposes can be very large, traditional centralized database systems may not be efficient. As a result, some industries and research groups have developed distributed data storage and computing systems on large clusters of shared-nothing commodity servers. Examples include Google’s MapReduce [1], Hadoop [2] from the open-source community, which is used at Yahoo, and Cosmos/Dryad [3], [4], which is used at Microsoft. These can achieve high parallelism by data partitioning and processing each partition concurrently using multiple servers. They scale well for massive data and have efficient methods for addressing load balancing, recovery from failure, and other issues. However, to achieve high performance, users are required to translate their application to the MapReduce model, and the translation can be difficult for some applications. This may involve certain limitations that may impact developer productivity and optimization opportunities.

However, parallel B-tree structures have few complications for users, in simplifying the management of data in distributed servers. B-tree structures [5] were called “ubiquitous” more than a quarter of a century ago [6], and they have since become even more widespread. Gray and Reuter asserted that “B-trees are by far the most important access-path structures in database and file systems” [7]. Despite many innovative proposals and prototypes for alternatives to the B-tree, this statement remains true today.

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 such as the Fat-Btree [8], but also an efficient concurrency control protocol for the parallel B-tree or Fat-Btree. Examples are the INC-OPT [9] and MARK-OPT protocols [10], which are suited to parallel B-trees on shared-nothing parallel machines. These protocols outperform conventional B-tree concurrency control protocols such as the B-OPT [11] and ARIES/IM protocols [12]. However, the performance of these protocols does not scale well with system size, because they may involve extensive high-cost network communication. To address this, with multicore processors and flash storage now being widely used, host of latch-free concurrency control protocols for shared-everything environments have been proposed for multicore processors or flash storage [13]-[27]. These protocols mainly aim to be latch-free in a multicore-processor environment.

The latch-free approach reduces cache pollution in each core in a multicore processor environment. The Foster B-tree [13] combines the advantages of B-link trees, symmetric fence keys, and write-optimized B-trees [28]. OLFIT [15], Mass-tree [16], and Silo [17] use the technique...
of checking the version of pages before and after reading pages instead of latching for page reads. [14] proposed the Master-tree for a rich NVRAM environment, which combines the Foster B-tree and Silo. [18]–[22] use read-copy-updates (RCU) and a compare-and-swap (CAS) technique. [23], [24] use Intel Transactional Synchronization Extensions. PALM [25] uses the bulk synchronous parallel (BSP) technique. PLP [26] and ATraPos [27] achieve their latch-free status by assigning processor cores to each partition in a value-range partition. Although multicore processors and rich memory hierarchies with flash storage and non-volatile memory can improve the performance with a single node dramatically, network communication has yet to be improved significantly. Therefore, another approach is needed for a high-performance system scaled in terms of many nodes.

We have proposed a new concurrency control protocol called the latch-coupling-free parallel B-tree (LCFB) concurrency control protocol, which is suited to Fat-Btrees. We published the basic ideas for LCFB in [29]. In this paper, we develop the LCFB, describe the algorithm for data migration, and evaluate the algorithm. It was found to reduce the cost of network communication compared with traditional protocols. To detect access-path errors in the LCFB protocol caused by the removal of latch-coupling, we assign boundary values to each index page. The fence keys for the Foster B-tree [13] are similar to our boundary keys, but their uses and effects are different. (Sect. 8 describes these differences.)

In addition, we propose LCFB-link, which reduces the cost of structure-modification operations (SMOs). LCFB-link is a combination of LCFB with the B-link algorithm. We improve the efficiency of request transfers by LCFB, and aim to derive the benefits of other efficient algorithms for SMOs. We also adopt the B-link algorithm. It had been difficult to apply B-link to a parallel B-tree structure because it is necessary to guarantee the consistency of the side pointers. The LCFB-link combination is a solution to the difficulty of consistency management for the side pointers in a parallel B-tree structure. B-link takes advantage of the premise of one-on-one pages, where SMOs do not delete pointers from a parent to a child. However, this premise may not hold for parallel B-tree structures. LCFB-link can improve the efficiency of both request transfers and SMOs within a node, because the B-link algorithm can be selectively applied. This is more difficult for other algorithms.

We implemented the proposed protocols and those for MARK-OPT, INC-OPT, and ARIES/IM

We implemented ARIES/IM based on [12], but omitted its recovery mechanism because we do not focus on recovery in this paper.

1

date ratios. The B-link algorithm is best known as a protocol without latch-coupling, but it is difficult to use the B-link algorithm on an entire parallel B-tree structure directly. Therefore, we address on the cost of latch-coupling in distributed environments, and propose a protocol without latch-coupling that uses B-link.

We focus on systems containing many processing elements (PEs) for very large data. Systems with many PEs face skewed access to data at each PE. To mitigate this skew, data should migrate between PEs. We compared the throughput of LCFB and LCFB-link with that for MARK-OPT, INC-OPT, and ARIES/IM while changing the frequency of data migration. The experimental results indicate that LCFB and LCFB-link are also effective when data migration occurs.

The concurrency control methods for update operations and data migration with parallel B-trees are important not only for parallel database systems, but also for parallel data storage in general. The number of unstructured data files can often increase rapidly. To manage them efficiently, they should be stored in a parallel data store that has a global name space. To realize efficient management functions, including skew handling for a large number of globally named files, an effective parallel B-tree with sophisticated concurrency control is required.

The remainder of this paper is organized as follows. First, the background to methods for partitioning data among PEs, parallel B-tree structures, and the concept of the Fat-Btree structure are reviewed, with data migration being introduced in Sect. 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 Sects. 4 and 5. Experimental results are reported in Sect. 6. Section 7 presents a discussion of recovery in parallel B-trees. We review related work in Sect. 8. The final section presents the conclusions of this paper.

2. Background

There are several ways to partition a large amount of data among PEs, including value–range, round robin, and hashing [31]. 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 nearby values. However, it can degenerate the load distribution because it can potentially skew the data distribution. Even though the initial data allocation may have no skew, repeated updates may destroy the balance because partitioning criteria are determined statically. Additionally, round robin partitioning can avoid skew. However, all PEs must participate in every query for strict or range matches because there may be no information about value locations.

In the remainder of this section, we explain hash partitioning and distributed directory structures, and we indicate the advantages of systems with the distributed directory structures that we are assuming. In addition, we intro-
duce the parallel B-tree structure and the Fat-Btree, which are particularly efficient for handling large amounts of data via distributed directory structures.

2.1 Hash Partitioning

Hash partitioning can support 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 [32], [33]. However, as described above, hashing cannot handle range queries or I/O clustering. In hashing, the dynamic data migration for handling access skew among PEs may also have a high cost.

2.2 Advantages of Systems with Distributed Directory Structures

The main advantages of systems with distributed directory structures in share-nothing environments are as follows. They have less severe performance bottlenecks than approaches that use a highly redundant centralized server. They offer not only simpler management to administrators, but also higher throughput to users. This is because they provide efficient access by handling the high skew associated with access by many users in a distributed environment. Finally, they reduce complexity by providing data placements that facilitate dynamic management. Therefore, they support the heavy demands of current applications.

2.3 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 [34]. Parallel B-trees are useful because they can manage strict match and range queries, clustering I/O operations, and the balancing of data for each PE via the index nodes of a B-tree. They can also perform dynamic data migration locally, which will have lower cost.

2.3.1 SMOs

SMOs involve page-split and page-merge operations. 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 only page merges when pages become empty. This follows nodes not being required to fill beyond being half full because this was not found to decrease occupancy significantly for practical workloads [35]. Because the consistency of the B-tree must be guaranteed when SMOs occur, concurrency control for the B-tree is necessary. Concurrency control is also very important because it largely influences the system throughput when SMOs occur.

2.3.2 Request Transfers

Sometimes, an access request cannot be served from the current PE because 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 rapid access transfers, the 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 interactions influencing system performance. It is important to reduce the number of interactions for high scalability because the number of interactions will increase as the number of PEs increases.

2.4 Partitioned B-Tree Structures

A partitioned B-tree comprises 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 indexes independently the data at its PE. However, the partitioned B-tree structure will not be height balanced because the number of records at each PE can be different, and differing heights for the PEs can degrade the overall performance.

An aB⁺-tree [36] is another two-tier index structure designed to maintain the global height-balanced property of indexes for 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. All the B⁺-trees across all PEs are then of the same height. However, balancing the heights comes at a cost.

2.5 The Fat-Btree Structure

A Fat-Btree [8] 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 an index node increases with the node’s 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.

Moreover, for Fat-Btrees, index pages are only required for locating the leaf pages stored in each PE. Therefore, Fat-Btrees will 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 to those for conventional parallel B-tree structures.

The main advantages of using a Fat-Btree as a distributed directory structure are as follows. It utilizes the advantages of parallel B-trees, which manage strict match and range queries and clustering I/O operations efficiently. It manages the balancing of data and the access to data in each PE more efficiently than other parallel B-trees. Finally, it reduces the cost of balancing the height of other parallel B-trees.

2.6 Data Migration with Fat-Btrees

Data migration is effective in handling the skew in an access-request distribution. For range-partitioned data placement, the total access frequency of data stored in each PE is balanced eventually by migrating data between PEs logically adjacent in the range partition. The outline of the algorithm for migrating nodes is described in [8]. Figure 2 gives an example of data migration in a Fat-Btree, where the rightmost leaf’s index page with data pages in a PE are migrated to its right-side PE. In this case, because only two consecutive PEs are involved in the data migration, the data migration can be achieved without blocking the processes in the other PEs.

The tree structure of a Fat-Btree is unchanged by data migration. However, the migration of a data page causes an update in its parent index node and, occasionally, in more remote ancestor nodes, by recursive updating. If the parent node already exists in the destination PE, only an update of the corresponding entry is required. Otherwise, the parent node must be moved to the destination PE. If the source PE contains other children of the parent node, the source PE must retain a copy of the node. Otherwise, the node in the source PE must be removed.

3. Concurrency Control Methods

In this section, we describe “latches” and the latch modes our proposed methods use for concurrency control. We then describe the main existing concurrency control methods and their shortcomings. The ways in which our proposed methods address these shortcomings are described in Sect. 4.

3.1 Latches

Some kind of concurrency control method for the B-tree is necessary to guarantee consistency. Instead of using locks, fast and simple latches are usually used for concurrency control during a traversal of the index nodes in the B-tree [7]. 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.

In this paper, a latch is assumed to have five modes: IS, IX, S, SIX, and X, as shown in Table 1 [7]. 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, such as ordering by logical PE number. This means that the synchronization cost will grow in proportion to the number of PEs storing a copy of the node to be latched.

3.2 The MARK-OPT Protocol

The MARK-OPT protocol [10] is suitable for parallel B-trees. It marks the lowest SMO occurrence point during latch-coupling operations and improves the response time.

<table>
<thead>
<tr>
<th>Mode</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IX</td>
<td>○</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>○</td>
<td></td>
<td>○</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIX</td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1 Latch matrix.
by reducing the frequency of restarts. In addition, MARK-OPT produces a higher system throughput by reducing the number of intermediate phases for distributing X latches and by removing unrequired X latches.

With MARK-OPT, searching for a key is simple. An IS mode latch is held on the root node initially, and then the following steps 1–3 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". When the traversal arrives at a leaf node, it acquires an S latch on the leaf and reads data from it.

The MARK-OPT protocol for an update has 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. Otherwise, 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 defined in detail in [10].

3.3 \textbf{B}^{\text{link}}-\textbf{T}ree

A B^{\text{link}}-tree [37], [38] links together all nodes at each level. A node contains a side pointer connecting a sibling index node and its key (the "high key"). A process with a search key higher than the high key uses the side pointer to find the appropriate page. Moreover, in the B^{\text{link}}-tree, neither the processes for searching nor the processes for updating involve latch-coupling on their way down to a leaf node.

In a B^{\text{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 the traversal of the 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, it acquires an S latch on the leaf and reads data from it. The 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. Otherwise, a split occurs in the leaf, performed by the updater. 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 invoke latch-coupling. Therefore, readers and updaters acquire a latch for only 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 associated with using the B-link algorithm on an entire parallel B-tree structure. Details are given in Sect. 5.1.

In [39], the balance of the B-link tree is maintained at all times, leading to logarithmic time bounds for search, update, and deletion operations. However, this algorithm does not consider parallel B-tree structures.

3.4 Problems with Existing Methods

MARK-OPT outperforms conventional B-tree concurrency control protocols such as the B-OPT [11] and ARIES/IM protocols [12]. However, MARK-OPT requires three sequential messages for any redirection between PEs. Because MARK-OPT involves this extensive high-cost network communication, the performance of MARK-OPT does not scale well with system size. The cost derives from the latch-coupling method of MARK-OPT. We describe the details of this cost, together with its resolution by the proposed methods, in Sect. 4.1.

The B-link algorithm provides for tree traversal without latch-coupling. It also takes advantage of the premise of one-on-one pages, for which SMOs do not delete pointers from a parent to a child. However, it is difficult for a simple implementation of B-link on parallel B-trees such as the Fat-Btree to maintain the side links. B-link requires a solution for parallel B-trees. We describe the details of this problem, together with the approach taken by the proposed methods, in Sect. 5.1.

4. \textbf{LCFB}

In this section and in Sect. 5, we present LCFB and LCFB-link as our new concurrency control protocols, which aim to improve the performance of parallel B-trees.

First, LCFB without latch-coupling reduces the cost of network communication in comparison with traditional protocols. This is, LCFB provides a solution to the cost issue associated with MARK-OPT. To detect access-path errors in the LCFB protocol caused by the removal of latch-coupling, we assign boundary values to each index page.
Second, LCFB-link is a combination of LCFB with the B-link algorithm. The LCFB-link combination provides a solution to the difficulty of consistency management for the side pointers in a parallel B-tree structure. LCFB-link reduces the cost of SMOs and traversals. LCFB-link can improve the efficiency of both request transfers and SMOs within a node, because the B-link algorithm can partially produce its effect.

4.1 Traversal without Latch-Coupling

In traditional MARK-OPT, a traversal reaches a leaf using latch-coupling with IS or IX latches. Therefore, the transfer of access requests to another PE requires three sequential messages per transfer (see Fig. 3 (a)). First, a message (“request transfer”) is sent to a destination PE (PEj) to acquire a latch on the child. Next, a message (“unlatch”) is sent to the source PE (PEi) to release the latch on the parent. Finally, a message (“acknowledge”) is sent to the destination PE (PEj) to process the child in the destination PE. However, if a traversal reaches a leaf without latch-coupling, as in LCFB, only one message per transfer is required (see Fig. 3 (b)). Therefore, LCFB improves the response time by reducing the frequency of network communication. Because the cost of network communication is large for traversal, LCFB can obtain some dramatic effects from relatively small changes.

4.2 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 occurring because of a split is shown in Fig. 4. In Fig. 4 (a), before index page (I2) splits, assume process A acquires a latch on (I1). Process A then finds the next page (I2) for the key “Keyi” from (I1) and releases the latch on (I1) without acquiring the latch on (I2). Therefore, other processes may modify the index pages shown in Fig. 4 (a) into the index pages shown in Fig. 4 (b). Process A then acquires a latch on (I2) and locates (I2), causing it to follow a link to the incorrect page (I2) rather than the correct page (I3).

In this case, with LCFB, the traversal restarts on the root page. However, a process cannot detect an access path error from the index pages shown in Fig. 4. From the information contained in (I2), it cannot determine that key “Keyi” is not contained in (I2), so a problem occurs whenever a search key is greater than “Keyi – 1”. The problem is that 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 will follow an incorrect access path.

To detect such access-path errors, we use the index pages shown in Fig. 5. 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 “Keyi” is greater than the boundary value “Keyi – 1” and can determine that page (I2) is incorrect.

Using this access-path error-detection step will involve additional processing in the node, compared with traditional protocols. However, the step requires only two comparisons with the boundary values. Therefore, the additional execution time will be minimal.

In the B-link [37], [38] and ARIES/IM [12] algorithms, 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, the links that chain 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 Fig. 4(b)) even before the 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 for which the value of the log sequence number (LSN) has not been updated will lead to a correct access path.

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

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 LSN-based technique of ARIES/IM or that of [40]. 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 LSN-based technique should therefore be used only within the same PE.

4.3 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 1–4 are performed during traversal of the 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, it acquires an S latch on the leaf and reads data from it.

4.4 Update

The LCFB process for updating comprises 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. Otherwise, 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. LCFB 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.

4.4.1 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, because 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 [41] with the use of 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 identify any access-path errors based on the value of the delete_bit.

4.4.2 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.

A Fat-Btree split involves one of three patterns (see Fig. 6). In Fig. 6(a), no page is deleted. In Fig. 6(b), (I3) is deleted. However, no pointer to (I3) exists. Therefore, no process can access the deleted page. Conversely, in Fig. 6(c), (I2) is deleted, but a pointer to (I2) remains. In this case, a process may try to access the deleted page.

To prevent access to pages deleted by splitting, we use

---

† In LCFB, the frequency of access-path errors is very low. Therefore, any difference in approach to returning to the correct access path will affect the performance only minimally. For the experiments described in Sect. 6, the maximum frequency of access-path errors was less than 0.00002.
the following procedure. Entries in (I3) are copied to (I2), and (I3) is deleted instead of (I2). There is no pointer to (I3) (see Fig. 7). Therefore, a process may not access the deleted page.

In this case, using the delete bit can also block access to the deleted page. However, the method shown in Fig. 7 is more effective than using a delete bit because the method restarts only those processes that access pages not containing the key (the frequency of restarts is halved).

4.5 Data Migration

It is easy to apply LCFB to concurrency control for data migration. We need only combine updating on LCFB with data migration for MARK-OPT, except for deletion of unused pages.

The data migration protocol for LCFB is similar to the update protocol for LCFB with respect to acquiring and releasing latches. However, it is necessary to involve two kinds of marking for LCFB as in MARK-OPT [10]. There is no problem with making new necessary pages for data migration in LCFB. This is because, with no latch-coupling, a target leaf page can be found via a copy page if it cannot access a newly-made page. This does have the potential for an increase in request transfers. However, it rarely occurs, and therefore has little effect on performance. In contrast, there is an issue with trying to access a deleted page when deleting unnecessary pages, again because of no latch-coupling. LCFB resolves the problem by combining the drain technique with a delete bit, similarly to the process of deleting pages. This guarantees a correct path for the target leaf page during data migration.

4.6 Correctness

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

When an updater realizes that it has not acquired all required X latches for an SMO, the updater releases all the latches without modifying any data. Therefore, LCFB essentially follows the two-phase locking/latching protocol, which ensures physical consistency of the B-tree structure for each update [42].

Traversals without latch-coupling are correctly executed when SMOs do not occur, but traversals without latch-coupling may access incorrect pages when SMOs do occur. These SMOs occur because of insert or delete requests. Accessing an incorrect page because of 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 because of an SMO for a delete request is detected by checking the delete bit in each page. Therefore, LCFB can detect accessing incorrect pages for 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 shows correctly if the page is valid) and a restart occurs at the root page whenever 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 is guaranteed to terminate because it acquires the latches top-down. Moreover, it restarts at the root page whenever a process accesses an incorrect page. Changing to traversal with latch-coupling after a certain number of restarts can avoid the situation of livelock (an infinite number of restarts). To show this, denote the level of a node by $h$, and the value that limits restarts by $r$. At worst, accessing $h*(r+1)$ pages can provide the ac-
acquisition of latches on the above destination pages. Because LCFB is deadlock free, it accesses \( h \ast (r + 1) \) pages within a finite length of time, and LCFB will therefore terminate within a finite length of time.

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

5. LCFB-Link

We propose a new concurrency control protocol “LCFB-link”, which involves combining LCFB with the B-link algorithm within each PE to reduce the cost of SMOs in the PE.

LCFB can perform searches quickly. This is because LCFB reduces the cost of request transfers by avoiding 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 therefore 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 combined system can utilize the advantages of the B-link, which can process both update processes and search processes efficiently.

5.1 Application of B-Link to the Fat-Btree

In B\textsuperscript{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 shown in Fig. 8. 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. Any advantage is lost because the acquisition is an inefficient process.

To avoid losing the advantage of the B-link, our system links index pages within each PE with side pointers, and does not link index pages in different PEs (see Fig. 9). For example, index pages \((1, 3), (2, 3)\) do not link to index page \((3, 4)\) in Fig. 9. The side pointer is deleted when splitting links to index pages in different PEs (see Fig. 8). Therefore, the B-link algorithm operates effectively if the side pointers link index pages only within each PE.

In B\textsuperscript{link}-tree combined with LCFB, an index page has a lower boundary value (see Fig. 10).

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. Therefore, we 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 that have copies.

The cost of updates in LCFB is higher than with the B-link algorithm. However, few index pages have copies\(^\dagger\). The B-link algorithm performs most SMOs because nodes with multiple pages exist at upper levels of the B-tree\(^\ddagger\). Therefore, LCFB-link performs SMOs effectively.

5.3 Search

The LCFB-link process for searching for a key is very similar to that of 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 the LCFB-link algorithm, the process follows the side pointer if the side pointer can lead to a correct access path.

![Fig. 8](image1.png) A page split on the Fat-Btree using B-link.

![Fig. 9](image2.png) Fat-B\textsuperscript{link}-tree.

![Fig. 10](image3.png) B\textsuperscript{link}-tree node with LCFB.

\(^\dagger\)In the Fat-Btree for the experiments in Sect. 6, only about 7.0% of all index nodes had multiple copies.

\(^\ddagger\)In the experiments in Sect. 6, SMOs involving nodes with multiple pages were involved in about 0.9% of all update requests.
5.4 Update

The LCFB-link algorithm process for updating comprises the following two phases:

The first phase: The traversal to a leaf is very similar to a 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. Otherwise, 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 1 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 \( C \). The LCFB-link algorithm is shown in Fig. 11.

5.5 Data Migration

In LCFB-link, achieving consistency of side pointers in data migration comes at a considerable cost, and efficient data migration requires this problem to be solved. To address this problem, we introduce “dummy pages”. We update the dummy pages instead of searching and updating sibling pages of migrating pages, which reduces the cost of data migration and simplifies the concurrency control protocol with respect to data migration.

Figure 12 shows a Fat-Btree with B-link which has dummy pages. The dashes show the dummy pages. The tree has two-way links between PEs. For example, in Fig. 12, an index page (dummy1) is the dummy page between index pages (2, 1) and (3, 1), and the tree has two-way links between the index page (dummy1) and the index page (3, 1). To update these links for migration, our method has to acquire X latches on the pages at each side of this link. Therefore, there could be deadlocks. To resolve this potential deadlock problem, we specify an order of priority for latching among the PEs. This ordering does not make migration less efficient, if the migration controller for skew handling invokes migration from a left PE to another right PE and migration from the right PE to the left PE at the same time.

5.5.1 Updating Side Pointers for Data Migration

In 5.5.1, we describe in detail each pattern of data migration. It is difficult to update the side pointers for making new necessary pages and deleting unnecessary pages for data migration using B-link. Side pointers in B-link are one-way links. Therefore, we can describe the migration to a PE logically...
First, we describe the process of migration to a “right” PE. It has to delete an unnecessary page on migration to the right PE, as shown in Fig. 13. It has to delete the side pointer from a “left” page (2, 2) to a page (2, 3) close to the dashed line in the figure. However, this left page does not exist on a path from the root page to a target leaf page, and a left page is not connected via the side pointer from the target leaf page. Therefore, there will be a high cost in updating this page.

To solve this problem, we set dummy pages to the right of the rightmost pages on each level in each PE. If a rightmost page has to be deleted during migration, the dummy page is deleted instead of the rightmost page. The rightmost page is then set as a dummy page. This method can effectively update side pointers for migration because it does not update pages outside of the traversal path. We set a dummy bit for each of the dummy pages. If the dummy bit for a page is 1, the page is a dummy page. A dummy page is similar to a normal page apart from its dummy bit being 1.

In addition, if a necessary page for migration to a right PE is required, it is necessary to make a side pointer to a right page (3, 4) from a page (3, 3) close to the dashed line shown in Fig. 14. However, it is difficult to access the left page because the left page does not exist between the root page and a target leaf page. It is very costly to make side pointers in this case.

To resolve this problem, we make a side pointer from a leftmost page in a logical right PE to a rightmost page in a logical left PE. This rightmost page is a dummy page, being present only to resolve the first problem. Therefore, this is a side pointer to a dummy page. This side pointer can remove the need for traversal via another path.

We now describe the migration to a “left” PE. If it has to make a necessary page for migration to a left PE, it is necessary to make a side pointer from a left page (2, 2) to a new page (2, 3) close to the dashed line shown in Fig. 15. However, it is difficult to access the left page because the left page does not exist between the root page and a target leaf page. It is very costly to make side pointers in this case.

To resolve this problem, we make a side pointer from a leftmost page in a logical right PE to a rightmost page in a logical left PE. This rightmost page is a dummy page, being present only to resolve the first problem. Therefore, this is a side pointer to a dummy page. This side pointer can remove the need for traversal via another path.

5.5.2 Data Migration with Dummy Pages

We can describe data migration with the above solution in terms of the example shown in Figs. 16 and 17. In these figures, the left numbers in the top-right box on each page indicate the dummy bits and the right numbers indicate the delete bit. Dashed boxes indicate dummy pages. Pages close to the chained line indicate the copied pages for each
Figure 16 shows an example of migrating a leaf node having keys “c” and “d” from a left PE (gray boxes in Fig. 16) to a right PE (white boxes in Fig. 16), and replicating a parent index page having keys “a” and “c” in the left PE to the right PE. After the traversal reaches a rightmost leaf on a source PE, the following three steps are performed. Item numbers indicate steps in the state transition in Fig. 16.

1. At the rightmost leaf page on the source PE and the dummy page on the right side of this leaf page, X latches are acquired. A copy of this leaf page is made on the destination PE. At the copied page, pointers from side to side are created. At the leftmost leaf page on the destination PE, an X latch is acquired, the side pointer to the left side is deleted, and the X latch is released. At the dummy leaf page, the delete bit is set to 1. At the copied leaf page on the source PE, the dummy bit is set to 1. At the dummy leaf page and the copied leaf page on the source PE, the X latch is released.

2. At the parent page and the dummy page on the right side of this leaf page, X latches are acquired. The entries of this leaf page on the source PE are copied to the dummy page. On the destination PE, a new dummy page is created. At the old dummy page, the dummy bit is set to 0. As for migration to the right PE, the side pointers are updated, as indicated by the upper-right boxes in Fig. 16.

3. The parent index page at the upper level has a copy on another PE. Therefore, traversal restarts from the root page. At the parent page and the copy of the page, X latches are acquired. These pages are updated, and the X latches are released for these pages.

Figure 17 shows an example of migrating a leaf node having keys “c” and “d” from a right PE (white boxes in Fig. 17) to a left PE (gray boxes in Fig. 17), and replicating a parent index page having keys “c” and “e” in the right PE to the left PE. After the traversal reaches a leftmost leaf on a source PE, the following steps are performed. Item numbers indicate steps in the state transition in Fig. 17.

1. At the dummy leaf page on the destination PE and the leftmost page on the source PE, X latches are acquired. The entries of this leaf page on the source PE are copied to the dummy page. On the destination PE, a new dummy page is created. At the old dummy page, the dummy bit is set to 0. As for migration to the right PE, the side pointers are updated, as indicated by the upper-right boxes in Fig. 17.

2. X latches are acquired at the parent page and the dummy page linked by the side pointer on the parent page. In the example, there is no index page holding a pointer to the migrated leaf page on the destination PE.
Therefore, the entries of this leaf page are copied to the dummy page as for the leaf page, and a new dummy page is created. The side pointers are then updated, as indicated by the lower left boxes in Fig. 17.

3. The parent index page at the upper level has a copy on another PE. Therefore, traversal restarts from the root page. X latches are acquired at the parent page and the copy of the page. These pages are updated, and the X latches are released for these pages.

5.6 Correctness

Because the LCFB-link algorithm proceeds according to the B-link algorithm alone when an SMO involving an index node that has only single pages occurs, the B-link algorithm processes correctly in this case. Therefore, we need only to show that the B-link algorithm processes correctly when there is an SMO involving an index node that has multiple pages.

For this case, a switch between LCFB and the B-link algorithm occurs. We can show that the LCFB-link algorithm guarantees that all processes behave correctly, even when such a switch 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 involving 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 if index pages have copies whenever latches are acquired.

Therefore, the LCFB-link algorithm guarantees that all processes behave correctly even when a switch between LCFB and the B-link algorithm occurs.

6. Experiments

To show that the proposed protocols are effective, we used an implementation of an autonomous disk system [30] using blade computers. We compared the performance of the proposed protocols LCFB and LCFB-link with the conventional protocols MARK-OPT, INC-OPT, and ARIES/IM. We used a Fat-Btree and evaluated the performance under a number of conditions.

We used an experimental setup for an autonomous-disk distributed storage technology. The experimental system was implemented on 160-node blade hardware using
Table 2 Experimental environment and parameters used for the experiments.

(a)

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

(b)

| 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 |

the Java programming language under Linux. We used 128 nodes for storing data and 32 nodes as clients to send requests. A preliminary experiment showed that the backbone network switch had adequate performance. The experimental environment is summarized in Table 2 (a). Table 2 (b) shows the parameter values set for the experiments. The construction of the initial Fat-Btree was as described in detail in [10].

The experiments described in 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-latch use. Finally, we evaluated the performance with data migration.

6.1 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 18 shows the performance of the five concurrency control protocols and the frequency of request transfers for LCFB, and Fig. 19 shows the frequency of sent messages per operation when the number of PEs is varied from 4 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 Fig. 18). INC-OPT and MARK-OPT with latch-coupling require three network messages per request transfer. Therefore, the high frequency of request transfers increases the network communication overhead. This is also shown in Fig. 19. Conversely, 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 of request transfers in each protocol are the same. These experimental results indicate that the proposed protocols have the highest scalability among the compared protocols.

6.2 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 20 shows the performance of the five concurrency control protocols, Fig. 21 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 high, LCFB offers a slight improvement on conventional protocols. This is because LCFB and MARK-OPT behave similarly when the processing of SMOs occupies the entire large processing time. Even so, LCFB improves throughput times, requiring only 56% of that for MARK-OPT. Conversely, LCFB has performs very much better than the conventional protocols when the update ratio is low. This is because processing request transfers occupies much of the processing time when searching. LCFB improves the throughput by 158% above that of MARK-OPT when the update ratio is 0%.

LCFB-link offers further improvements over LCFB when the update ratio is high. This is because LCFB-link reduces the frequency of acquisition of X latches (see Fig. 21) on pages that have no copies because of the way the B-link algorithm functions for such pages.

The frequency of request transfers is update-independent, and the frequency of access-path errors is highest when the update ratio is 100%. Therefore, LCFB is at its greatest disadvantage when the update ratio is 100%. Experimental results confirm that the frequency of restarts with access-path errors in LCFB increases as the update ratio increases (see Fig. 21). 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 for all update ratios included practical situations, and the disadvantage of access-path errors never exceeded the advantage of efficient request transfers in all such situations.

6.3 Comparison with a Changing Ratio of Data Migration

To change the frequency of data migration we changed the number of parallel processes involved in migrating the data. Each process migrated the data on different PEs. To migrate repeatedly 100 leaf nodes between two adjacent PEs, the clients sent data-migration requests at intervals of 20 milliseconds.

Sixteen clients (64 threads in parallel per blade) sent requests to 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 40%. Figure 22 shows the performance of the five concurrency controls when the number of processes migrating in parallel changed from 0 through 16. The horizontal and vertical axes are the number of processes migrating in parallel and the throughput respectively. In addition, Figure 23 shows the frequency of latch
collisions per operation for the four concurrency controls.

ARIES/IM requires considerable time to perform data migration. This is because the data migration is sure to cause an SMO. Moreover, the decline in the throughput of MARK-OPT is much less than that of INC-OPT as the frequency of data migration increases. This is because MARK-OPT reduces latch collisions by reducing X latches, as shown in Fig. 23. The decline in the throughput of LCFB is much less than that of MARK-OPT. The cost of migration in MARK-OPT and LCFB are the same because data migrations do not occur for request transfers. This is because LCFB can reduce latch collisions by reducing request transfers for searches and updates, as shown in Fig. 23.

LCFB-link can provide a higher throughput than LCFB. This is because of the efficiency of updates in B-link. Moreover, when the frequency of data migrations is high, LCFB-link offers greater improvements than for low-frequency cases. This is because LCFB-link suppresses the rising incidence of latch collisions, even when the frequency of data migration increases.

7. 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, namely transaction, system, and media failures. For recovery from each type of failure, it is important to treat all transactions as atomic actions [43], 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 [42]. We have also proposed BA-1.5PC [44], which can perform logging efficiently and handle transaction failures in a distributed environment. BA-1.5PC adopts the 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 logs are maintained. In terms of transaction throughput, it outperforms several well-known commit protocols significantly.

It would be possible to adapt the techniques used in ARIES/IM [12] for recovery from system failures. Viable recovery strategies for the B-link algorithms have also been proposed [43]. To improve data availability in the face of media failures, we can choose systems with primary-backup declustering such as the autonomous disk [30]. Recovery from media failures in the Fat-Btree has also been discussed in [45].

Although the discussion of recovery issues is beyond the scope of this paper, it should be noted that viable approaches do exist.

8. Related Work

In this section, we discuss recent concurrency control protocols. Because many of these controls focus on shared-everything environments on multicore processors, they do not consider network communication. Moreover, although multicore processors and rich memory hierarchies with flash storage and nonvolatile memory can improve dramatically the performance of a single node, network communication technology has shown little improvement recently. Therefore, few of these protocols focus on network communication costs. For a high-performance system scaled to many nodes, an alternative approach, such as the LCFB proposed in this paper, is needed to decrease network communication costs. In the remainder of this section, we provide a brief description of related work, which we review from this viewpoint.

Graefe uses fence keys in leaf pages that are similar to our boundary keys [28]. In [28], adding techniques for 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. The solution involves retaining a lower and upper fence key in each page to define the range of keys that may be inserted into those pages in the future. 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 for correctness in a commercial database, which might be corrupted by hardware or software errors. Moreover, the fence keys affect key range locking. In summary, the fence keys are similar to our boundary keys, but their uses are different.

The use of multicore processors and flash storage has spread rapidly, and a host of latch-free concurrency control
protocols in shared-everything environments have been proposed for multicore processors or flash storage [13–27]. These protocols mainly aim to be latch-free in multicore processor environments. The latch-free approach reduces cache pollution for each core in a multicore processor environment.

Graefe et al. use symmetric fence keys for continuous and comprehensive self-testing in Foster B-trees [13], which combines the advantages of fence keys, write-optimized B-trees, and Blink-trees. In Foster B-trees, the use of the fence keys is similar to our use of boundary keys, but their assumed inconsistencies, their use, and their effects are different. In Foster B-trees, unexpected inconsistency can be caused by software or hardware failures. Because of this, Foster B-trees check the state of the B-tree by comparing the fence keys of each child node to the keys of the parent node. In LCFB, inconsistency is assumed in the performance-related reduction of latching between nodes, and this inconsistency is a normal state occurring in SMOs. To address this, LCFB checks the state by comparing the boundary keys of a leaf node to a traversal target key. Moreover, Foster B-trees make partial use of techniques in Blink-trees only in an intermediate state. Efficient partial use of Blink-trees is similar to our approach. Because timelike partial use in Foster B-trees is different from the spacelike partial use in LCFB, these two methods are different.

OLFIT [15] is an optimistic, B-tree latch-free protocol for multiprocessor systems. OLFIT uses the technique of checking the version of pages before and after reading them instead of using latches for page reading. The OLFIT scheme completely eliminates latching during the index traversal. Even inserts do not incur any latching operation until the traversal reaches the insertion node. Mass-tree [16] is a latch-free protocol for in-memory systems on symmetric multiprocessor machines. Mass-tree combines B-trees and tries, with the concurrency control of Mass-tree being based on OLFIT. Silo [17] has an improved performance for in-memory databases on multicore machines by aiming to avoid conflicts while transactions acquire a transaction ID. To achieve this, Silo performs commits using an epoch number. It uses a concurrency control for tree operations that is based on Mass-tree. FOEDUS [14] was proposed for several thousand processors and a rich DRAM and NVRAM environment. FOEDUS proposed using Master-tree as both the index and the concurrency control for databases. Master-tree combines the Foster B-tree and Silo, with the concurrency control in Master-tree being based on Silo.

In [18–22], RCU and CAS techniques are used. KISS-Tree [18] uses a latch-free in-memory index that is optimized for a minimum number of memory accesses. KISS-Tree achieves latch-free status by using an RCU technique. When KISS-tree copies data from an old page to a new page, it merges the update data on the new page, and switches from a pointer to the old page to a pointer to the new page via the CAS technique. It compares a pointer to the current old page and a pointer to the old page before the new page is created. Bw-Tree [22] is an efficient B-tree for a multicore-CPU environment. When Bw-Tree creates a new page that includes an update operation and a pointer to an old page, it switches from a pointer to the old page to a pointer to this new page via the CAS technique. It compares the pointer to the current old page and the pointer to the old page before the new page is created. In [19, 20] CAS-based latch-free hashes for multicore CPUs were proposed. In [21], a latch-free transaction manager for reducing latch conflicts in the lock manager for multicore CPUs was proposed.

In [23, 24], Intel Transactional Synchronization Extensions (TSX) were used to achieve latch-free status in multicore-CPU environments. TSX are extensions to the x86 instruction set architecture that add support for hardware transactional memory.

PALM [25] is a scalable latch-free concurrency control for batch processes. In PALM, all batch operations traverse leaf pages, after which all needed SMOs are executed by the BSP technique. The BSP technique achieves latch-free SMOs.

PLP [26] achieves latch-free status by assigning processor cores to each partition in a value–range partition. PLP uses a multiroot B-tree for the value–range partition. Each core in the partition can avoid latches because only one core accesses a page in each partition. ATraPos [27], which is based on PLP, is a storage manager design that is aware of the nonuniform access latencies of multisocket systems. To reduce the latches for system states in PLP, AtraPos divides the system state into system states for each CPU socket. AtraPos also supports dynamic repartition for a multiroot B-tree based on the usage of CPU cores.

As described above, these protocols for shared-everything environments cannot consider network communication. For a high-performance system scaled to many nodes, an alternative approach, such as the LCFB proposed in this paper, is needed to decrease network communication costs.

9. Conclusion

We propose a new concurrency control for Fat-Btrees that is 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 significantly increases system throughput by avoiding such 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 combine the LCFB protocol with the B-link algorithm (LCFB-link) within each PE. The combination of LCFB and the internal B-link is an effective approach to the problem of difficult
consistency management of 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. Our experimental results for a variety of system sizes indicate that the proposed protocols are always effective, especially in large-scale configurations. The results for a variety of update ratios indicate that the LCFB-link algorithm reduces the frequency of X latches and is effective at higher update ratios. No other protocol for high scalability like LCFB has existed.

In addition, we evaluated the performance of the proposed protocols with respect to data migration. We have demonstrated that the proposed protocols improve the system throughput for data migration. Therefore, the proposed protocols are effective as concurrency controls for data migration. (Data migration should be executed based on load evaluation, although clients sent requests for data migration in this paper.) An autonomous disk system can execute these operations independently. Therefore, it will be necessary to experiment with the data-migration function.

In the future, parallel B-tree structures will become more important for content servers such as the SAS Scalable Performance Data Server [46]. Therefore, there will be significant advantages in providing efficient, scalable concurrency control for parallel B-tree structures. We can offer a substantial amount of scalability for such content servers via the LCFB design.

Acknowledgments

We thank Dr. Jun Miyazaki of Tokyo Institute of Technology 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 (Japan Science and Technology Agency), SRC (Storage Research Consortium), and the NHK science and technical research laboratories.

References


Tomohiro Yoshihara received his B.E. and M.E. degrees from Tokyo Institute of Technology, in 2005 and 2007, respectively. He is currently working in the Research & Development Group at Hitachi, Ltd. He is engaged in research on data engineering and storage systems.

Dai Kobayashi received his B.E., M.E., and Ph.D. degrees from Tokyo Institute of Technology in 2003, 2005, and 2008, respectively. He was engaged in research on data engineering and storage systems while he was with Tokyo Institute of Technology.

Haruo Yokota received his B.E., M.E., and Dr.Eng. degrees from Tokyo Institute of Technology in 1980, 1982, and 1991, respectively. He joined Fujitsu Ltd. in 1982, and was a researcher at ICOT for the 5th Generation Computer Project from 1982 to 1986, and at Fujitsu Laboratories Ltd. from 1986 to 1992. From 1992 to 1998, he was an associate professor at the Japan Advanced Institute of Science and Technology (JAIST). He moved to Tokyo Institute of Technology in 1998, where he is currently a full professor in the Department of Computer Science. His research interests include the general research areas of data engineering, information storage systems, and dependable computing. He has been a vice-president of DBSJ, chair of the ACM SIGMOD Japan Chapter, a trustee board member of IPSJ, the editor-in-chief of the Journal of Information Processing, and an associate editor of the VLDB Journal. He is currently a board member of the DBSJ, a fellow of the IEICE and the IPSJ, a senior member of the IEEE, and a member of IFIP-WG10.4, JSAI, ACM, and ACM-SIGMOD.