PAPER

Fast Rebuilding B+-Trees for Index Recovery

Ig-hoon LEE[†a)], ***Member***, **Junho SHIM**[††], ***and*** **Sang-goo LEE**[†], ***Nonmembers***

SUMMARY Rebuilding an index is an essential step for database recovery. Fast recovery of the index is a necessary condition for fast database recovery. The B+-Tree is the most popular index structure in database systems. In this paper, we present a fast B+-Tree rebuilding algorithm called Max-PL. The main idea of Max-PL is that at first it constructs a B+-Tree index structure using the pre-stored max keys of the leaf nodes, and then inserts the keys and data pointers into the index. The algorithm employs a pipelining mechanism for reading the data records and inserting the keys into the index. It also exploits parallelisms in several phases to boost the overall performance. We analyze the time complexity and space requirement of the algorithm, and perform the experimental study to compare its performance to other B+-Trees rebuilding algorithms; Sequential Insertion and Batch-Construction. The results show that our algorithm runs on average at least 670% faster than Sequential Insertion and 200% faster than Batch-Construction.

key words: *B+-Trees, index rebuilding, index reconstruction, index recovery*

1. Introduction

The goal of a recovery manager in a database system is to ensure the consistency of the database against the failure of the system. Following a crash, the recovery manager performs several tasks not only to bring up the data itself online but also to restore the indices that were built on the data. Since no transactions may be processed until the recovery process finishes, how to speed up the recovery process has been an interesting subject in the database community. Especially, in a main memory database system under which environment a huge number of transactions are processed, halting transaction processing even during the recovery activities should be minimized [3], [5], and therefore, a fast restoring of the indices as a part of database recovery is demanded [2], [10]. Generally, in order to perform the recovery, bookkeeping activities such as performing backups or checkpoints are processed during the normal operation of the system. The data stored at bookkeeping activities can be later used for making a faster recovery process [8].

The B+-Tree is the most widely-used index structure in database systems. It is, however, rather slow to construct an entire B+-Tree by sequentially inserting keys into the index, since it incurs the node splits and level propagations. Under the case that we can bulk-load all the keys into the index, the batch-constructions of B+-Tree have been used in many commercial database systems [11], and a particular approach is presented in [7]. However, those batch construction mechanisms are limited in a sense that they must perform sorting all the keys before inserting them into the index, which makes it hard to utilize the parallelism.

In this paper we present a fast B+-Tree rebuilding algorithm called Max-PL. The main idea of Max-PL is as follows:

(1) To construct a B+-Tree index structure by using the pre-stored information: Our algorithm requires the checkpoint operation to store the max keys of each leaf node. At recovery it reads the pre-stored max-key values and builds the index structure. After rebuilding the index structure, then it inserts the keys from the data records and corresponding data pointers into the index.

(2) To employ the parallelisms in several phases: In order to speed up the rebuilding process, the algorithm employs parallelisms in such phases as i) pipelining to read the data records and insert the keys into the index, ii) inserting the keys of a data record parallel to those of other records using the page-level index locking, and iii) sorting the leaf nodes in parallel to each others.

In the following subsection, we present an illustrative example to overview how our algorithm works, mainly focusing on the first idea. We will discuss the details in later sections.

Illustrative Steps of Our Algorithm

Suppose that at the time of rebuilding a B+-Tree index we have a list of max keys of leaf nodes, <3, 5, 7, 10, 12>, found at the checkpoint block. Assuming that the fanout of the B+-Tree is 4, i.e., every internal node except the root of the B+-Tree should have at least 2 keys, we may construct an index structure of the B+-Tree as shown in Fig. 1 (a), by reading those max key values. Once we have built the index structure we are not changing it but filling the leaf nodes.

Then we begin to read data blocks, and insert the keys of the records into the B+-Tree. Figure 1 (b) illustrates the snapshot of B+-Tree after processing the first four data blocks. Note that reading a data block and inserting the keys found at the block may be pipelined. The key values appearing in the leaf nodes are not in order. Finally we perform the sorting for each leaf node in parallel (Fig. 1 (c)).

Readers may find that this example is naïve in that the

Manuscript received May 27, 2005.
Manuscript revised October 24, 2005.

[†]The authors are with the School of Computer Science & Engineering, Seoul National University, Seoul 151–742, Korea.
[††]Corresponding Author. The author is with the Department of Computer Science, Sookmyung Women's University, Seoul 140–742, Korea.
a) E-mail: ihlee@europa.snu.ac.kr

DOI: 10.1093/ietisy/e89–d.7.2223

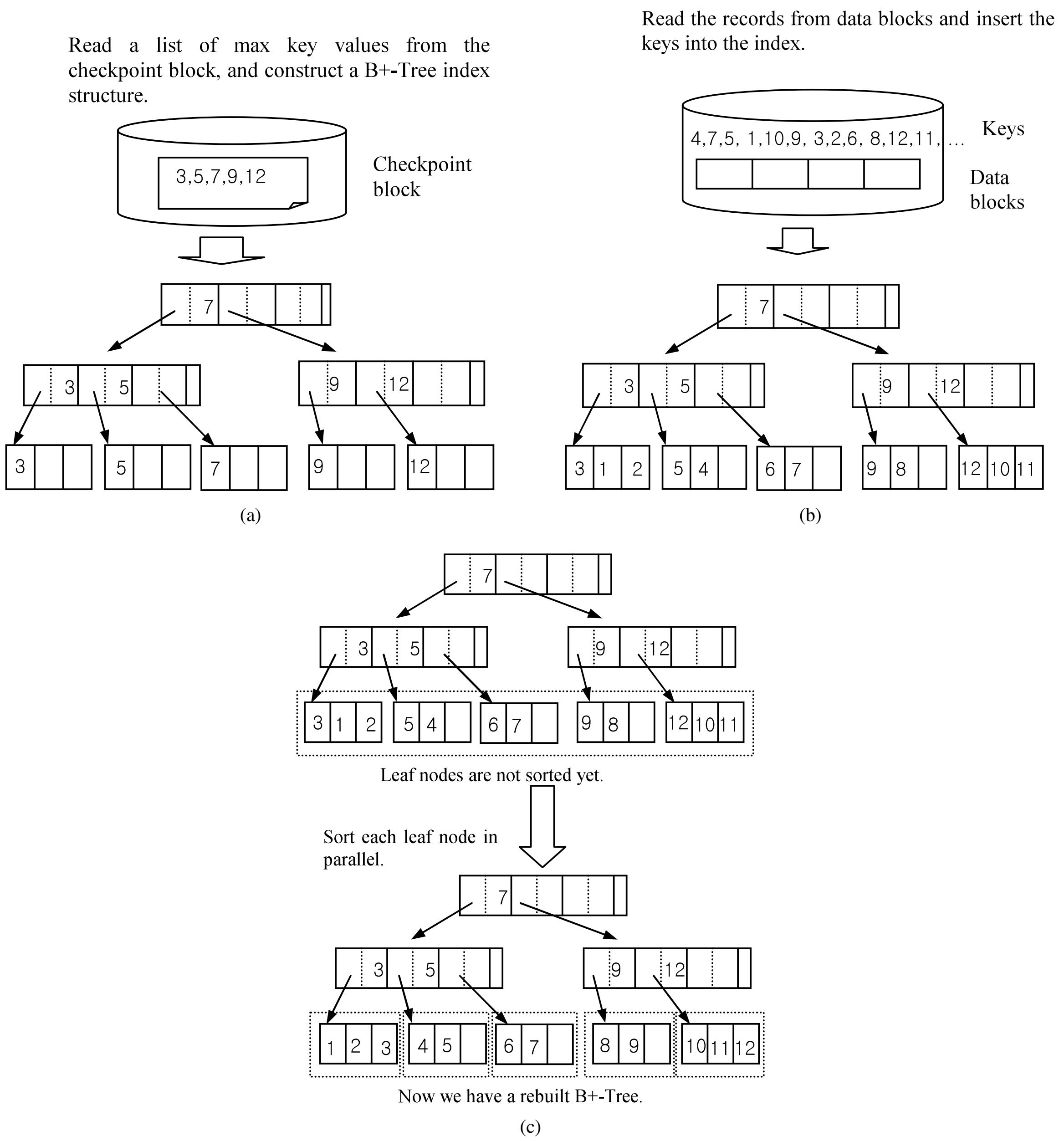

Read a list of max key values from the checkpoint block, and construct a B+-Tree index structure.
3,5,7,9,12
Checkpoint block
(a)
Read the records from data blocks and insert the keys into the index.
4,7,5, 1,10,9, 3,2,6, 8,12,11, ...
Keys
Data blocks
(b)
Leaf nodes are not sorted yet.
Sort each leaf node in parallel.
Now we have a rebuilt B+-Tree.
(c)

Fig. 1 (a) Constructing an index structure using the pre-stored max key values. (b) Inserting keys and data pointers into the index. (c) Sorting the keys of the leaf nodes in parallel.

example neither considers the detailed issues of a B+-Tree construction such as how to distribute the keys between the internal nodes with regard to the fit-ratio, nor the detail algorithms in terms of parallelism. We will discuss each detail in later sections.

The rest of this paper is organized as follows. Section 2 provides the background and overview some of the related work. Section 3 illustrates our B+-Tree rebuilding algorithm, and Sect. 4 provides the complexity analysis of the algorithm. Section 5 presents an experimental study. Finally, Sect. 6 provides the conclusion.

2. Background and Related Work

2.1 B+-Trees

The internal nodes of a B+-Tree of fanout (or *order*) p is of the form $(P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q)$ where $q \leq p$, K_i is a key such that $K_1 < K_2 < \ldots < K_q$, and P_i is the tree pointer such that for all search key values X in the subtree pointed by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_i < X$ for $i = q$. The leaf nodes of a B+-Tree of fanout p is of the form $(P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q)$ where for all $1 \leq i < q$, P_i represents a data pointer to the record of which search key value is K_i, while P_q points to

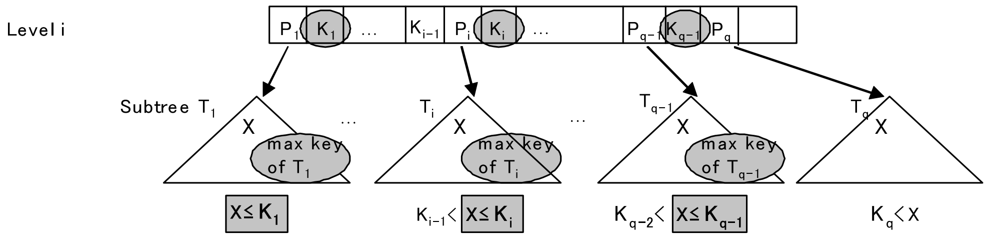


Fig. 2 A B+-Tree index node structure.

the next leaf node of the B+-Tree. Same as to the internal node structure, all search key values are sorted, i.e., $K_1 < K_2 < \ldots < K_q$. The *filling ratio*, q/p, of each node but the root should be equal or greater than $\lceil p/2 \rceil$.

Intuitively, one easy way to build $<P_i, K_i>$ values of an internal node is to have K_i as the maximum search key value of the subtree pointed by P_i (Fig. 2). This assumes that when we build a node at level i, all of its children nodes at level $i + 1$ have been already built. This is feasible in the Max-PL algorithm, since we know the maximum search key values of every leaf node, and the tree is rebuilt level by level from the leaf level to the root.

Several practical DBMS employs a so-called key compression in order to maximize the fanout value in a given node size (Fig. 3). Maximizing the fan-out is important, since the height of the B+-Tree is proportional to $\log_{fanout}$ and the number of disk I/Os to retrieve a node tends to proportional to the height. The key compression is a commonly used tact to achieve this goal [11]. During the comparison at an internal node with a given search key X, we identify two index entries with search key values K_{i-1}, K_i such that $K_{i-1} < X \leq K_i$, and then follow the path pointed by P_i. This does not mean that either node pointed by P_{i-1} or by P_i must have K_{i-1} or K_i appear in its search key values. Within a key compression, K_i value may be set as the minimal difference between the max key value of the subtree pointed by P_i and the min key value of the subtree pointed by P_{i+1}. This means that we need to know both max and min key values of the leaf nodes in order to support a B+-Tree reconstruction with the key compression in bottom-up fashion. Note that Max-PL uses only the maximum key values. However, if we pre-stored the min as well as the max key values of the leaf nodes, we can extend the Max-PL algorithm to contain the key compression method.

B+-Trees are popularly used in many practical systems, such as Oracle, Microsoft SQL Server, or PostgreSQL on DBMS, or JFS [4], XFS [13] on Linux. The proposed Max-PL can be used to rebuild B+-Tree indices in those DBMSs or Linux file systems. Moreover, since many modern DBMSs and Linux file systems support multi-processor or multi-threading capabilities, the Max-PL algorithm may benefit better parallelism to improve the performance.

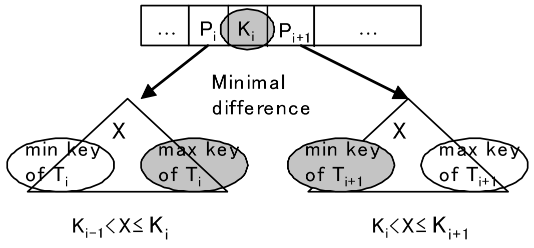


Fig. 3 The min and max values for the key compression.

2.2 Reconstruction of B+-Trees

2.2.1 Sequential Insertion

A brute-force reconstruction method of a B+-Tree is to insert keys one-by-one to the tree using the basic B+-Tree insertion algorithm. Each key insertion requires a search from the root node to leaf nodes for the position of the given key. Each insertion requires moving keys within the leaf node which the given key is inserted into. Furthermore, if the leaf node is full, the node needs splitting and the split process may be propagated upwards. Moving keys and node split are serious performance degradation overheads. However, insertions to a B+-Tree index can be carried out in parallel to the loading (or reconstruction) of data records.

We call this basic B+-Tree insertion algorithm as Sequential Insertion throughout the paper.

2.2.2 Batch-Construction

Using characteristics of the B+-Tree that the leaf nodes are on the same level and every node except the root node must be full to a certain filling ratio, it is possible to create a B+-Tree in a batch [6], [7], [11]. Unlike Sequential Insertion, the leaf nodes are built first and then the internal nodes, so there are no node splits and moving keys within a node. Figure 4 illustrates an example of performing Batch-Construction.

The problem with Batch-Construction is that all the data records must be reconstructed and the key values sorted before the index construction can start. While time can be saved from eliminating node splits and moving keys, the cost for sorting can be high and parallel processing is limited.

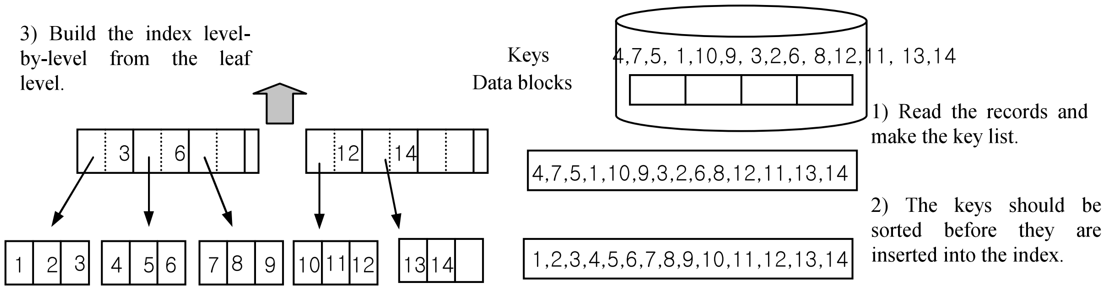

3) Build the index level-by-level from the leaf level.
Keys
Data blocks
4,7,5, 1,10,9, 3,2,6, 8,12,11, 13,14
1) Read the records and make the key list.
4,7,5,1,10,9,3,2,6,8,12,11,13,14
2) The keys should be sorted before they are inserted into the index.
1,2,3,4,5,6,7,8,9,10,11,12,13,14

Fig. 4 Example of performing Batch-Construction [7].

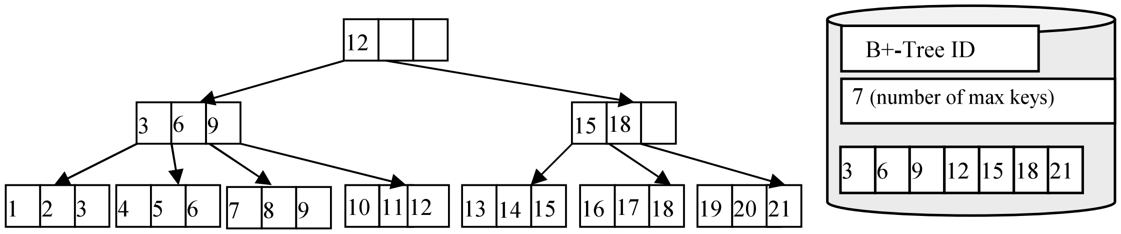


Fig. 5 Example of backing up the max key values of a B+-Tree.

3. Algorithms

The key idea of our algorithm is to use the max key values of the leaf nodes. Therefore, in order to make it run properly, we need bookkeeping activities to save the max key values of the leaf nodes during the normal operation of the system. These activities may be performed at checkpoints.

3.1 Checkpoint Requirements

In addition to its normal operations, the checkpoint needs to perform the operations for saving the max key values of the B+-Tree index. We may use an existing checkpoint algorithm such as [12] or [9] to include this activity. Nevertheless we provide a simple algorithm to perform it in Fig. 6. As the algorithm says, the information we need to store at checkpoints are the B+-Tree identifier, number of max keys, and the list of max keys. Figure 5 illustrates an example to show which information needs storing at checkpoints for a given B+-Tree.

As we illustrate in Sect. 4.2, the space required to store the max keys in a checkpoint block is less than 0.38% of the whole index size, which may be negligent in practice. Each leaf node of a B+-Tree maintains the number of key values stored in the node and a pointer to the next leaf node. Therefore performing the algorithm shown in Fig. 6 may not incur a significant system overhead.

3.2 A Basic Algorithm for Fast Rebuilding B+-Trees

In this section, we present a basic version of our algorithm for rebuilding a B+-Tree. We call it MAX in this section. The Max-PL algorithm that is presented in the next section is an extension of MAX with respect to the parallelism.

```
Procedure checkpointforBackupIndex() {
    NumLeafBlocks = get the number of leaf blocks
    Write NumLeafBlocks into checkpointblock.
    i = 0;
    do {
        node = getNextLeafNode();
        max = getMaxKey(node);
        Write max into checkpoint block;
    } while (i < NumLeafBlocks)
}
```

Fig. 6 A checkpoint algorithm to include the max key backup operations.

```
MAX()
  Max_constructIndexStructure();
    // 1) Build a B+-Tree index structure
  Max_insertKey2Index();
    // 2) Read the data records and insert keys into the index.
  Max_sortLeafNodes();
    // 3) Sort the keys in every leaf node.
end
```

Fig. 7 A pseudo code of the MAX B+-Tree rebuilding algorithm.

The pseudo code of MAX is shown is Fig. 7. Basically, it performs the following three steps:
(1) Build a B+-Tree index structure.
(2) Read the data records and insert keys into the index.
(3) Sort the keys in every leaf node.

In the following subsections, we discuss each step in more detail.

3.2.1 Build a B+-Tree Index Structure

The algorithm for building a B+-Tree index structure is shown in Fig. 8. The process is more complicated than its

```
Max_constructIndexStructure()
1:  read numMaxKeys from the checkpoint block;
2:  create an array of nodes childNodes[numMaxKeys];
3:  create an array of key list keyList[numMaxKeys];
4:  // first step - create leaf nodes
5:  for (i=0; i< numMaxKeys; i++) {
6:    read a pre-stored max_key from the checkpoint block;
7:    create a leaf node;
8:    link the leaf node to childNodes[i-1] as the next leaf node
9:    childNodes[i] = leaf node;
10:   keylist[i] = max_key;   // add key into the keylist [i] for the next
                                 level;
11: }
12: // second step - create internal nodes level by level
13: numNextKeys=0;  // initialize the # of keys for the next level
14: while( numMaxKeys > fanout ) {
15:   i=0;  // the position of key (within keylist) to be inserted next
16:   remainder = numMaxKeys mod (fanout);
17:   // when the # of max keys is not a multiple of fanout
18:   if( remainder != 0 && remainder <= fanout/2 ) {
19:     node = make a new internal node;  // first internal node
20:     insert the "(remainder + fanout )/2-1" keys to node;
21:     link the node to "(remainder + fanout )/2" child nodes as
          children;
22:     add the "(remainder + fanout )/2" th key to keylist[];
23:     childNodes[numNextKeys++] = node;

24:     node = make a new internal node; // second internal node
25:     insert the "(remainder + fanout )/2-1" keys to node;
26:     link the node to (remainder + fanout )/2 child nodes as children;
27:     add the "(remainder + fanout)"th key to keylist[];
28:     childNodes[numNextKeys++] = node;
29:     i = i + "(remainder + fanout)";
30:   }
31:   else if ( remainder !=0 && remainder > fanout /2 ) {
32:     node = make a new internal node;  // first internal node only
33:     insert the "(remainder-1" keys to node;
34:     link the node to "remainder" child nodes as children;
35:     add the "remainder" th key to keylist[];;
36:     childNodes[numNextKeys++] = node;
37:     i = i + "remainder";
38:   }
39:   // make/link internal nodes & store next keys until the last max
        keys
40:   for ( ; i< numMaxKeys- fanout; ) {
41:     node = make a new internal node;
42:     for ( j=0; j < fanout -1; j++ ) {
43:       node.child[j] = childNodes[i];  node.keys[j] = keylist[i];
44:       node.nKeys++; i++;
45:     }
46:     node.child[j] = childNodes[i++];  childNodes[numNextKeys] =
            node;
47:     keylist[numNextKeys++] = keylist[ i++];
48:   }
49:   numMaxKeys = numNextKeys; numNextKeys=0;
50: } // end of while
51: // last step: make a root node:
52: root = make a new internal node;
53: for( j=0; j < numMaxKeys-1; j++ ) {
54:   root.child[j] = childNodes[j]; root.keys[j] = keylist[j]
55: }
56: root.child[j] = childNode[j];
end
```

Fig. 8 Max_constructIndexStructure algorithm for building the B+-Tree index structure.

overview which was illustrated in Introduction Section, considering that each node is filled out accordingly to the fanout and fit-ratio values of the rebuilt B+-Tree index.

At first, we find the total number of pre-stored max keys (*numMaxKeys*) and read *numMaxKeys* number of keys from the checkpoint block. For each key one leaf node is created and the key is inserted into the leaf node and also added to the *keyList* for the next level, i.e., *height-1* level (line: 1–11).

Then we create and fill out the nodes level by level in a bottom-up and left-to-right fashion until we do the root (line: 12–56). For each level (*height-1*, …, *1*), every keys at multiple positions of *fanout* in the *keyList* for the current level are not used for filling a node at the current level. Instead they are used as search key values of nodes at its upper level. If the number of keys in the *keyList* for the current level is not a multiple of the fanout, we check the following condition to distribute the *remainder* (= *numMaxKeys mod fanout*) keys, and then the number of rest keys is a multiple of *fanout* and therefore those keys can be fully inserted into the next nodes of the current level (line: 15–38).

(1) *Remainder* $> fanout/2$: In this case, (*remainder-1*) keys are inserted into the first node of the current level, and the key at *remainder*th position is stored into the *keyList* for the next level. (An example will be shown at below.)

(2) *Remainder* $\leq fanout/2$: Since a node except the root of a B+-Tree should be filled more than half of the fanout values, we may not fill (*remainder-1*) keys into the first node as above. Instead, we distribute the *remainder plus fanout* keys into the first two nodes by half each so that both nodes are filled more than half of the fanout. (The example illustrated in Introduction Section belongs to this case.)

This process is repeated for each level until we create and fill the root node. That is, if the number of keys to be filled at the current level, i.e., the number of keys in the *keyList* for the current level, is less than or equal to *fanout* (line: 14), we can fill all the keys into a node, which should be the root (line: 51–56). Then the algorithm terminates.

Note that for simplifying the illustration we do not consider the *fit-ratio* of a B+-Tree index in the algorithm. Nevertheless, the algorithm can be easily extended to accommodate *fit-ratio* if we apply "*fanout* × *fit-ratio*" value instead of "*fanout*" value to fill each node.

Figure 9 shows an example of constructing an index structure when we have the pre-stored max keys of {3, 6, 9, 12, 15, 18, 21} at the checkpoint block. Assume that the *fanout* of the B+-Tree is 4, i.e., every node but the root of the B+-Tree should have at least 2 keys. Figure 9 (a) shows the process of creating leaf nodes from the max key values. After we create the nodes at leaf level, the *keyList* for *height-1* level is set to the pre-stored max key list {3, 6, 9, 12, 15, 18, 21}.

Figure 9 (b) shows the process of creating nodes at *height-1* level by using the *keyList* for *height-1* level. Since the number of keys in the *keyList*, 7, is not a multiple of the fanout value 4, and *remainder* (7 mod 4 = 3) is greater than half of fanout, we can insert the first two keys {3,6} into the first node and leave the third key 9 as a search key for the upper level, i.e., add 9 into the *keyList* for *height-2* level. Then the rest number of keys {12, 15, 18, 21} is a multiple

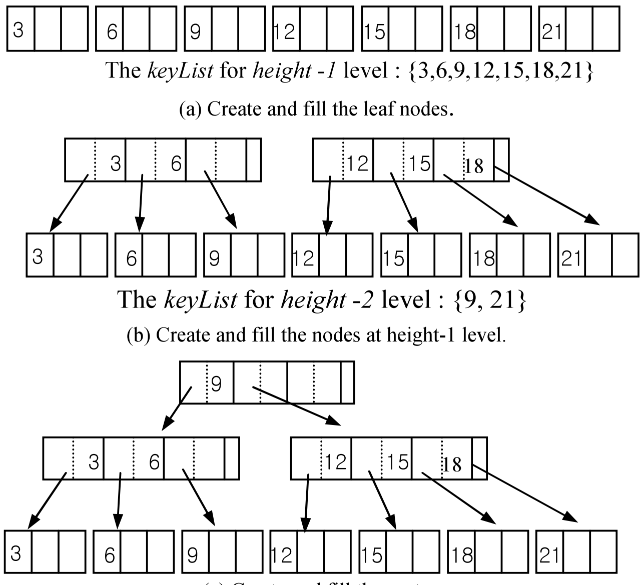

3
6
9
12
15
18
21
The keyList for height -1 level : {3,6,9,12,15,18,21}
(a) Create and fill the leaf nodes.
The keyList for height -2 level : {9, 21}
(b) Create and fill the nodes at height-1 level.
(c) Create and fill the root.

Fig. 9 Example of building an index structure: Max key values {3, 6, 9, 12, 15, 18, 21}.

```
Max_insertKey2index()
while ((record = read a data record) != NIL) {
    (key, pointer) = get a key and record pointer for the record;
    find a leaf node for the key ;
    insert (key, pointer) into the leaf node;
}
end
```

Fig. 10 Max_insertKey2index: To read the data records and insert keys into the index.

of 4. According to the algorithm, the keys at multiple positions of *fanout* in the *keyList* for the current level are not used for filling a node at the current level. Therefore the first *fanout-1* = 3 keys {12, 15, 18} are inserted into a next node of the current level, and leave 21 as a search key for the upper level. After we are done for *height-1* level, the *keyList* for *height-2* level becomes {9, 21} of which cardinality is 2.

Since 2 is less than the fanout value, we can build the root node. Then the algorithm terminates (Fig. 9 (c)). Note that the root does not need to have 21 (the last key of *keyList*) as a search key. That is why we insert *numMaxKeys-1* number of keys into the root in the algorithm (line: 51–56).

3.2.2 Read the Data Records and Insert Keys into the Index

Once we construct a B+-Tree index structure, we read the data records and insert the keys of records into the corresponding leaf nodes of the index. The algorithm is shown in Fig. 10. Note that during this process we are not changing the B+-Tree internal index structure but filling the existing leaf nodes.

It is costly and definitely unnecessary to sort a leaf node whenever a key is inserted into the node. Instead, we defer

```
Max_sortLeafNodes()
    leaf = find the first leaf node;
    while (leaf != NIL) {
     sort the keys in leaf node;
     leaf = find the next leaf node;
    }
end
```

Fig. 11 Max_sortLeafNodes: To sort the keys in leaf nodes.

```
Max_PL ()
    shared_indexReady = 0;
    thread_create( MaxPL_readNreconstructData );
    thread_create( MaxPL_insertKey2Index );
    thread_create( MaxPL_constructIndexStructure );
    thread_create( MaxPL_sortLeafNodes );
    thread_join( );
end
```

Fig. 12 Max_PL: Introducing a parallelism to max algorithm.

to sort the leaf node until we finish loading up all the keys from the data records.

3.2.3 Sort the Keys in Every Leaf Node

The next step is to sort the keys within each leaf node (Fig. 11). As mentioned above, this procedure can be called after we finish loading up all the keys into the leaf nodes. Or since the keys are ordered between leaf nodes, we may perform sorting a leaf node independently to another leaf node at some point, i.e., when a node is fully filled.

3.3 The Max-PL Algorithm

Max-PL is the algorithm that applies a parallelism to the MAX algorithm. The pseudo code of Max-PL is shown in Fig. 12.

As in Fig. 12, Max-PL employs multiple threads *MaxPL_readNreconstructData*, *MaxPL_insertKey2Index*, *Max_constructIndexStructure*, *MaxPL_sortLeafNodes* to perform parallelism as follows in the subsections, and synchronization between two threads is done by the *conditional mutex*.

3.3.1 Pipelining to Read Data Records and Insert the Keys into the Index

Once we reconstruct a B+-Tree index structure, it is possible to pipeline read a data record and insert the key for the record into the index. Figure 13 shows the procedure. After it read a data block for data records, it increments the number of read records, *shared_numReadRecs*, and send a signal to *MaxPL_insertKey2Index* thread to insert the keys of data records. Note that this is not feasible in Batch-Construction, since it needs to read all the data blocks in order to sort the keys.

```
MaxPL_readNreconstructData ()
  find the total numRecs records to be rebuilt;
  shared_numReadRecs = 0;
  while ( numRecs > 0 )  {
    read a records and numRecs--;
    mutex_lock(mutex );
      put the record key & RID to shared_recs[
                                    shared_numReadRecs];
      shared_numReadRecs++;
    mutex_unlock(mutex);
    send signal to MaxPL_insertKey2Index thread;
  }
end
```

Fig. 13 MaxPL_readNreconstructData: Pipelining to read data and insert the keys.

```
MaxPL_insertKey2Index ()
while( ! shared_indexReady )
  wait the signal from MaxPL_constructIndexStructure thread;
numInserted=0; numReadySort =0;
while ( numInserted < numRecs )  {
  while ( numInserted >= shared_numReadRecs )
    wait the signal from MaxPL_readNreconstructData thread;
  mutex_lock(mutex);
    numRecs2Insert = shared_numReadRecs - numInserted;
  mutex_unlock(mutex);
  for( i=0; i < numRecs2Insert; i++) {
    find a leaf node for the key( shared_recs[numInserted]) ;
        lock( leaf );
        insert the key to the index ;    unlock( leaf );
    if( leaf.nKeys >= NUM_KEYS_IN_NODE ) {
      mutex_lock(sortMutex); put the leaf to shared_sortLeafs;
          numReadySort++;
          send signal to MaxPL_sortLeafNodes;
      mutex_unlock(sortMutex); }  // end of if
    numInserted++;  } // end of for
} // end of while
end
MaxPL_constructIndexStructure ()
   read the max keys from the backup data;
   create the index structure same as the Max Algorithm;
   shared_indexReady = 1;
   send a signal to MaxPL_insertKey2Index;
end
```

Fig. 14 MaxPL_insertKey2Index: Inserting the keys of data blocks in parallel.

3.3.2 Inserting the Keys of Data Records in Parallel

When we insert the keys of the data records into the index, we are not changing the internal index structure but the leaf nodes only. That is, we only need to lock leaf node page, which enables simultaneous modifications to multiple leaf nodes. Figure 14 shows the procedure. It first waits until the main working Max_PL thread finishes reconstructing a B+-Tree index structure. As long as the number of keys from the data records (*shared_numReadRecs*) is less than or equal to the number of records (*numInserted*) that have been already inserted into the index, i.e., there is no keys to be inserted, it waits for a signal from the *MaxPL_readNreconstructData* until the data records are read. Once the signal comes, it begins to insert the keys. Once we fill a leaf node (the number of inserted keys is more than fanout or fanout*fit-ratio), we may sort the node. Then it sends a signal to *MaxPL_sortLeafNodes* thread.

```
MaxPL_sorLeafNodes ()
numLeafs= getNumOfLeafs(); sortedLeafs=0;
while ( sortedLeafs < numLeafs ) {
  while ( sortedLeafs >= numReadySort ) {
    wait signal from MaxPL_insertKey2Index thread;
  numSort= numReadySort - sortedLeafs;
  for( i=0; i < numSort; i++ ) {
    mutex_lock(sortMutex);
      leaf = get the leaf node from shared_sortLeafs;
    mutex_unlock(sortMutex);
    sort leaf;
  }
  sortedLeafs = sortedLeafs + numSort;
}
end
```

Fig. 15 MaxPL_sorLeafNodes: Sorting the leaf nodes in parallel.

Table 1 Parameters and their typical values.

Parameter	Explanation (value for presentation)
n	number of keys
S	data record size in bytes
K	key size in bytes (4)
P	pointer size in bytes (4)
B	node block size in bytes (1024)
R	fit-ratio (0.67) [11]
M	fanout
L	number of leaf nodes

3.3.3 Sorting the Leaf Nodes in Parallel to Each Other

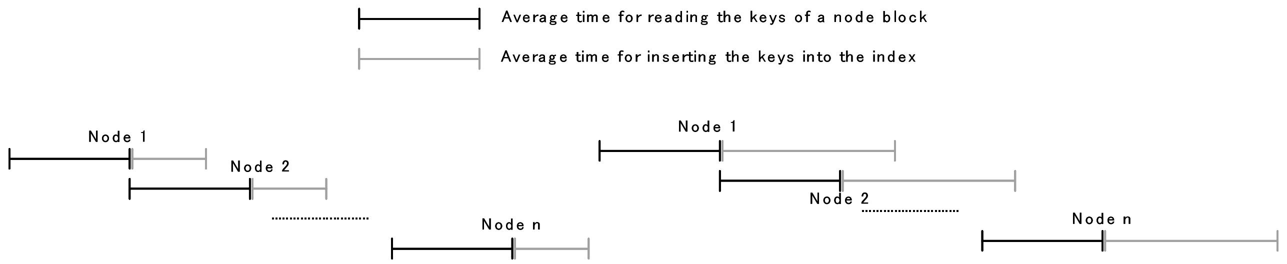
Keys in a leaf node can not be greater than any key in a right neighbor leaf node. That is, we may perform sorting in parallel a leaf node with others. Figure 15 shows the procedure. The *MaxPL_sorLeafNodes* thread needs to wait until we have a leaf node ready from the *MaxPL_insertKey2Index* thread.

4. Complexity Analysis

In this section, we analyze the time and space complexities of our algorithm, and compare them to those of the Sequential Insertion and Batch-Construction algorithms. The parameters for the complexity analysis are shown in Table 1. To simplify the presentation, we assume that a key (K) and a child pointer (P) take the same amount of space, 4 bytes, and a node block size is 1024 bytes. Then the branch factor or fanout, M ($= \lfloor (B-P)/(K+P) \rfloor + 1$), becomes 128. This value is close to the one found in practice [11].

4.1 Time Complexity

We analyze the time complexity of the algorithm by each phase.


Average time for reading the keys of a node block
Average time for inserting the keys into the index
Node 1
Node 2
Node n
Node 1
Node 2
Node n

(a) Loading takes longer than inserting. (b) Inserting takes longer than loading.

Fig. 16 Time for pipelining the reading and inserting the keys in Max-PL.

1st phase: In order to build a B+-Tree index structure, we need to reload the max-keys stored in the checkpoint block. The number of stored max-keys is the same as to the number of leaf nodes, $L = n/(M \times R)$.

Therefore reloading the max-keys incurs $(L \times K)/B$ number of block reads. Then we build the internal index nodes by copying the max-key values of child nodes into their parent nodes. Number of memory copies which this incurs is

$$\frac{n}{(M\times R)} \times \sum_{k=0}^{\log_{(M\times R)} n} \left(\frac{1}{(M\times R)^k}\right).$$

If unfolding this, we can get

$$\frac{na-1}{a(a-1)} \geq 2 \text{ (where, } a = M\times R,\ a \geq 2).$$

Therefore, we get the time complexity $O(n)$.

2nd phase: In order to read all the data blocks, we need $(n\times S)/B$ number of block reads, which is required for any index rebuilding algorithm in common. Within a block there are $\lfloor B/S \rfloor$ records. In other words, a block contains $\lfloor B/S \rfloor$ number of keys to be inserted into the index. Therefore we require $\lfloor B/S \rfloor \times \log n = O(\log n)$ time complexity for processing all the keys of a block.

Note that Max-PL may pipeline reading and inserting the keys of a block to those of the other block. In other words, after a thread A finishes to read the keys of a block, a thread B may begin to insert the keys while the thread A keeps reading the keys of next block. As shown in Fig. 16, this should take $(n \times S)/B$ for loading blocks and $\lfloor B/S \rfloor \times \log n = O(\log n)$ for inserting the keys of the last read block, no matter whether the average block-read time takes longer or shorter than the average inserting time.

3rd phase: Sorting the $M \times R$ number of (key, data_pointer) pairs stored in a leaf node requires $(M \times R) \times \log(M \times R)$ time complexity. There are $L = n/(M \times R)$ number of leaf nodes, and each leaf node can be sorted independent to others. Therefore, similarly to the 2nd phase, if we have P degree of parallelism for this phase (i.e., sorting leaf nodes in parallel with P number of threads), the time complexity becomes

$$\frac{1}{P} \times \left[\left(\frac{n}{M\times R}\right) \times O\left\{(M\times R)\times \log(M\times R)\right\}\right] = O(n).$$

4.2 Space Requirement

Max-PL needs to store the max keys of each leaf node in the checkpoint block. The space required to store the max keys in a checkpoint block is *the number of leaf nodes* $\times$ *key size*. Then the number of leaf nodes over the number of entire node ratio in a B+-Tree is $M^{h-1}/\{(M^h-1)/(M-1)\}$, where M is a branch factor and h is the height of the tree. For example, given that h is 3, M is 128 and the *fit-ratio* is 0.67, then this ratio becomes about 0.98. The total space for holding the max keys over the total index size ratio is (*the number of leaf nodes* $\times$ *key size*) / (*the number of entire node* * *node block size*), which is $0.98 \times K/B$. For example, if K is 4 and B is 1024, then this value results in $3.8 * 10^{-3}$. In other words, the size of the contents to be stored is less than 0.38% of the whole index size. Since index size is usually smaller than the actual data size, we can conclude that additional work of our method is far less than the work for the whole data backup. Moreover, by processing data backup and index backup in parallel, we may reduce the overhead even further.

In "Building the B+-Tree structure", the space requirement for the keyList is linear to the number of keys, i.e., $K \times n/(M \times R) = O(n)$. "Sorting the keys in every leaf node" can be done using an in-memory algorithm such as quick sorting or heap sorting. Therefore in terms of space requirement, Max-PL is the same as to Sequential Insertion.

Note that the Batch-Construction algorithm needs to sort the keys before it starts to construct the index structure. Sorting requires loading up all the keys from the disk space, which is sometimes infeasible due to the large number of keys. Although employing an external sorting such as *k-way merging* can handle this, it still requires extra memory space.

5. Experimental Evaluation

5.1 Experimental Environment

For the performance comparison study, we implement all the algorithms: Sequential Insert, Batch-Construction [7], Max, and Max-PL. All the algorithms are implemented in C, and compiled by *cc* of the Sun WorkShop 6 v.5.3 with *-fast* option to optimize the codes. We run our experiments on an Ultra Sparc III machine (two 1.2 GHz CPUs, 4 GB

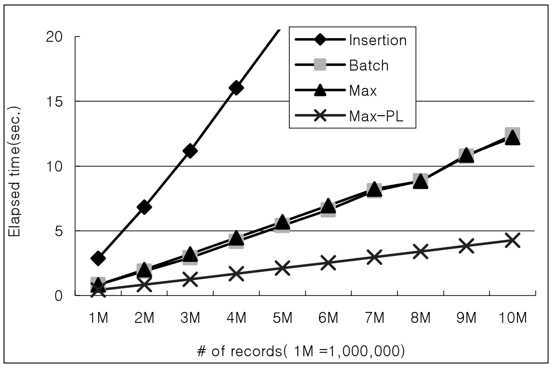


Fig. 17 Rebuilding times for each algorithm to rebuild a B+-Tree index.

RAM) running Solaris 5.9.

We first create a table with one million to ten million records, and measure the times to rebuild a B+-Tree index for the table. Each record is of 128-byte size and contains a key which is chosen randomly within the range from 0 to 20 billion. We choose keys to be 4-byte integer. We also assume a data pointer take 4 bytes. We perform the various settings for the node size and fit-ratio of the B+-Tree index, to see whether they affect the performance of algorithms. The time we measured is the wall-clock time to rebuild a B+-Tree index. For each experiment, we repeated each test five times and report the minimal time.

The Max-PL algorithm implemented for this experiment employs the parallelism in pipelining to read the records and insert the keys and in sorting the leaf nodes. We use the conventional buffered I/O rather than parallel I/O in that the actual behavior of I/O depends on the OS kernel and the buffered I/O is more generic and easy to implement than the parallel IO. The buffer size we use is 8 K and 16 K, and both result in similar results. In this paper we illustrate the case for 16 K size only.

5.2 Results

In the first experiment, we vary the number of records for a table, from 1 million to 10 million, and measure the time that each algorithm takes to rebuild a B+-Tree index for the table. Figure 17 shows the result. The Sequential Insertion algorithm shows the slowest performance. While the MAX algorithm shows similar performance to the Batch-Construction algorithm, the Max-PL algorithm shows the best performance among all others. Using Max-PL, we could achieve a speedup of 2.0 to 2.9 over Batch-Construction and 6.7 to 11.7 over Sequential Insertion. On average Max-PL performs 245% and 920% better than Batch-Construction and Sequential Insertion, respectively.

Figure 18 shows the CPU portions of the work-phases that each algorithm requires to rebuilding a B+-tree index. While the total elapsed time of MAX is a bit smaller than and almost similar to the one of Batch-Construction, the work-phases and its CPU portions of MAX are quite different to those of Batch-Construction. In MAX, the time for reading the data records and inserting keys into the index (after building a B+-Tree index structure) takes the most, and then the time for sorting the keys in leaf nodes takes next. Compared to this, in Batch-Construction, sorting the keys should be performed before building an index, and it takes most of the elapsed time.

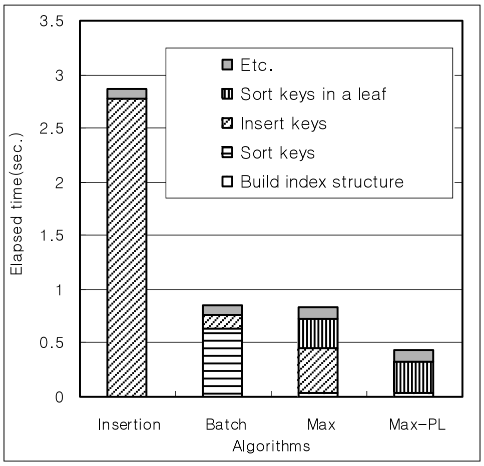


Fig. 18 Partitioning rebuilding work.

A most benefit of MAX comes from the feature that sorting the keys in leaf nodes may take place in parallel to reading the data records and inserting keys into the index, which makes it possible to change MAX into its parallel version the Max-PL. Compared to this, sorting the whole keys in Batch-Construction can be performed only after reading all the records from the database, i.e., sorting keys and reading the data records can not be performed in parallel. In other words, we may parallelize to sort the whole keys in Batch-Construction, by using K-way merging and multiple threads, but may not to parallelize to read the data records and to sort the keys from the records. Note that in this experiment the time for reading the data records and inserting keys into the index is negligible in that it is done in parallel within the time for sorting the keys. Furthermore, sorting keys in a leaf node can be performed immediately after the node becomes full over a specific filling ratio. Finally, sorting keys of a node needs to read a just one block node, which benefits the locality.

As mentioned in the previous section, the current implementation of Max-PL does not employ the parallelism in disk I/O. We can expect that, by reducing the I/O time to read data records from disk using a parallel I/O, the performance difference will increase.

The experiment results of the index rebuilding time in accordance to the node size are shown in Fig. 19. The table for this experiment contains 1 M number of records. For the case of Sequential Insertion, as the size of the node increases, it takes more time because the overhead of moving other keys within the node increases. On the other hand, for Batch-Construction and our algorithms, as the node size increases, the height of a tree decreases, thus the index re-

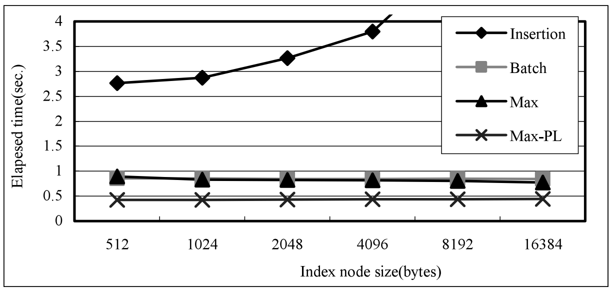


Fig. 19 Results in accordance to the node size.

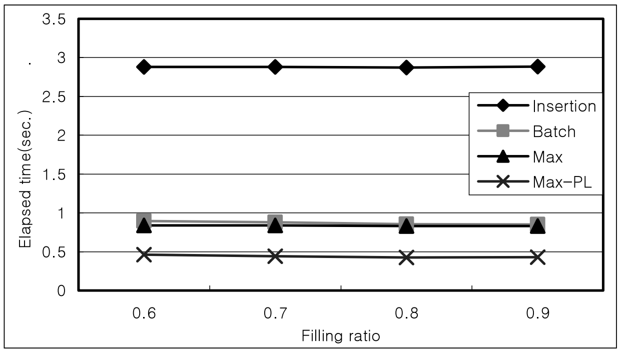


Fig. 20 Results in accordance to the fit-ratio.

building time decreases (In the experiments of Fig. 17, the node size was 1 K bytes.).

Finally, we measure the index rebuilding time in accordance with the fit-ratio. Figure 20 shows the results. The Sequential Insertion algorithm shows the almost same rebuilding times for the different fit-ratio values. On the other hand, for Batch-Construction and our algorithms, as the fit-ratio decreases, the height of a tree increases, thus the total reconstruction time increases.

6. Conclusion

In this paper we present an algorithm for fast rebuilding B+-Trees. The algorithm requires the checkpoint activity to include the operations of storing the max key value of each leaf nodes, and use them to rebuild the B+-Tree index structure. It may employ parallel processing in several phases, and we call the parallel version of the algorithm as Max-PL.

Unlike Sequential Insertion, a basic B+-Tree insertion algorithm, Max-PL has no overhead of the node split and allows the parallel processing of restoration processes, which is different from Batch-Construction, a batch construction method. Thus, the time needed to reconstruct the index could be reduced. According to the experiment results, Max-PL runs on average at least 670% faster than Sequential Insertion and 200% faster than Batch-Construction.

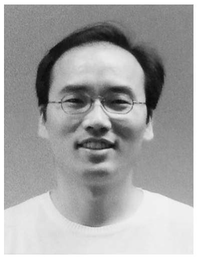
We are currently examining the performance improvement aspect of Max-PL by introducing and implementing additional features into the algorithm. One of them is to adapt the pre-fetching method [1] to Max-PL. We also plan to analyze how much reducing the I/O time in recovery improves the performance of Max-PL using a parallel I/O.

Acknowledgments

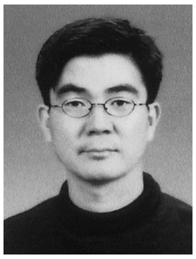
This research was supported in part by the Ministry of Information & Communications, Korea, under the Information Technology Research Center (ITRC) Support Program.

References

[1] S. Chen, P.B. Gibbons, and T.C. Mowry, "Improving index performance through prefetching," Proc. 2001 ACM SIGMOD Conf., pp.235–246, Santa Barbara, USA, May 2001.
[2] L. Gruenwald, M. Dunham, J.J. Huang, J. Lin, and A. Peltier, "Recovery in main memory databases," International Journal of Engineering Intelligent Systems, vol.4, no.3, pp.57–63, 1996.
[3] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," IEEE Trans. Knowl. Data Eng., vol.4, no.6, pp.509–516, 1992.
[4] IBM Corporation, http://www-128.ibm.com/developerworks/linux/library/l-jfs.html, JFS overview, Jan. 2000.
[5] H.V. Jagadish, A. Silberschatz, and S. Sudarshan, "Recovering from main memory lapses," Proc. 19th Conf. on Very Large Databases, pp.391–404, Dublin, Ireland, Aug. 1993.
[6] J.H. Kim, J.Y. Kim, S.W. Kim, S.H. Ok, and H.Y. Roh, "An efficient strategy for adding bulky data into B+-tree indices in information retrieval systems," Lecture Notes in Computer Science 2555, p.531, 2002.
[7] S.W. Kim and H. Won, "Batch-construction of B+-Trees," Proc. 16th ACM Symposium on Applied Computing, pp.231–235, Las Vegas, USA, March 2001.
[8] E. Levy and A. Silberschatz, "Incremental recovery in main memory database systems," IEEE Trans. Knowl. Data Eng., vol.4, no.6, pp.529–540, 1992.
[9] J. Lin and M.H. Dunham, "Segmented fuzzy checkpointing for main memory databases," Proc. 11th ACM Symposium on Applied Computing, pp.158–165, Philadelphia, USA, Feb. 1996.
[10] D. Lomet and B. Salzberg, "Concurrency and recovery for index trees," VLDB Journal, vol.6, no.3, pp.224–240, 1997.
[11] R. Ramakrishnan and J. Gehrke, Database Management Systems, 3rd ed., McGraw-Hill, 2003.
[12] K. Salem and H. Garcia-Molina, "Checkpointing memory-resident databases," Proc. 5th International Conf. on Data Engineering, pp.452–462, Bangalore, India, March 1989.
[13] Silicon Graphics, Inc., http://oss.sgi.com/projects/xfs/, XFS: A high-performance journaling file system, July 2005.



Ig-hoon Lee received the B.S. and M.S. degrees in Computer Science from University of Seoul at Seoul, Korea, in 1996 and 1998, respectively. He also received the Ph.D. in Computer Science from Seoul National University at Seoul, Korea, in 2005. He is currently a director of Technology Group at Prompt, Co. Ltd., Seoul, Korea. His current research interests include database systems, main-memory systems, product information management, and Ontology.



Junho Shim received his M.S. degree from Seoul National University at Seoul, Korea, in 1994 and the Ph.D. degree in Computer Science from Northwestern University at Evanston, Illinois, USA, in 1998. He is currently an Associate Professor in Department of Computer Science at Sookmyung Women's University, Seoul, Korea. Previously he worked at Computer Associates International, New York, USA, and held assistant professor position with Drexel University, Pennsylvania, USA. His current research interests include database systems, data warehousing, e-Commerce, product information management, and Ontology.

Sang-goo Lee received the B.S. degree in Computer Science from Seoul National University at Seoul, Korea, in 1985. He also received the M.S. and Ph.D. in Computer Science from Northwestern University at Evanston, Illinois, USA, in 1987 and 1990, respectively. He is currently an a professor in School of Computer Science and Engineering at Seoul National University, Seoul, Korea. He has been the director of Center for e-Business Technology at Seoul National University, since 2001. His current research interests include e-Business technology, database systems, product information management, and Ontology.