A Degradable B^{link}-Tree with Periodic Data Reorganization

ING-RAY CHEN*

Institute of Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan, Taiwan
Email: irchen@ws1.iie.ncku.edu.tw

This paper develops a periodic data reorganization algorithm for the B^{link}-tree data structure in concurrent environments, and identifies conditions under which the data reorganization should be performed in order to minimize the response time per access operation.

Received October 3, 1994; revised January 17, 1995

1. INTRODUCTION

The classic B^{link}-tree data structure due to Lehman and Yao [8] allows a higher level of concurrency than traditional B^+-trees [4] (called B*-trees by Lehman and Yao) for reading, updating, deletion and insertion concurrent operations by introducing cross-linked pointers for nodes at the same level. This unique design, henceforth referred to as the LY algorithm, allows a read operation, which is fundamental to other operations, to be processed without having to lock any tree nodes as it searches from top-down for a leaf node that contains the key to be searched, thereby increasing the degree of concurrency over traditional B^+-tree algorithms which require read locks for read operations. The LY algorithm resolves the interference between a read and an insert operation by making use of the cross-linked pointers. That is, if a reader cannot find the key in node a when it is guided by the tree structure, it can follow the cross-linked pointer to a's right neighbor node b to continue searching for the key, knowing that due to concurrency a may be split into two nodes, a and b, by a concurrent insert operation.

The LY algorithm is efficient because a reader never locks any node; an update or a delete operation only locks one node (i.e. the leaf node that contains the key to be updated or deleted); and an insert operation locks at most three nodes at a time. Consequently, the degree of interference among concurrent operations is minimum. The LY algorithm is also 'clean' in the sense that the deadlock-free property can be proved easily by showing that concurrent insert operations—which are the only operations that may lock more than one node—do not cause deadlocks. A less straightforward proof was used to show that the functional requirements of all operations are satisfied [8].

Figure 1 shows a possible node structure of a k-degree B^{link}-tree in which each node has $2k$ keys, $2k+1$ child pointers, a high key indicating the upper bound on the key values that may be stored in the subtree rooted with the node, and a cross-linked pointer pointing to the next node at the same level (or pointing to null if the node is the rightmost node on a level). Figure 2 shows an instance of the B^{link}-tree with $k = 2$. Note that nonleaf nodes only serve as index nodes. The child pointers of a leaf node directly point to records associated with the key values stored in the leaf node.

A major drawback of the LY algorithm is that the solution for the delete operation is the same as that of the update operation, i.e. just updating the leaf node that contains the key to be deleted without regard to whether the leaf node will become underflowed or empty after the key is removed from the node. For example, in Figure 2 after deleting key value 38 from the tree, the node that originally contains 38, i.e. node E, becomes underflowed (i.e. less than half-full) and after further deleting key value 41 from the tree, it becomes empty. This simple solution for the deletion operation may needlessly waste space at the leaf node level. Furthermore, it is possible that the height of the tree may become bigger than necessary because at the nonleaf levels only the insert operation can split internal nodes, while the delete operation never merges internal nodes. This disadvantage can be described based on the concept of *data reorganization*. The LY algorithm can be viewed as an algorithm that performs 'on-the-fly' data reorganization for update and insert operations only, but not for delete operations which can introduce garbage nodes into the data structure. Consequently, the performance can degrade over time since garbage nodes are left unmaintained.

Two possible approaches exist for overcoming this drawback. One approach is to devise on-the-fly algorithms for the delete operation so that it can lock as few nodes as possible and merge underflowed or remove empty nodes when necessary. Unfortunately, such algorithms have not been reported in the literature for the B^{link}-tree data structure, possibly due to complexity reasons. Other on-the-fly algorithms do exist in the literature for other variants of B-trees (e.g. a new structure of B-trees with lazy parent split [9]). However, for these latter class of algorithms, read locks have to be used for read operations in order to resolve read-write access conflicts since cross-linked pointers are not used in the structure as in B^{link}-trees. Therefore, we expect that the degree of concurrency achievable by this latter group of algorithms is lower than that achievable by

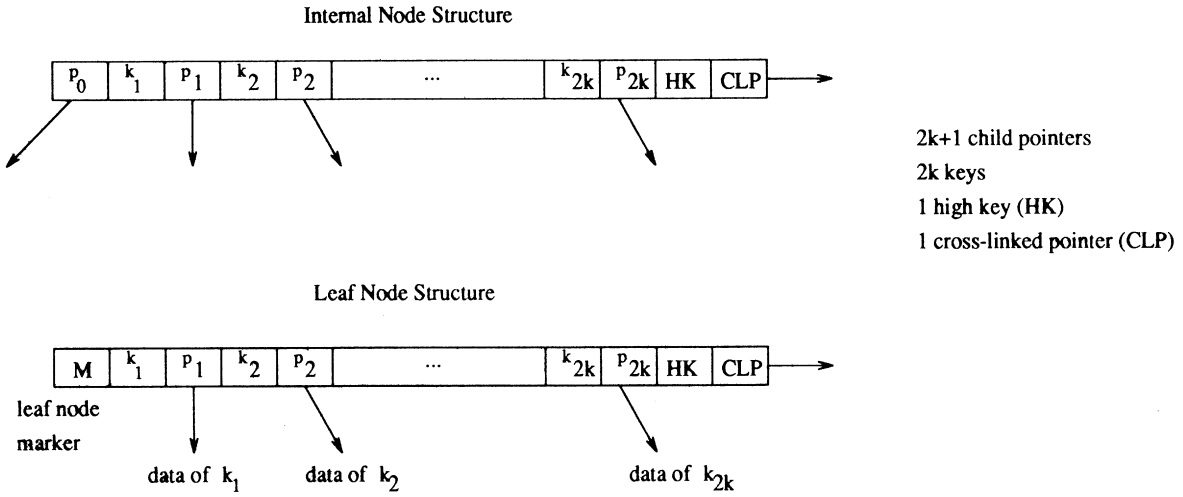

FIGURE 1. B^{link} tree node structure.

the LY algorithm which requires no read locks for read operations.

Another approach is to introduce one or more concurrent maintenance processes [3, 6, 10, 11] to reorganize the tree nodes at both the parent and leaf levels and clean up garbage nodes created by delete operations at the leaf level. Sagiv [11] has introduced such maintenance processes in his algorithm. He modified the LY algorithm such that the insert, delete, and maintenance operations (or processes) only need to lock 1, 1, and 3 nodes, respectively, while the read operation still does not need to lock any node at all. However, his algorithm requires the use of time-stamps to keep track of when deleted nodes can be garbage-collected, and also the use of a special mechanism to restart a reader when the reader reaches a wrong (garbage) node. As a result, the arguments used in the proof for deadlock-free and functional properties in his algorithm are more complicated and difficult to follow. Furthermore, without a performance analysis, it is not clear how much performance improvement can be obtained.

This paper investigates *periodic* reorganization [1, 2] as a design alternative to overcome the drawback of the LY algorithm on concurrent B^{link}-trees. Previously, we have studied the performance and stability behavior of degradable data structure queueing servers that serve operations sequentially in a client-server computational model. Here, we consider the use of periodic maintenance in a shared data-space model. We make a shared B^{link}-tree degradable by (i) performing data reorganization only periodically instead of concurrently; (ii) modifying the insert operation in the LY algorithm so that during the maintenance-free time intervals, the response time of the insert operation is also minimum. The benefits of periodic data reorganization are: (i) it has a very simple and clean solution when compared with a concurrent maintenance solution or the classic LY algorithm so that the design is easy to follow and the proof of the deadlock-free and functional properties is trivial; (ii) every non-read operation needs to lock only one node so that the degree of concurrency is maximum during the maintenance-free periods; (iii) the tree struc-ture is reorganized periodically with all the garbage nodes removed so that the problem of indefinite growth of the tree space is avoided; (iv) the behavior of the tree can be described by a model which allows the best time interval between two successive periodic data reorganization operations to be determined as a function of database environment variables (e.g. the number of con-current

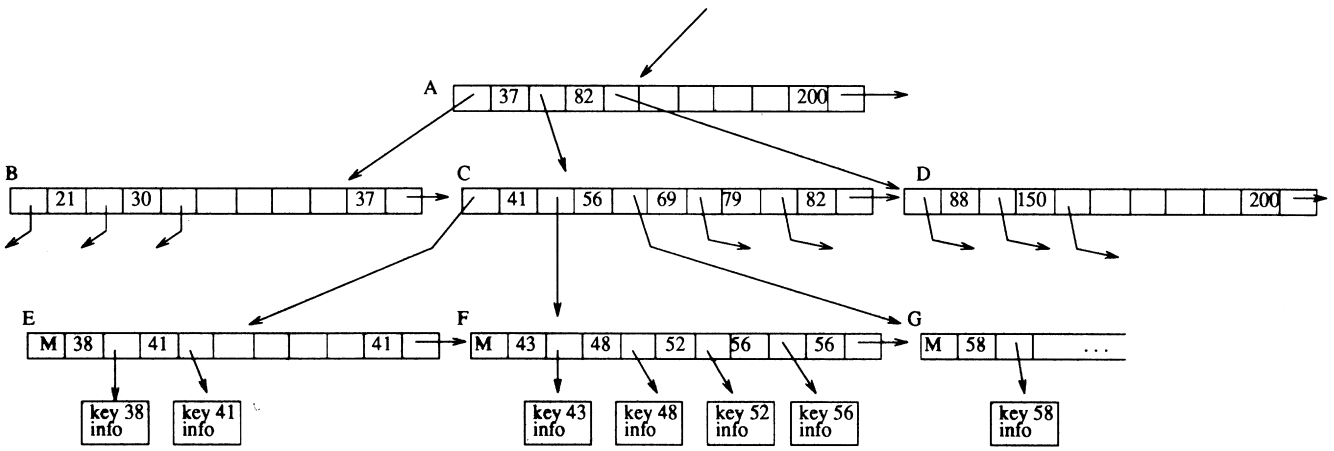

FIGURE 2. An example B^{link}-tree with $k = 2$.

operations and probability of insert, etc.) such that the performance of the resulting system can be ascertained.

The rest of the paper is organized as follows. Section 2 briefly reviews the LY algorithm, and discusses how we modify the insert operation so that the response times for all operations (i.e. insert, delete, update and read operations) are virtually the same. Section 3 begins by posing an optimization problem which we are trying to solve for seeking the best time interval for performing periodic data reorganization such that the system response time per operation is minimum. It then addresses the performance issue by constructing a Markov model for describing the dynamical behavior of the B^{link}-tree under our algorithm. Then, the solution for the best time interval between two consecutive data reorganization operations is derived as a function of database environment variables. Section 4 illustrates the proposed technique with an example system in a con-current database environment and gives physical interpretations of the result. Finally, Section 5 summarizes the paper and outlines some future research areas.

2. A PERIODIC MAINTENANCE ALGORITHM FOR B^{link}-TREES

The read, update and delete operations in our algorithm for the B^{link}-tree structure as shown in Figure 2 are the same as those in the LY algorithm [8]. Specifically, the read operation searches a key from the top of the tree without locking any node. In cases a read operation in searching for a key is led to a node, say *a*, by the tree structure but the highest key value contained in node *a* is smaller than the search-key value, then the cross-linked pointer of node *a* is followed so that the search can continue from the right neighbor of node *a*. Eventually, a node with a range covering the search-key value will be found from which the reader can determine whether the search-key exists or not. This provision handles the interference problem between concurrent read and insert operations as the latter may split a node into two nodes while the former is trying to find a key in the node just being split. For example, if an insert operation adds a key value of 50 into the tree shown in Figure 2, thus causing node F to be split into nodes F and F′ (see Figure 3 or 4 for illustration) and if at this moment a search operation is also led to node F trying to locate key value 56 but just to find that the highest key value in node F after split is 50, then the search operation will follow the cross-linked pointer of F and continue the search in node F′.

The update (delete) operation calls the search operation first to find the node containing the key to be updated (deleted). Then, the node is locked, updated, and unlocked (in that sequence).

The insert operation of our algorithm is different from its counterpart in the LY algorithm. Instead of updating the parent nodes in a bottom-up fashion in case a leaf node is split into two, our algorithm leaves the task of data reorganization to a maintenance operation which is invoked periodically. As a result, the insert operation is similar to an update or delete operation, i.e. it first calls the search operation to find the node to insert the key; then, it locks the leaf node to update the node; then, it unlocks the node. The same procedure follows even if the leaf node being inserted is full and therefore must split into two, in which case the keys are distributed evenly between the node being split and a newly-allocated node with the cross-linked pointers properly updated before the node being split is unlocked. Our modified algorithm eliminates the locking/unlocking overhead of the internal nodes for all operations. In other words, the only locking/unlocking overhead for all non-read operations (i.e. update, delete or insert) is limited to locking/unlocking a single node at the leaf level. The reason that the internal nodes are not updated for insert operations in our algorithm is that data reorganization is performed periodically to not only remove empty and underflowed nodes but also update internal nodes pointers so that the amortized data reorganization cost per operation is minim-ized, thus making updating internal nodes for insert operations on-the-fly unnecessary and cost-ineffective.

Figures 3 and 4 illustrate the difference between the LY and our algorithms in insertions. Figure 3 shows the tree structure after inserting key value 50 into the tree shown in

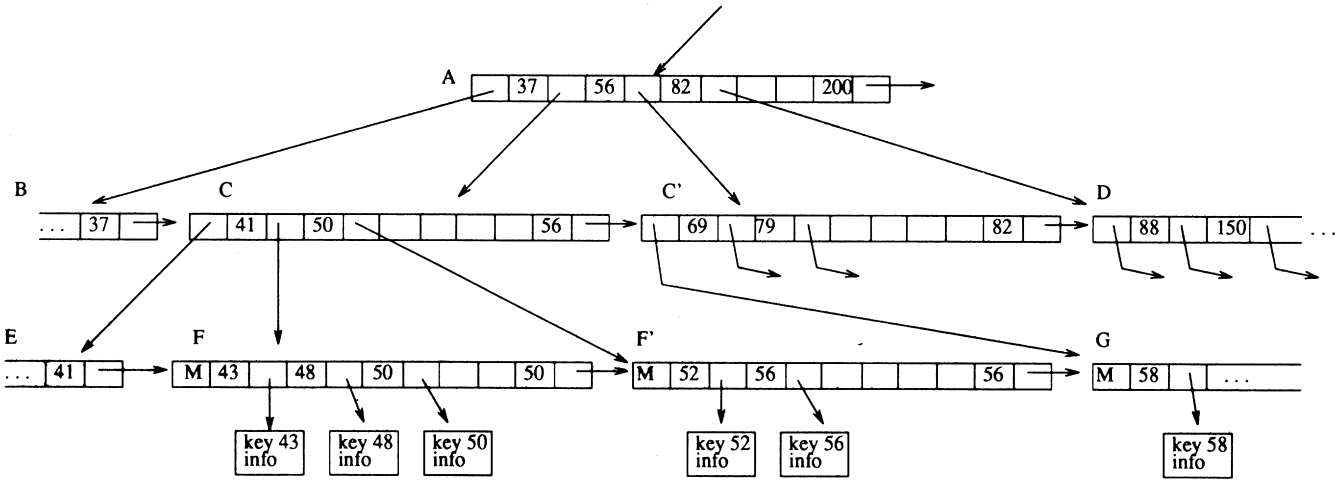

FIGURE 3. Tree structure after inserting 50 based on the LY algorithm.

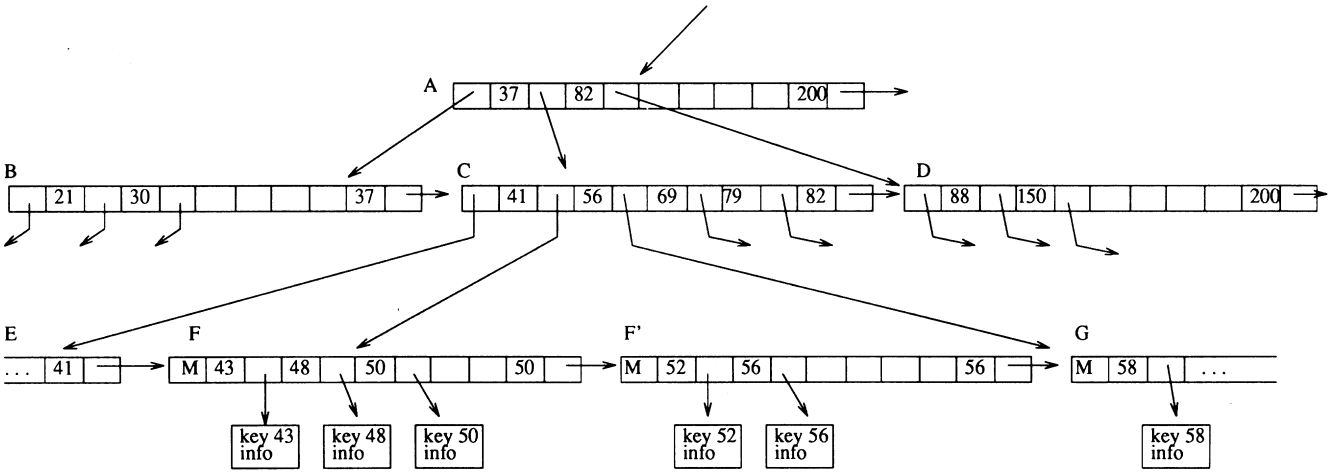

FIGURE 4. Tree structure after inserting 50 based on our algorithm.

Figure 2 under the LY algorithm, while Figure 4 is for the same insert operation under our algorithm. In Figure 3, parent nodes of node F are updated recursively in a bottom-up fashion, i.e. node C is split into nodes C and C′ and node A is also updated as a result of node C being split. In this case, the insert operation must lock nodes F, C and A (although not all at once) under the LY algorithm. In Figure 4, no parent nodes of node F are updated under our algorithm and therefore only node F must be locked.

3. PERFORMANCE ASSESSMENT

We assume that the system allows N transactions to access the B^{link}-tree concurrently. Each transaction is an insert operation with probability q. For ease of presentation, we also assume that each transaction is a delete operation with probability q and a read/update operation with probability $1–2q$. In this formulation, the probability of insert is the same as the probability of delete so that the system remains in the steady state at time infinity. As will be seen later, the last assumption is not needed in the analysis. Each transaction acts independently and accesses a key randomly. When a transaction is completed, another transaction immediately takes its place so the number of transactions in the system is always N. We assume that the database system has M keys which represents the stable database size in the steady state. We distinguish the 'database size' from the 'tree size' which is the number of nodes (disk pages) used by the tree, covering both the internal and leaf nodes. Our algorithm performs data reorganization periodically and, after a data reorganization operation is performed, all internal nodes are updated and all garbage leaf nodes (i.e. nodes that are underflowed or empty due to delete operations) are collected such that each node is two-thirds full.

Conceptually, we say that the B^{link}-tree immediately after a data reorganization is in its *strong* state [10] because the performance of the B^{link}-tree is the highest at that point. As more operations are serviced following a strong state, the B^{link}-tree gradually migrates to a *weak* state because more nodes are being split and more internal nodes are not being updated and, consequently, the performance of the B^{link}-tree deteriorates because each operation has to travel through more leaf nodes in the B^{link}-tree in order to access a key. The performance of the B^{link}-tree continues to deteriorate until a data reorganization operation is performed to bring the data structure to its strong state again. In the paper, we will use the term 'maintenance' interchangeably with the term 'data reorganization'.

The migration of the B^{link}-tree from a stronger state to a weaker state can be characterized by the growth of the tree size. We call the tree size after a periodic maintenance as the *stable tree size* at which the B^{link}-tree is at its strongest state. Then, we can measure the performance degradation of the B^{link}-tree by the increase of its tree size dynamically. We call the difference between the tree size at any time following a maintenance operation and the stable tree size as the 'degradation level' of the B^{link}-tree. Naturally, the higher the degradation, the lower the performance. Eventually, a maintenance operation has to be performed to reorganize the tree so that the degradation level is zero again. With this concept, the modeling point of interest is to determine the degradation level d at which a maintenance operation should be performed so that the average response time per operation is mini-mum. Note that in this formulation, the time period in which the degradation level increases from 0 to d corresponds to the time interval between two successive maintenance operations. Before a maintenance operation is performed, all active database operations are allowed to complete, but during maintenance no new database operations can access the B^{link}-tree.

The service times of operations are, of course, affected by the degradation level of the tree because as the degradation level increases each operation will have to traverse more nodes at the leaf level to access a key. We model the real time (not the CPU time) elapsed to complete a search operation when the degradation level is j by an exponentially distributed random variable with an average of $T_r(j)$ time units. Similarly, $T_u(j)$ for an update operation; $T_d(j)$ for a delete operation; and $T_i(j)$ for an insert operation. Then, the rates at which search, update, delete, and insert operations are serviced at degradation level j are given by

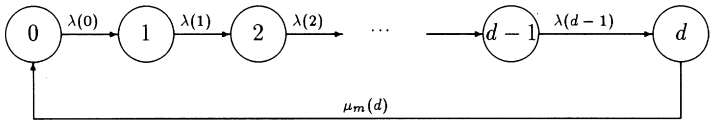

FIGURE 5. A performance model for periodic data reorganization on B^{link}-tree.

$\mu_r(j) = 1/T_r(j)$, $\mu_u(j) = 1/T_u(j)$, $\mu_d(j) = 1/T_d(j)$, and $\mu_i(j) = 1/T_i(j)$, respectively. These model parameters can be estimated from measurement data as we will illustrate in the Section 4.

Now we are interested in knowing how fast the degradation level is increased so that we may determine the optimal interval between two successive periodic maintenance operations. There are N concurrent operations (transactions) in the system in which qN are insert operations on average. Consequently, $qN\mu_i(j)$ is the rate at which insert operations are completed. Let $p_s(j)$ denote the probability that an insert operation splits a node at the leaf node when the degradation level is j. Then, the rate at which insert operations split nodes, or, equivalently, the rate at which the degradation level increases, when the degradation level is j, is given by $\lambda(j) = Nq\mu_i(j)p_s(j)$.

Assume that the maintenance time for performing the data reorganization when the degradation level is j is also an exponentially distributed random variable with an average of $T_m(j)$ time units or a rate of $\mu_m(j) = 1/T_m(j)$. The behavior of the B^{link}-tree with respect to the increase of the degradation level can be described by a Markov model shown in Figure 5 where the number in a circle represents the degradation level which increases from 0, 1, ..., d–1, to d at which point a maintenance is invoked to bring the degradation back to 0 again. The following defines the notation used in the paper:

- d: the optimal degradation level at which a maintenance of the B^{link}-tree should be performed so that the average response time per operation is minimum.
- q: the probability of an insert operation; it is also the probability of a delete operation.
- N: total number of concurrent operations (or transactions) accessing the B^{link}-tree.
- $\mu_r(j)$: $1/T_r(j)$, the *service* rate of a read operation when the degradation level of the B^{link}-tree is j.
- $\mu_u(j)$: $1/T_u(j)$, the *service* rate of an update operation when the degradation level of the B^{link}-tree is j.
- $\mu_d(j)$: $1/T_d(j)$, the *service* rate of a delete operation when the degradation level of the B^{link}-tree is j.
- $\mu_i(j)$: $1/T_i(j)$, the *service* rate of an insert operation when the degradation level of the B^{link}-tree is j.
- $\mu_m(j)$: $1/T_m(j)$, the *maintenance* rate of a maintenance operation when the degradation level of the B^{link}-tree is j.
- $p_s(j)$: the probability that an insert operation will cause a node to be split into two when the degradation level of the B^{link}-tree is j.
- $\lambda(j)$: $qN\mu_i(j)p_s(j)$—the split rate of leaf-level nodes when the degradation level of the B^{link}-tree is j.
- $P(j)$: the steady state probability that the degradation level of the B^{link}-tree is j.

By balancing flow into and out of each state j, $0 \leqslant j \leqslant d$, in Figure 5, we can obtain d independent global balance equations [5]. Solving these equations yields the steady state probability that the degradation level of the B^{link}-tree is j, i.e., $P(j)$, as

$$P(j) = \begin{cases} \dfrac{\lambda(0)}{\lambda(j)}P(0) & \text{if } 0 \leqslant j < d \\ \dfrac{\lambda(0)}{\mu_m(d)}P(0) & \text{if } j = d \end{cases} \tag{1}$$

Using $\sum_{j=0}^{d} P(j) = 1$, we get:

$$P(0) = \frac{1}{\dfrac{\lambda(0)}{\mu_m(d)} + \sum_{j=0}^{d-1}\dfrac{\lambda(0)}{\lambda(j)}}$$

and therefore

$$P(j) = \frac{\dfrac{\lambda(0)}{\lambda(j)}}{\dfrac{\lambda(0)}{\mu_m(d)} + \sum_{j=0}^{d-1}\dfrac{\lambda(0)}{\lambda(j)}} \quad \textit{for } 0 \leqslant j < d. \tag{2}$$

Based on the expression for $P(j)$ and assume that the probabilities of insert, delete, update and read operations are, say, q, q, $\frac{1-2q}{2}$ and $\frac{1-2q}{2}$, respectively, we can compute the throughput of the B^{link}-tree, X, as

$$X = \sum_{j=0}^{d-1} P(j)\left(qN\mu_i(j) + qN\mu_d(j) + \frac{1-2q}{2}N\mu_u(j) + \frac{1-2q}{2}N\mu_r(j)\right) \tag{3}$$

By Little's Law [7], the average response time per operation (transaction) is given as

$$R = NX \tag{4}$$

Note that the computation of X above excludes the case when the degradation level is d because the system is not doing useful work (i.e. servicing operations) during a maintenance period.

4. CASE STUDY

As a utility of the performance analysis given in the last section, we consider a detailed case study below. Consider a B^{link}-tree being used as an internal data structure to handle a rapidly changing database system for which each key's information is contained in a disk page pointed to by a leaf node in the B^{link}-tree. Suppose that they are $M = 1000$ keys being randomly accessed in

the system and the number of transactions concurrently accessing the database system is $N = 3$ which is just an arbitrary choice. Each transaction independently and continuously performs a delete-key operation with probability q, an insert-key operation with probability of also q (so that the number of keys remains the same in the steady state), a read-key operation with probability of $\frac{1-2q}{2}$ and an update-key operation also with probability of $\frac{1-2q}{2}$. The B^{link}-tree is to be maintained based on our periodic algorithm described in Section 2. The questions are (i) what would be the optimal maintenance period in terms of d and q? (ii) what would be the system throughput and response time under the optimal condition? To answer these questions, we illustrate below how we obtain the values of (i.e. parameterize) model parameters.

Table 1 shows the dynamic data of the B^{link}-tree as a function of the degradation level of the tree under the operational environment described above. These data are collected by running a single transaction process accessing the B^{link}-tree initially at $j = 0$. During the data collection period, the degradation level of the B^{link}-tree was allowed to increase as a result of insert operations that split the leaf nodes of the tree. Copies of the B^{link}-tree file were saved on disk at various degradation level checkpoints, i.e., at $j = 2, 4, 8, \ldots, 1024$. The data collection period was ended when the degradation level j had reached a specified target degradation level (at $j = 1024$). Since each such copy saved reflects the B^{link}-tree at a particular j value, we measured $T_r(j)$, $T_i(j)$ (which is the same as $T_d(j)$ and $T_u(j)$ in our algorithm), $T_m(j)$, and $p_s(j)$ statistically by simulating operations to access the corresponding B^{link}-tree copy. Specifically, to obtain $T_i(j)$, we simulated a sufficient number of read operations with random keys to access the B^{link}-tree copy saved earlier for that particular j value and obtained the average number of read pages per read operation, $n_r(j)$, from which the value of $T_i(j)$ is computed. Note that an insert operation which causes a leaf node to be split actually writes two nodes. This special case is considered when $\lambda(j)$, which deals with such insert operations, is computed.

$$\begin{aligned} T_i(j) = {} & \text{time for reading a disk page} \times n_r(j) \\ & + \text{time for writing a disk page} \times 1 \\ & + \text{time for locking a node} \\ & + \text{time for unlocking a node.} \end{aligned} \tag{5}$$

where we note that unlike reading a disk page, the multiply unit for writing, locking, or unlocking is one because each non-read operation in our algorithm only writes, locks and unlocks one node (1 disk page) of the B^{link}-tree. We used the same estimate for $T_u(j)$ and $T_d(j)$ since an update or a delete operation accesses the B^{link}-tree in a similar way as an insert operation.

Three critical points should be mentioned at this time. First, unlike $p_s(j)$ and $T_m(j)$ whose values are not affected by N (the number of concurrent transactions), $T_r(j)$, $T_i(j)$, $T_d(j)$ and $T_u(j)$ are, by definition, a function of N so as to include the effect of context-switch due to concurrent processing. Specifically, $T_i(j) = N \times T_i^*(j)$ (and similarly for others) where $T_i^*(j)$ is the average insertion time when there is only one transaction in the system; $T_i^*(j)$ can be computed by first measuring the times needed for reading, writing, locking and unlocking a disk page on the target machine by a single transaction process without context-switch, and then utilizing Equation 5. The data in Table 1 for $T_r^*(j)$ and $T_i^*(j)$ were obtained by following this computational procedure, i.e. we first measured the times required for reading, writing, locking and unlocking a disk page (the last two operations each involve a message passing) as 0.000195, 0.000686, 0.000066 and 0.000066 CPU seconds, respectively, by running a measurement program alone on the target machine (a SUN SPARC10 workstation); then, $T_r^*(j)$ and $T_i^*(j)$ at different j values were computed based on Equation 5. The second critical point that should be noted is that since $T_i(j) = N \times T_i^*(j)$, $\lambda(j)$, defined as $qNp_s(j)/T_i(j)$, can be computed as $qp_s(j)/T_i^*(j)$ to account for the context-switch overhead associated with running N concurrent transactions in the system. The third critical point is that all parameters listed in Table 1 can be easily *recomputed*. For example, when q or N changes, we can recompute $\lambda(j)'s$ easily without having to collect another set of data again. The last point facilitates 'what if' types of performance assessment on the projected B^{link}-

TABLE 1. B^{link}-tree data as a function of degradation level j.

j	$T_r^*(j)$	$T_i^*(j)$	$T_m^*(j)$	$p_s(j)$	$\lambda(j)$ *at* $q = 0.4$
0	0.002395	0.003213	0.434874	0.439000	45.041810
100	0.002644	0.003462	0.951877	0.192500	18.561723
200	0.002875	0.003693	1.542469	0.136000	12.423185
300	0.003093	0.003911	2.166713	0.109281	9.508783
400	0.003315	0.004133	2.825810	0.080375	6.670834
500	0.003538	0.004356	3.484906	0.051469	4.083275
600	0.003769	0.004587	4.332100	0.048172	3.653965
700	0.004002	0.004820	5.204944	0.048367	3.513726
800	0.004235	0.005053	6.077787	0.048563	3.384860
900	0.004467	0.005285	6.950631	0.048758	3.266037
1000	0.004700	0.005518	7.823475	0.048953	3.156128

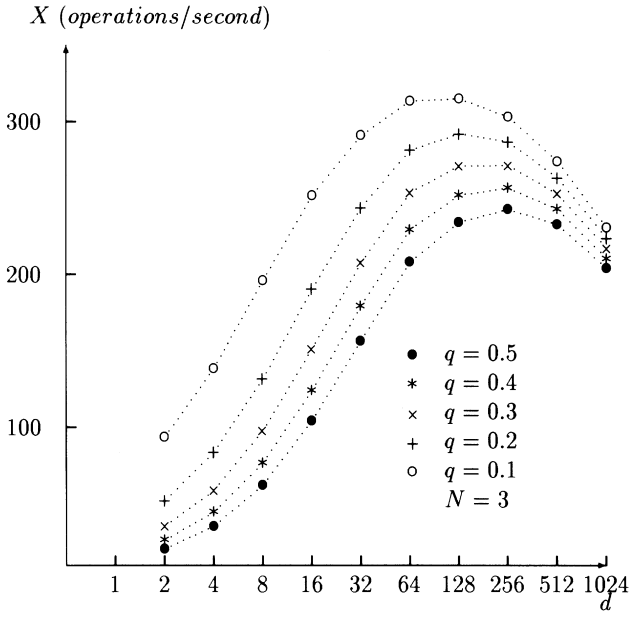

FIGURE 6. Throughput X as a function of d and q.

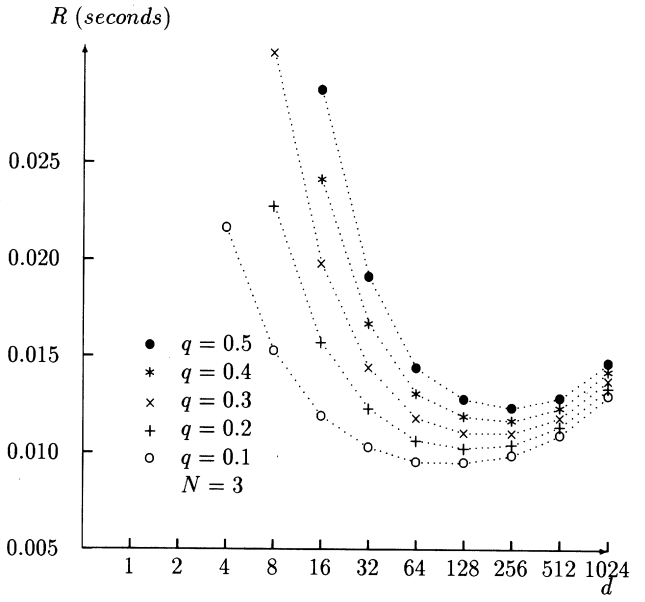

FIGURE 7. Response time R as a function of d and q.

tree data structures with periodic data reorganization technique.

Table 1 shows that $p_s(j)$ decreases while $T_i(j)$, $T_r(j)$ and $T_m(j)$ all increase as the degradation level j increases. This is expected because as j increases, more and more leaf nodes are being split which are likely to be less than two-thirds full and more and more internal nodes are not updated in the tree. As a result, the split probability ($p_s(j)$) becomes smaller and smaller and the access time per operation ($T_r(j)$ or $T_i(j)$) becomes higher and higher as j increases.

After the values of model parameters are obtained this way, the system throughput X and the average response time per operation R at different d values (at which a periodic maintenance operation is performed) are computed based on Equations 2, 3 and 4. The results are summarized in Figure 6 which gives the system throughput (number of operations completed per second) as a function of d and q, and in Figure 7, which gives the response time per operation, also as a function of d and q. These performance assessment results indicate that for the system described in the case study the maintenance operation should be invoked once when the degradation level of the system has accumulated to 128 or 256 (since the last maintenance operation has performed) so that the system throughput is optimized, almost for *all* q values.

There are two interpretations of the results. First, the response time per operation increases as q increases because at a higher q value, non-read operations occur more frequently than read operations but take more time to complete. Second, although the result shows that the maintenance operation should be performed when the degradation level is accumulated to 128 or 256 to optimize the system performance *for almost all q values*, the elapsed time interval between two consecutive invocations of the maintenance operation is actually different when q is different. The system will take a longer time to reach $d = 128$ at a smaller q value and conversely a shorter time at a higher q value because q determines how fast leaf nodes are split due to insert operations which occur with probability q. Therefore, Figures 6 and 7 actually show that for different q values, the optimal periodic maintenance intervals are different, in addition to the fact that the system throughput and response time are also different.

5. SUMMARY

In this paper, we have introduced the concept of periodic maintenance for improving the performance of B^{link}-trees by developing a new algorithm that modifies the insert operation of the classic LY algorithm so that all operations take about the same time to complete without having to maintain the internal nodes of the B^{link}-tree on-the-fly, thus leaving the maintenance work to a maintenance process which is invoked only periodically at optimizing intervals such that the amortized maintenance overhead per operation is minimized. A performance analysis was given and exemplified with a practical case study to determine the best maintenance interval between two consecutive invocations of the maintenance operation for optimizing the system performance. Such analysis technique is believed generally applicable to other database environments.

Some possible future research areas include (i) comparing the performance of B^{link}-trees with periodic maintenance and with concurrent maintenance and identifying conditions under which periodic maintenance is better than concurrent maintenance and vice versa; (ii) performing a similar analysis but considering other performance metrics such as space utilization or a mixed performance metric considering both space and time.

REFERENCES

[1] Bastani, F.B., Chen, I.R. and Hilal, W. (1991) A model for the stability analysis of maintenance strategies for linear list. *Comp. J.*, **34**, 80–87.

[2] Chen, I.R. and Banawan, S.A. (1992) A reduced Markov model for the performance analysis of data structure servers with periodic maintenance. *Comp. J.*, **35**, A363–A368.

[3] Chen, I.R. and Banawan,S.A. (1993) Modeling and analysis of concurrent maintenance policies for data structures using pointers. *IEEE Trans. Soft. Eng.*, **19**, 902–911.

[4] Comer, D. (1979) The ubiquitous B-tree. *Comp. Surveys*, **11**, 121–137.

[5] Kleinrock, L. (1975) *Queueing Systems, Vol. 1: Theory*, John Wiley, Chichester, pp. 155–156.

[6] Kung, H.T. and Lehman, P.L. (1980) Concurrent manipulation of binary search tress. *ACM Trans. Database Systems*, **5**, 354–382.

[7] Lazowska, E.D., Zahorjan, J., Graham, G.S. and Sevcik, K.C. (1984) *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice Hall, NJ.

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

[9] Manolopoulos, Y. (1994) B-trees with lazy parent split. *Information Sci.*, **79**, 73–88.

[10] Moitra, A., Iyengar, S.S., Bastani, F.B. and Yen, I.L. (1988) Multilevel data structures: models and performance. *IEEE Trans. Soft. Eng.*, **14**, 858–867.

[11] Sagiv, Y. (1986) Concurrent operations on B*-trees with overtaking. *J. Comp. Sys. Sci.*, **33**, 275–296.