Real-Time Index Concurrency Control

Jayant R. Haritsa

Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India
haritsa@serc.iisc.ernet.in

S. Seshadri

Dept. of Computer Science and Engineering
Indian Institute of Technology
Bombay 400076, India
seshadri@cse.iitb.ernet.in

1 Introduction

Research on real-time database systems (RTDBS) has been underway for close to a decade now. So far, this research has focused mainly on identifying the appropriate choice, with respect to meeting the real-time goals, for the various database system policies such as priority assignment, transaction concurrency control, memory management, etc. (see [16] for a recent survey). However, the design and evaluation of concurrency control algorithms for **indexes**, which are an integral part of database systems, has received virtually no attention in the real-time environment. In our view, this lacuna is rather surprising since it appears reasonable to expect that indexes would be one of the primary mechanisms by which a well-designed RTDBS would try to meet transaction timing constraints. In this situation, using an "off-the-shelf" index concurrency control algorithm that has not been specifically designed for the real-time environment may lead to many transactions missing their deadlines.

To address the above issue, we have initiated a project called **RICC** (Real-Time Index Concurrency Control), whose goals are to identify the basic forces that determine real-time index performance and to develop index concurrency control algorithms that are tuned to the real-time environment. This project is being jointly conducted at the Indian Insitute of Science (Bangalore) and at the Indian Institute of Technology (Bombay) in India. In this article, we present an overview of the results obtained so far and identify future research avenues.

2 Project Overview

Commercial database systems typically use **B$^+$-tree** indexing [3] as the preferred access method to efficiently locate data items on disk. For database systems supporting high transaction rates, the contention among transactions concurrently using the B-tree may itself form a performance bottleneck. To address this issue, a number of high-concurrency algorithms have been proposed for B-tree access (e.g. [1, 2, 8, 9, 10, 11, 14]). The performance of a representative set of these algorithms has recently been profiled in [7, 15] and their results indicate that **B-link** algorithms [10] provide the best performance over a wide range of workloads and system operating conditions.

The above-mentioned performance studies were done in the context of a conventional DBMS where transaction throughput or response time is the primary performance metric. In a real-time database system, however, performance is usually measured in terms of the *number* of transactions that complete before their deadlines. That is, a transaction that completes just before its deadline is no different, from a performance perspective, to one that finishes much earlier. Due to the difference in objectives, the performance of index concurrency control algorithms has to be reevaluated for the real-time domain.

Another important difference between our study and those of [7, 15] is that transactions consist of *multiple* index actions in our model. In contrast, the earlier studies modeled "tree" transactions wherein each transaction performs only a single B-tree operation (search or update). The tree model is not appropriate for the real-time environment since the metric of missed deadlines is meaningful only when applied to complete transactions. In our model of an RTDBS, therefore, transactions are capable of performing multiple index operations and mechanisms for ensuring transaction serializability are implemented.

There are two major issues that need to be explored with regard to real-time index concurrency control: First, how do we adapt the concurrency control protocols to the real-time domain? Second, how do these real-time variants compare in their performance? In our initial research work, we have addressed these questions for the "firm-deadline" [6] application framework, wherein transactions that miss their deadlines are considered to be worthless and are immediately discarded from the system without being executed to completion. We have developed real-time variants of several classical B-tree concurrency control algorithms and compared their performance using a detailed simulation model of a real-time database system. The performance metric is the steady-state percentage of transaction deadlines that are missed by the system.

3 B-tree CC Algorithms

In this section, we briefly describe the set of B-tree concurrency control algorithms considered in our study. We assume, in the following discussion, that the reader is familiar with the basic features and operations of B-tree index structures [3, 15].

We consider three algorithms in the **Bayer-Schkolnick** class [2] called B-X, B-SIX and B-OPT, respectively. In all these algorithms, readers descend from the root to the leaf using lock-coupling with IS locks. They differ, however, in their update protocols: In **B-X**, updaters lock-couple from the root to the leaf using X locks. In **B-SIX**, updaters lock-couple using SIX locks in their descent to the leaf. On reaching the leaf, the SIX locks in their scope are converted to X locks. In **B-OPT**, updaters make an *optimistic* lock-coupling descent to the leaf using IX locks. The descent is called optimistic since, regardless of safety, the lock at each level of the tree is released as soon as the appropriate child has been locked (i.e., it is optimistically assumed that the leaf will not need to be split). After the descent, updaters obtain a X lock at the leaf level and complete the update if the leaf is safe. Otherwise, the update operation is restarted, this time using SIX locks.

In the **Top-Down** class of algorithms (e.g. [9, 12]), readers use the same locking strategy as that of the Bayer-Schkolnick algorithms. Updaters, however, perform *preparatory* splits and merges during their index descent: If an inserter encounters a full node it performs a preparatory node split while a deleter merges nodes that have only a single entry. This means that unlike updaters in the Bayer-Schkolnick algorithms who essentially update the entire scope at one time, the scope update in Top-Down algorithms is split into several smaller, atomic operations.

We consider three algorithms in the Top-Down class called TD-X, TD-SIX and TD-OPT, respectively: In **TD-X**, updaters lock-couple from the root to the leaf using X locks. In **TD-SIX**, updaters lock-couple using SIX locks. These locks are converted to X-locks if a split or merge is made. In **TD-OPT**, updaters lock-couple using IX locks in their descent to the leaf and then get an X lock on the leaf. If the leaf is unsafe, the update operation is restarted from the index root, using SIX locks for the descent.

Another class of index concurrency control algorithms is based on *B-link* trees. A B-link tree [9, 10] is a modification of the B-tree that uses links to chain together all nodes at each level of the B-tree. Specifically, each node in a B-link tree contains a high key (the highest key of the subtree rooted at this node) and a link to the right sibling. These links are used to split nodes in two phases: a half-split, followed by the insertion of an index entry into the appropriate parent. Operations arriving at a newly split node with a search key greater than the high key use the right link to get to the appropriate node. Such a sideways traversal is called a *link-chase*. Merges are also done in two steps, via a half-merge followed by the appropriate entry deletion at the next higher level.

In the **B-link** class of algorithms, readers and updaters *do not* lock-couple during their tree descent. Instead, readers descend the tree using IS locks, releasing each lock *before* getting a lock on the next node. Updaters also behave like readers until they reach the appropriate leaf node. On reaching the leaf, updaters release their IS lock and then try to get an X lock on the same leaf. After the X lock is granted, they may either find that the leaf is the correct one to update or they have to perform link-chases to get to the correct leaf. Updaters use X locks while performing all further link chases, releasing the X lock on a node before asking for the next. If a node split or merge is necessary, updaters perform a half-split or half-merge. They then release the X lock on the leaf and propagate the updates, using X locks, to the higher levels of the tree. We have considered only one B-link algorithm, referred to as **LY** algorithm in [15], which exactly implements the above description.

4 Real–Time Index CC

In a real-time database system, the resource scheduling policies can be reasonably expected to be priority-driven with the priority assignment scheme being tuned to minimize the number of missed deadlines. The index concurrency control algorithms described above do not take transaction priorities into account. This may result in the undesirable phenomenon of high priority transactions being blocked by low priority transactions, a phenomenon known as *priority inversion* in the real-time literature.

To address the above problem, we have incorporated priority into the index CC algorithms in the following manner: When a transaction requests a lock on an index node that is held by higher priority transactions in a conflicting lock mode, the requesting transaction waits for the node to be released (the wait queue for an index node is maintained in priority order). On the other hand, if the index node is currently held by only lower priority transactions in a conflicting lock mode, the lower priority transactions are *preempted* and the requesting transaction is awarded the lock. The lower priority transactions then restart, from the beginning, their *current* index operation (not the entire transaction).

5 Simulation Model and Methodology

In the previous section, we discussed various index concurrency control algorithms and their real time versions. To evaluate the performance of these algorithms, we developed a detailed simulation model of a firm-deadline real-time database system. The organization of our model is based on a loose combination of the database model of [6] and the B-tree system model of [15]. A summary of the parameters used in the model are given in Table 1.

Table 1: Model Parameters

Parameter	Meaning	Value
ArrRate	Transaction arrival rate	$0 - \infty$
TransSize	Average transaction size	8
SlackFactor	Deadline Slack Factor	4
SearchProb	Proportion of searches	0.0 – 1.0
InsertProb	Proportion of inserts	0.0 – 1.0
DeleteProb	Proportion of deletes	0.0 – 1.0
AppendProb	Proportion of appends	0.0 – 1.0
InitKeys	No. of keys in initial tree	100,000
MaxFanout	Key entries per node	300
NumCPUs	Number of processors	$1 - \infty$
SpeedCPU	Processor MIPS	20
LockCPU	Cost for lock/unlock	1000 inst.
LatchCPU	Cost for latch/unlatch	100 inst.
BufCPU	Cost for buffer call	1000 inst.
SearchCPU	Cost for page search	500 inst.
ModifyCPU	Cost for key insert/delete	500 inst.
CopyCPU	Cost for page copy	1000 inst.
NumDisks	Number of disks	$1 - \infty$
PageDisk	Disk page access time	20 ms
NumBufs	Size of buffer pool	$1 - \infty$

Transactions arrive in a Poisson stream and each transaction has an associated deadline. A transaction consists of a sequence of index access operations such as search, insert, delete or append of a key value. After each index operation, the corresponding data access is made. A transaction that is restarted due to a data conflict makes the same index accesses as its original incarnation. If a transaction has not completed by its deadline, it is immediately aborted and discarded.

The *ArrRate* parameter specifies the mean rate of transaction arrivals. The number of index accesses made by each transaction varies uniformly between half and one-and-a-half times the value of *TransSize*. The overall proportion of searches, inserts, deletes and appends in the workload is given by the *SearchProb*, *InsertProb*, *DeleteProb* and *AppendProb* parameters, respectively. All index search and update operations are point (single key) operations, as in [7, 15].

Transactions are assigned deadlines with the formula $D_T = A_T + SF * R_T$, where D_T, A_T and R_T are the deadline, arrival time and resource time, respectively, of transaction T, while SF is a slack factor. The *resource time* is the total service time at the resources that the transaction requires for its data processing. The *slack factor* is a constant that provides control over the tightness/slackness of deadlines.

A single B-tree is modeled and all transaction index accesses are made to this tree. The initial number of keys in the index is determined by the *InitKeys* parameter. Each index node corresponds to a single disk block and the *MaxFanout* parameter gives the node key capacity.

The physical resources in our model consist of processors, memory and disks. There is a single queue for the CPUs and the service discipline is preemptive-resume, with preemptions being based on transaction priorities. Each of the disks has its own queue and is scheduled with a priority Head-of-Line policy. Buffer management is implemented using a simple two-level priority LRU mechanism wherein higher priority transactions steal buffers from the lowest priority transaction that currently owns one or more buffers in the memory pool. The *NumCPUs*, *NumDisks* and *NumBufs* parameters quantitatively determine the resource configuration. The processing cost parameters for each type of index operation are also given in Table 1.

6 Results

We performed several experiments using a simulator (written in C++) that implemented the above simulation model. The transaction priority assignment scheme used in these experiments was *Earliest Deadline*: transactions with earlier deadlines have higher priority than transactions with later deadlines. The performance metric of our experiments is *Miss Per-*

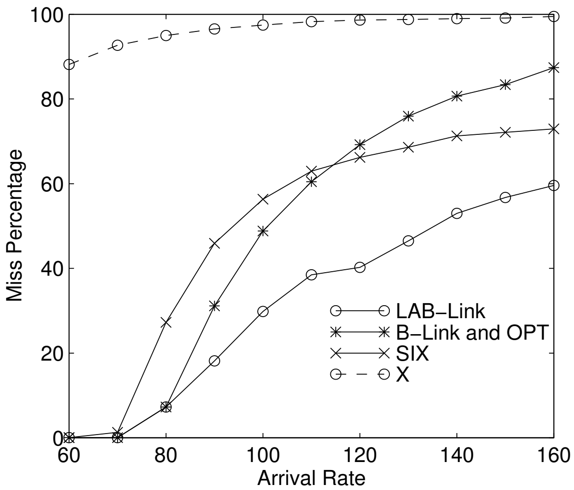

Figure 1: Real-Time Index Performance

cent, which is the percentage of input transactions that the system is *unable* to complete before their deadlines. A detailed description of these experiments is available in [4]. Here, we present the results for one representative experiment, wherein the workload consisted of 80% searches, 10% inserts and 10% deletes, and the the resource parameter settings were: $NumCPUs = 1$, $NumDisks = 8$, and $NumBufs = 250$.

No appreciable difference was observed between the performance of the corresponding algorithms from the Bayer-Schkolnick and Top-Down classes (i.e. between B-OPT and TD-OPT, B-X and TD-X, B-SIX and TD-SIX); we will therefore simply use OPT, SIX and X to denote these algorithms in the following discussion.

For this experiment, Figure 1 shows the miss percent behavior as a function of transaction arrival rate. The poor behavior of the X algorithms is due to the root of the index tree becoming a bottleneck, resulting in large lock waiting times. In the corresponding (non-real-time) experiment in [15], the B-link algorithms performed much better than the SIX algorithms. However, in our case, though B-link performs the best for low arrival rates, the situation is reversed at high arrival rates where the SIX algorithms are found to outperform the B-link algorithms. The explanation for this counter-intuitive performance is as follows: The SIX algorithms give preferential treatment to read index operations over update index operations (by allowing readers to overtake updaters during tree traversal) [15]. This leads to SIX providing commensurately better performance for transactions that have either no updates or only a few updates. In [15], for the SIX algorithms, the slower updaters used to clog up the system, resulting in much higher contention levels and poor performance – in the firm real-time environment, however, that does not happen because transactions are *discarded* as soon as their deadlines expire. Therefore, this type of clogging is inherently prevented.

If we view the above result from a different angle, we observe that the SIX algorithms, by giving preferential treatment to read only transactions, are applying a form of *load control*. On the other hand, the B-link algorithms tend to *saturate* the disk due to treating all transactions uniformly and therefore miss significantly more deadlines than SIX. This naturally suggests that the performance of the B-link algorithms could be improved by adding a load-control component *without* sacrificing their desirable fairness feature.

To evaluate the potential of the above idea, we developed a variant of the B-link algorithm called **LAB-link** (Load Adaptive B-link) which ensures that the utilization of the bottleneck resource is not allowed to exceed acceptable levels. This is achieved through a simple feedback mechanism that monitors the utilization at all the system resources and prevents new transactions from entering the system whenever the utilization of the bottleneck resource exceeds a prescribed amount. Transactions which are denied entry are eventually discarded when their deadlines expire and, for the miss percent computation, are considered to be transactions that have missed their deadlines. The details of the feedback mechanism are available in [4].

The performance of the LAB-link algorithm is shown in Figure 1. It is clear from this graph that the overload performance of LAB-link improves dramatically over B-link. As explained above, this is due to its admission control policy which ensures that the bottleneck resource (in this case, the disk) does not become saturated, thereby comfortably completing the admitted transactions.

The performance of the OPT algorithms is almost identical to that of B-link algorithms since the number of splits and merges is very low for the workload considered in this experiment.

7 Summary

The goal of the RICC project is to identify index concurrency control algorithms that are tuned to the real-time environment. This is being done through extensive experimentation on a detailed simulation model of an RTDBS. Currently, the deadline miss percent performance of real-time variants of three different classes of index CC algorithms: Bayer-

Schkolnick, Top-Down and B-link, have been evaluated under a range of workloads and operating conditions. A detailed description of the experiments is available in [4].

The experimental results show that two factors characteristic of the (firm) real-time domain: addition of priority and discarding of late transactions, significantly affect the performance of the index concurrency control algorithms. In particular, B-link algorithms miss many more deadlines at high loads as compared to lock-coupling algorithms. This is in contrast to conventional DBMS where they (B-link) always exhibited the best throughput performance. In fact, the very reason for their good performance in conventional DBMS (full resource utilization) turns out to be a liability here. Secondly, the optimistic algorithms perform almost as well as the B-link algorithms even under high index contention conditions (in contrast to conventional DBMS). This is because prioritization of transactions caused a marked decrease in the number of index operation restarts. In short, these experiments show that the performance behaviors exhibited by index CC algorithms in conventional DBMS cannot be blindly assumed to be valid in the corresponding real-time situation also.

We introduced the LAB-link algorithm, which augments the basic B-link algorithm with a simple load-control system to ensure that the bottleneck resource is not saturated. The LAB-link algorithm significantly reduces the miss percentage of B-link in the overload region and thereby provides the best performance over the entire loading range. This clearly demonstrates the need for load control in index management in real-time database systems. Interestingly, such need for load control has also been identified in other modules of real-time database systems (e.g. [5, 13]).

Our study has so far been limited to point (single key) index operations. In our future work, we plan to extend our study to include range (multiple key) operations. In its current implementation, LAB-link uses a utilization-based load control. We also plan to evaluate the performance that would be obtained by implementing load-control through priority assignments (as done, for example, in Adaptive Earliest Deadline [5]).

Acknowledgements

Brajesh Goyal of Indian Institute of Technology (Bombay) and V. Srinivasan of IBM Santa Teresa Laboratory have collaborated in the research work summarized in this article. The work of J. R. Haritsa was supported in part by a research grant from the Dept. of Science and Technology, Govt. of India.

References

[1] A. Biliris. A comparative study of concurrency control methods in B-trees. In *Proc. of Agean Workshop on Computing*, Loutraki, July 1986.

[2] R. Bayer and M. Schkolnick. Concurrency of Operations on B-Trees. In *Acta Informatica*, 9, 1977.

[3] D. Comer. The Ubiquitous B-Tree. In *ACM Computing Surveys*, 11(4), 1979.

[4] B. Goyal, J. Haritsa, S. Seshadri, and V. Srinivasan. Index Concurrency Control in Firm Real-Time DBMS. In *Proc. of VLDB Conf.*, September 1995.

[5] J. Haritsa, M. Carey, and M. Livny. Earliest Deadline Scheduling for Real-Time Database Systems. In *Proc. of IEEE Real-Time Systems Symp.*, December 1991.

[6] J. Haritsa, M. Carey and M. Livny. Data Access Scheduling in Firm Real-Time Database Systems. In *Journal of Real-Time Systems*, September 1992.

[7] T. Johnson and D. Shasha. A framework for the performance analysis of concurrent B-tree algorithms. In *Proc. of ACM Symp. on Principles of Database Systems*, April 1990.

[8] Y. Kwong and D. Wood. A new method for concurrency in B-trees. *IEEE Trans. on Software Engineering*, SE-8(3), May 1982.

[9] V. Lanin and D. Shasha. A Symmetric Concurrent B-tree Algorithm. In *Proc. of Fall Joint Computer Conf.*, 1986.

[10] P. Lehman and S. Yao. Efficient Locking for Concurrent Operations on B-trees. In *ACM Trans. on Database Systems*, 6(4), 1981.

[11] C. Mohan and F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method using Write-Ahead Logging. In *Proc. of ACM SIGMOD Conf.*, June 1992.

[12] Y. Mond and Y. Raz. Concurrency Control in B^+-trees Databases Using Preparatory Operations. In *Proc. of VLDB Conf.*, September 1985.

[13] H. Pang, M. Carey and M. Livny. Managing Memory for Real-Time Queries. In *Proc. of ACM SIGMOD Conf.*, May 1994.

[14] D. Shasha and N. Goodman. Concurrent search structure algorithms. In *ACM Trans. on Database Systems*, 13(1), March 1988.

[15] V. Srinivasan and M. Carey. Performance of B-Tree Concurrency Control Algorithms. In *Proc. of ACM SIGMOD Conf.*, May 1991.

[16] Ulusoy, O. Research Issues in Real-Time Database Systems. *Tech. Report BU-CEIS-94-32*, Dept. of Computer Engg. and Information Science, Bilkent University, Turkey, 1994.