Simplified Predicate Locking Scheme for Concurrency Control on R-tree

*Ying Xia[1], Kee-Wook Rim[2], Jae-Dong Lee[3], Hae-Young Bae[1]
[1]Dept. of Computer Science & Engineering, Inha University
[2]Dept. of Industrial Engineering, Sunmoon University
[3]Dept. of Computer Science, Dankook University
g1992131@inhavision.inha.ac.kr

Abstract

Despite extensive research on R-trees, most of the proposed schemes have not been integrated into existing DBMS due to the lack of protocol to provide consistency in concurrent environment. R-link tree is an acceptable data structure to deal with this issue, but still not enough. In this paper, we focus on a simplified predicate locking mechanism based on R-link tree for concurrency control and phantom protection. An in-memory operation control list (OCList) used to suspend some conflicting operations is designed here. The main features of this approach are (1) it can be implemented easily and do not need any extra information. (2) Only-one-lock is held when descending R-tree even when node split happens, while lock-coupling scheme is performed when ascending. No deadlocks are possible. (3) Searches and insertions are not unnecessarily restricted. (4) Insert and Delete phantom in R-link tree are avoided through beforehand predication.

1 Introduction

As one of the most important multidimensional data structures, a several variants of R-trees has been proposed for a rather long time, but most of them have not been used in existing DBMS. The main reason is the lack of applicable protocol to guarantee the consistency in the presence of concurrent operations. Some proposed schemes are too complex and require special definitions on storage manager [1,8].

Concurrency control on R-tree is much more difficult than on B-tree variants. Some well-designed concurrency control schemes such as ARIES/IM are not suitable for multidimensional indexing [1,4]. R-link tree is an acceptable data structure to deal with this issue but it is still not enough to resolve all the problems appeared during concurrent executions. 1) It self has some restrictions and some special cases are not considered in details, such as multiple insertions, and delete algorithms. 2) It cannot resolve the phantom problem [3,8]. Our scheme is based on R-link tree structure, at the same time, a simplified predicate locking approach using an in-memory operation control list (OCList) is designed to delay only conflicting operations, and resolve phantom problems.

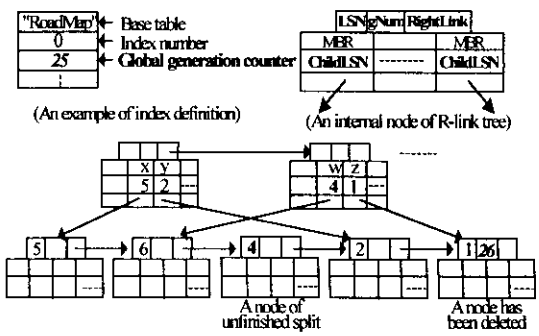
The rest of the paper is organized as follows. Section 2 reviews the related work about R-link tree. Our supporting data structure OCList is designed in Section 3. In Section 4 we discuss the relative operation algorithms based on R-link tree and OCList. At last, we show performance analysis and conclusions.

2 Related Work

R-link tree based on R-tree and link technique, is proposed by Leman and Yao in [3]. Extra information are designed in this structure, 1) each node is assigned a *LSN* (Logical Sequence Number) to determine whether an unfinished split exists during descending and thus decide how to move through the tree. 2) A *global generation counter* for the whole tree and a generation number (*gNum*) of each node determine whether a node is deleted or not. 3) A *RightLink* of each node is used for compensating unfinished node splits. 4) Each entry of internal nodes keeps the LSN of associated child node by *ChildLSN* [3,8]. See Figure 1.

Whenever a node splits, the new right sibling is assigned the old node's LSN and the old node receives a new greater LSN. We can always find the child node with the same LSN as ChildLSN in an entry of parent node by following right-link. For each split, after upward propagating, a corresponding entry can be found in parent level even if splits have also happened on parent nodes [3,8].

When a node is deleted, the tree global counter is incremented and the new value is assigned to the deleted node. Before descending, remember the global generation counter when get the pointer of one node and compare it to the generation number of that node when finally visiting it, a higher number in that node indicates that it must have been removed by other operations after the pointer was read. In this case, descending operation has to be restarted from the lowest valid ancestor node [3].



(Figure 1. A possible State of a Subsection of R-link Tree)

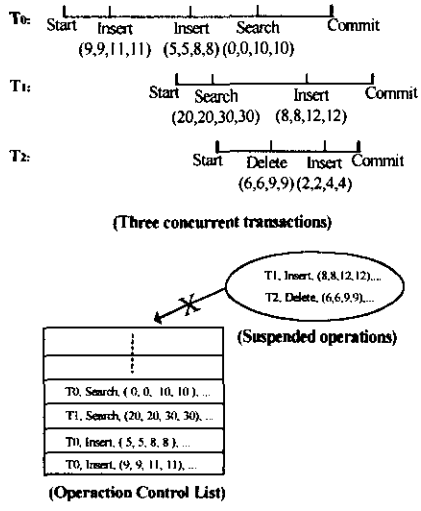
* This Research is supported by Ministry of Information and Communication under work of university S/W research center.

During tree descendent, only one node needs to be locked at any time. But during tree ascent, lock-coupling scheme is used to avoid upward propagated information are overtaken by other update operations and remaining the tree structure inconsistent. No deadlock is possible by using this locking strategy [3].

3 Supporting data structure OCList

There are at least two reasons that why we need an aid structure for concurrency control. Firstly, R-link tree allows multiple insertions are executed concurrently, but it does not overcome phantom [3]. When an insertion commits, the new key might be visible to a re-scan. Secondly, as for deletion, finding leaf node which includes the key we want to delete is not as simple as finding proper leaf node to insert, because the geometrically optimal leaf for insertion has only one, but the leaf nodes which intersect the deleted key might be more than one. Before the proper path from root to leaf been found, no structure modifications are expected.

The extent of concurrent level and the algorithm complexity is a tradeoff relation. Since we choice to implement a simplified scheme, some conflicting operations from different transactions need to keep waiting. The OCList is a global in-memory array structure designed to serialize all conflict operations. We give an example based on three concurrent transactions and show a possible state of OCList in Figure 2.



(Figure 2. A Possible State of Operation Control List)

Before an operation is performed, first lock OCList and check whether this operation is conflict with any other active ones. If not, a corresponding element includes transaction identify, operation type, region or key, and start time will be added in before OCList released. Otherwise, this operation is delayed.

When this operation committed, lock OCList again, and release it after removing the relative element. The start time field is used to assure that no any operations which has canceled due to transaction aborts are still exist in OCList. If some elements have exceeded the response time limitation, they will be deleted whenever any other operation checks OCList.

4 Concurrent operation algorithms using OCList

Basic algorithms about R-link trees can be consulted in [3,8]. In this paper, we focus on control approaches on OCList when search, insert, and delete are requested.

4.1 Search

According to the general usage of applications, given a query window and a query predication, the results will be processed through cursor one by one. So search algorithm includes FindFirst and FindNext functions, both of them return one record every time. One FindFirst and a serial of FindNext are invoked during search until there are no any other records satisfied. Checking OCList and adding element are performed at the beginning of FindFirst. Just like algorithm in R-link tree, a stack is used to keep all yet-to-be visited node, and descending tree is performed according to the rule of R-link tree. When search is finished, the relative operation element is deleted from OCList.

To avoiding phantom, there should no any insertions (or deletions) with a key falling in the new search rectangle are still active, otherwise uncommitted insertion (or deletion) might make query results different when re-scan. So a search request would check the OCList, if any uncommitted insertion keys fall in the query range, the search will suspend itself and keep rechecking. When all colliding insertion (or deleteion) committed, the search will be activated. Figure 3 shows the pseudo-code of OCList checking for search.

```
/*FindFirst(TransID,MBR)*/
/*check*/
Lock(OCList)
IF (no keys of insertion or deletion of other transactions
fall in MBR) THEN
  Add element (Trans,Search,MBR,..) in OCList
  Release OCList
ELSE
  Release OCList
  Sleep and Recheck
/*Find the first one*/
IF (FindFirst(TransID,MBR)==FALSE) THEN
  Lock(OCList)
  Remove element (Trans,Search,MBR,..) from OCList
  Release(OCList)

/*FindNext(TransID,MBR)*/
/*Find next one*/
IF (FindNext(TransID,MBR)==FALSE) THEN
  Lock(OCList)
  Remove element (Trans,Search,MBR) from OCList
  Release(OCList)
```

(Figure 3. Pseudo-Code for OCList Checking of Search)

4.2 Insert

For a given key and record identify, we need to locate the geometrically optimal leaf and remember the access path first. After insert new key into leaf, we must propagate the changes upward until a parent node does not need to be changed. If the leaf was split, a new entry should be installed in the parent node, it is also full, node split recursively performed until a node with enough free space arrived or alternatively split the root.

According to the lock scheme in R-link tree, descending in R-link tree only need lock one node, while ascending the tree lock-couple will be performed. This completely avoid deadlock,

and allow concurrently multiple insertions.

But insertion should be suspended if the new key falls in the specified ranges of any active searches in the system, otherwise this uncommitted insertion might cause phantom. Before insert, OCList is checked to confirm that there is no any colliding active search, and no deletion is finding leaf. If so, add an element about this insertion into OCList. After insertion committed, remove the relative element.

4.3 Delete

Similar with Insert, deletion needs to first find the leaf node in which includes the key we want to delete and keep the access path, then delete it and propagate the changed MBR upward. But find leaf node and remember the path for delete is much more difficult than for insert, because if when we arrived a leaf but the key is not in there, we have to back up to parent level and continue finding. To let our algorithm simple, before find the proper leaf node, we don't wish the tree structure modified by any other operations.

In our approach, deletion is somewhat pessimistic. Finding leaf node is only performed after confirm that no uncommitted insertions or deletions and intersected active search still exist by checking OCList and add relative information in it. After deletion committed, remove the relevant element.

4.4 Update

Update is modeled by the deletion of the original object followed by the insertion of the new object. We allow the indexing attributes been modified, and even new object with the same key as the original one will be relocated in the tree.

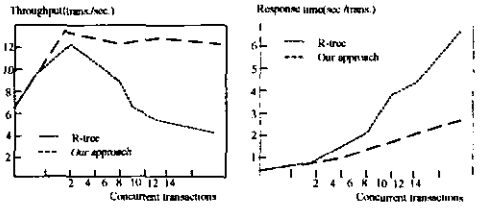
5 Performance

Performance show in [3] only performs comparisons about insert and search operations, no delete operations are considered. So we compare the performance of the proposed scheme with R-tree which allowing only multiple searches in terms of throughput and response time when considering search, insertion and deletion. Each transaction is a set of 100000 accessing operations in which 40% are searches, 30% are insertions, and the others are deletions. And we also compare these two terms with R-link tree when only apply 50% searches and 50% insertions in one transaction. We experimented with two-dimensional map data. The platform is Intel Pentium 700, 256MB RAM, 20GB hard disk, Windows 2000. As Figure 4.1 depicts, the experimental results show that our proposed scheme achieves better performance gains on both response time and throughput over R-tree on the average when all kinds of operations are included in a transaction. And as Figure 4.2 shows, this scheme gains similar performance with R-link tree when apply only insertions and searches by using simple algorithms and more details are considered.

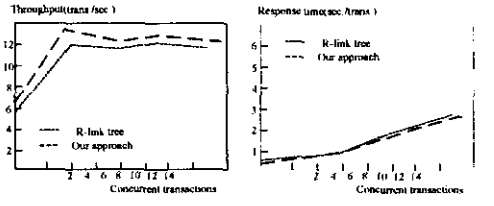
6 Conclusions

By using this strategy, 1) the average throughput is keep higher and rather constant as the number of transactions increasing. 2) The average response time is shorter than that of R-tree, which means we can serve more users at a means response level. And 3) the response time values are distributed in a narrow value range than that of R-tree, which means transactions do not block each other for a long time.

Algorithms proposed in this paper are easy to implement, but since delete operations in our approach is a little pessimistic, this scheme is especially suitable for applications with less deletions and updates. And because OCList is an in-memory structure, it is unlikely to keep too much elements of uncommitted update operations for long update transactions due to the space limitation.



(Figure 4.1 Average throughput and response time compare with R-tree when search, insert and delete are applied)



(Figure 4.2 Average throughput and response time compare with R-Link tree when search and insert are applied)

References

[1] K.Chakrabarti and S.Mehrotra. Dynamic Granular Locking Approach to Phantom Protection in R-trees. In Proc. of International Conf. on Data Engineering. Feb. 1998

[2] A.Guttman. A Dynamic Index Structure for Spatial Searching. In ACM SIGMOD Conf. 1984

[3] M.Kornacker and D.Banks. High-Concurrency in R-trees. In Proc. 21th International conference on VLDB. 1995

[4] P.L.Leman and S.B.Yao. Efficient Locking for Concurrent Operation on B-Trees. ACM TODS, 1981

[5] C.Mohan. ARIES/KVL: Akey-Value Locking Methiod for Concurrency Control of Multiaction Transactions Operationg on B-Tree Indexes. In IBM Research Report, 1989

[6] C.Mohan and F.Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In IBM Research Report, 1989

[7] V.Ng and T.Kameda. Concurrent Accesses to R-trees. In Proc. of Symposium on Large Spatial Database , 1993

[8] S.Song, S.H.Lee, J.S.Yoo and J.S.Lee. An Efficient Concurrency Control Algorithm for High-Dimensional Index Structures. In Proceedings of the International Conf.on Information Intelligence and Systems. 1999

[9] T.Sellis, N.Roussopoulos and C.Faloutsos. The R+-tree: A Dynamic Index for Multidimensional Objects. In Proc. 13th International Conf. On VLDB, 1987