Just-in-time Index Reference Updates

Chendong Zou
921 S.W. Washington Ave, Suite 670
Portland, OR97205
email: zou@informix.com
fax: 503-221-2633

Abstract

In this paper, we present an efficient method for updating index references. The basic idea is to defer index reference updates as much as possible and piggyback the pending updates with other user transactions. Not only is our method efficient in terms of number of I/Os, but it is also safe. Our method also pays attention to system issues that are often overlooked such as concurrency and recovery issues.

1 Introduction

Normal database operations such as insertions, deletions and updates of records generally involves updating the corresponding references in the indexes. In a typical database implementation, the operation (insert, delete, and update) and the corresponding reference updates are encapsulated and done at once as an atomic operation. Since a record can be referenced in more than one indexes, this usually implies multiple I/Os for index reference updates.

In order to decrease the number of I/Os for index reference updates, we present a new method that defers index references updates, and does the update just-in-time – piggybacking the updates with other user transactions' I/O. Our method guarantees that the database buffer is always consistent.

Section 2 will presents some data structures that are used by our method. It will also describe how do we defer index updates. The just-in-time reference updates logic is discussed in section 3. Section 4 will briefly describe some system considerations such as concurrency and recovery with our method. Finally section 5 concludes the paper with some future research directions.

2 Deferred Updates

The basic idea of our method is to do index updates as much as we can without doing any disk I/Os. Then if there are still updates that need to be done, we defer them and put them in some

in-memory tables. These tables are:

- **Pending Changes Table Q_is**

  Tables Q_i, $i = 1, \ldots, s$, where s is the number of secondary indexes, are used to keep information about the changes that should be made to secondary indexes. An entry in Q_j has the format (*Index_value*, *Page_number*, *Old-ID*,*New-ID*, *tid*). *Old-ID* is the old record identifer before the operation, and *New-ID* is the new record identifer after the operation. If the operation is an insertion, then *Old-ID* is *NULL*. If the operation is a deletion, then *New-ID* is *NULL*. *tid* is the transaction ID of the operation. Both *New-ID* and *Old-ID* are not *NULL* if the operation is a record move. The *Index_value* field is the index value of the record for that secondary index. The *Page_number* field is the page number of the secondary index page which either stores the *Index_value* (leaf page) or has a descendant which stores the *Index_value* (upper level index page). Table Q_is are hash tables. They are accessed by hashing on *Page_number*.

- **Transaction Table** T Transaction table T keeps track of all transactions that have pending updates in one of the Q_i tables. An entry of T consists of a transaction ID and a list of pointers to Q_i table entries. T is also a hash table (hash on *tid*).

With these lookup tables, we can outline our algorithm for deferring updates:

1. Check to see if the look-up tables are full. If they are, do clean-up for those tables.
2. Obtain all the locks necessary for the operation. These also include locks on the secondary index leaf pages that need to be updated.
3. Do the operation (insert, delete, update).
4. For each secondary index leaf page that needs to be updated:
   (a) If the page is in the buffer, make the update and log the change.
   (b) Otherwise, put the change in the look-up table and update the corresponding entry in the transaction table.

Figure 1: Algorithm for Deferred Reference Updating

Figure 1 illustrates the main idea of the algorithm. We first check if there are still rooms for deferred updates in the in-memory tables. If there isn't any room, then we would do cleanup. That is, we would bring in index pages and finish those pending updates to clean up the in-memory tables.

After the possible cleanup step, we get all necessary locks and do the operation. For each secondary index leaf page in which a reference should be updated, if the page is already in the buffer, we make the update and log it. If the page is not in the buffer, we record the update in the look-up tables. The locks shall be released after the corresponding transaction is finished. After each operation, the database buffer is *consistent*. That is, all index leaf pages in the buffer have correct references to records.

3 Just-in-time Updates

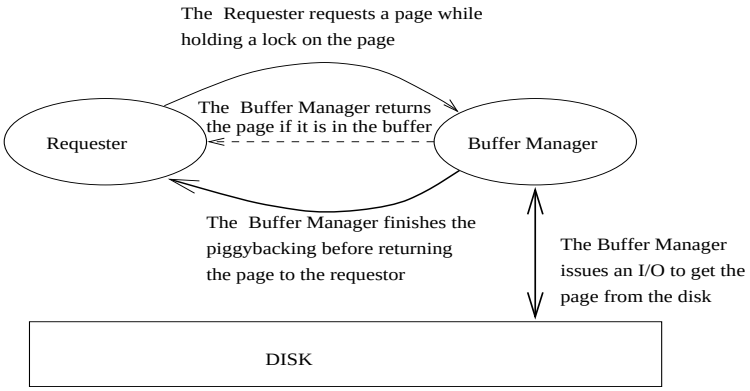

Figure 2: Just-in-time Updates

Figure 2 shows the interactions between the requester and the buffer manager, and how just-in-time updates are done. The dotted line represents the buffer manager's action when the requested page is already in the buffer, the bold line represents the buffer manager's actions when the requested page is not in the buffer. The idea here is that the database buffer manager will do the index reference updates just-in-time before anyone else access the page that contains pending updates. When the requester gets the page, the page already has the correct content. So the database buffer is *always consistent*. When all pending changes of a transaction have been done, its entry in the transaction table T shall be removed.

Notice here that the just-in-time update is page-oriented. That is, it is done for all pending updates on that page, so one gets the effect of batch processing of updates. Piggybacking updates shall be logged as in [Che98].

Piggybacking of changes can be complicated if there isn't enough room left on the page for those changes. This would happen if the new address for the moved record is larger than the old address. We call this the *overflow* problem. Detailed discussion of the overflow problem and our solution can be found in [Zou96].

4 System Issues

In this section, we will briefly discuss some system issues.

4.1 Concurrency and Recovery Concerns

With this approach, all operations can use record-level locking protocol. Details can be found in [Che98]. The main difference here is that we will not release locks until the transaction finishes. We have to be careful about phantoms with our method. In the worst case, one might have to bring in the parent page of an index leaf page to avoid phantoms.

Because all the lookup tables are in main memory, if there is a system crash, it is important that we could recover them correctly. Most of the techniques presented in [Che98] should apply here except *forward recovery*. At the end of the redo phase, we will undo those uncommitted transactions.

This means that we should delete the corresponding entries of those aborted transactions in the lookup tables.

4.2 Transaction Abort

When a transaction aborts, we should delete all the pending updates from the Q_i tables, and remove its entry in the transaction table T.

In order to support savepoint rollback, one might need to add a savepoint number as a field in the lookup table entry.

5 Future Work

In this paper, we outline a method to do index reference update just-in-time. The idea is to defer any updates that might need a disk I/O, and piggyback the updates with other user activities' I/O. The updates are done just-in-time so that index pages in the buffer are always consistent. By deferring index updates and piggybacking them (page-oriented) with other user activities' I/O, one can save a great amount of disk I/O. We would like to implement the method and measure its performance implications. We would also like to investigate other concurrency protocol to avoid phantoms with our method.

References

[Che98] Chendong Zou and Betty Salzberg. Safely and Efficiently Reference Updates during On-Line Reorganization. In *International Conference on Very Large Data Bases*, 1998.

[Zou96] Chendong Zou. *Dynamic Hierarchical Data Clustering and Efficient On-line Database Reorganization.* PhD thesis, College of Computer Science, Northeastern University, 1996.