Selective Deferred Index Maintenance & Concurrency Control in Integrated Information Systems

P. Dadam, V. Lum
IBM Scientific Center
Heidelberg, W. Germany

U. Praedel, G. Schlageter
University of Hagen
Hagen, W. Germany

Abstract

New applications of database management systems as in office automation and engineering require the system to process both textual and formatted data. To support text search appropriately, text indexes must be created and on-line text index maintenance be provided. Unfortunately, text index maintenance is generally a time-consuming task and does not fit well in an on-line environment, where short transaction processing times are usually required. In this paper we discuss how the time for those transactions, which cause text index updates, can be shortened by integrating a dedicated predicate-oriented concurrency control method and a selective deferred index update strategy. We also show some practical implementation techniques and some aspects of their performances.

1. Introduction

In the past, computer based information systems have been separated into two categories: database management systems (DBMS) and information retrieval systems (IRS). DBMS's have been designed to process formatted data, composed of a fixed number of atomic fields (attributes). Search conditions in queries are generally precise but relatively simple, allowing only the usual arithmetic comparison operators like less, equal, etc. w.r.t. field values. On the other hand, on-line concurrency control in DBMS's is generally very sophisticated. With the use of the transaction concept, where each transaction is treated as a unit, inconsistencies as a result of conflicts are avoided.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

In contrast, IRS's have been designed to process documents (books or articles) where unformatted (textual) data is the norm and a very long string (number of pages) of text is equivalent to an atomic field in a DBMS. Search conditions within this atomic IRS text field are less precise but can be rather complex. In a query one can search for certain words, or for certain substrings (fragments) of words, and even for a sequence of their appearance in the text. To efficiently support retrieval queries, one generally uses indexes. These are constructed from key terms or searchable terms (simply called *terms*) from the documents, and associated with each term is a pointer list pointing to the appropriate documents. As a long text string generates a large number of terms, index maintenance in IRS is a time-consuming task and is usually done in a batch mode during the night. Because the need to have the most current data is not critical, this approach is satisfactory in such an environment. Hence no on-line concurrency control is needed.

New applications like office automation, CAD/CAM, etc., having a mixture of both formatted and unformatted data, require both DBMS and IRS functions in one system. This means that the new system (called *Integrated Information System* (IIS)) must have indexes into the text fields as in IRS's and the concurrency control techniques as in DBMS's to provide efficient processing. As has been already recognized in /KSW79/, straightforward integration of both the IRS indexing technique and the DBMS concurrency control would produce long transaction time, which is detrimental to performance. The Advanced Information Mangement (AIM) project at the Heidelberg Scientific Center /Lum85/, in collaboration with Hagen University, attempts to find an alternate solution to reduce the transaction time in an IIS. Our proposed solution is reported below.

As overnight batch update to indexes is not acceptable in an IIS, because such a system does require current data, one must search for a solution that will reduce index maintenance time to obtain shorter transaction time. One strategy is to do index maintenance on only those terms that are most useful. Updating terms selectively permits the system to end a transaction immediately after its commit. However, as consistency must be maintained, and current data

is necessary, the deferred update on the remaining terms must be handled properly.

Deferring index updates selectively raises several questions, like

> How to recognize the terms that have to be updated to avoid consistency problems?
>
> How to represent the deferred update information?
>
> How to do bookkeeping for these updates?

The problem to decide which terms and their pointer lists have to be updated to avoid consistency problems (obsolete information) for a subsequent transaction is a concurrency control problem. In the following sections, we shall first discuss the concurrency control strategy, then the consistency aspects, the strategies for performing the deferred updates, and implementation methods.

2. Concurrency Control for Text Indexes

Concurrency control in DBMS's is usually done by applying variants of the Directed Acyclic Graph (DAG) locking philosophy /GLP75, GLP76, Gra78/. For special kinds of data, especially secondary data, dedicated concurrency control methods can sometimes be used to improve overall efficiency /BS77,KW84,Lau84,LY81,ML84/ (one can also see this from the aspect of synchronizing shared Abstract Data Types /SP84/). Earlier studies on how to perform concurrency control for text indexes /DPS82,DPS83/ have shown that performance can be improved by using a special method for synchronizing concurrent transactions on text attributes. The idea behind this approach is to use information derived from the search predicate of the query, or from the document to be processed, to decide more precisely than with standard DAG locking in a predicate oriented manner which transactions interfere among one another. The same approach is also useful for performing deferred index updates.

Let us first consider the standard DAG locking philosophy: A *read* transaction has to lock at least *one* access path to the object. An *update* (insert, modify, delete) transaction has to lock *all* access paths to the object.

Assume, that there is a query q which asks for all documents of type X, containing the terms *data*, *base*, *operating*, and *systems* and that there is another transaction u which inserts a document d of type X containing the terms *data*, *base*, *information*, and *systems*. The DAG locking would require that the inserter u acquires locks for the access paths corresponding to *data*, *base*, *information*, and *systems*. As the query q must lock at least one of its access paths too, there is a relatively high risk that query q and inserter u can not run in parallel. (They can only run in parallel if q locks the access path corresponding to "operating".) However, one can see that document d does not contain "*operating*", and is therefore not a match for q. In other words, d does not belong to the read set of q.

To avoid this kind of pseudo conflicts, we propose a solution which uses the terms contained in a document to be updated to form a *lock predicate*. The same is done with the terms occurring in a query. The method is designed to work for text indexes whose set of terms is stable, meaning it does not grow or shrink with insertions, modifications, or deletions. How it can be extended to work with 'unstable' term sets will be shown at the end of this paper (section 6).

Indexing methods of this kind include the Fragment String Method /KW81,KSW79,Sche78,Sche81/ or full word indexes basing on a controlled word set. With these approaches, the set of key terms, called *fragments* in the following, can be ordered and represented as a list, say FL:= $(frag_1,frag_2,...,frag_n)$ where $frag_i$ stands for the i-th fragment in this list. Having FL, any text document d can be (non-uniquely) represented by a bit vector F(d):= $(t_1,t_2,...,t_n)$ where $t_j = 1$, $1 \leq j \leq n$, if and only if fragment $frag_j$ is contained in d.

Analogously, queries can be represented by a bit vector as well. The bit vector (read fragment vector) F(q):= $(t_1,t_2,...,t_n)$ for a given query q contains $t_j = 1$ if and only if $frag_j$ is specified in the query. This means that q asks for all documents each of which must contain all the corresponding fragments having a '1' in this bit vector. (A zero in the read fragment vector can be interpreted as a "don't care" indicator for the corresponding fragment.)

Consider the example we have mentioned earlier. Assume FL:= (*data*, *base*, *compiler*, *construction*, *information*, *operating*, *systems*). The bit vector for our document d containing *data*, *base*, *information*, and *systems* becomes F(d):= (1100101) while that of our query q becomes F(q):= (1100011). A query q' asking only for terms not contained in FL, say, "computer" and "hardware", would get a bit vector consisting only of zeros. The same is true for an update transaction inserting a document d' containing only the terms "computer" and "hardware". (The latter is slightly different when using 'unstable' fragment sets as discussed in section 6.)

Denote AND to be the logical AND operation for intersecting equal length bit vectors. We say that bit vector F(a) *is contained in* bit vector F(b) if and only if F(a) AND F(b) = F(a). That is, for every bit of F(a) containing a 1, F(b) must have a 1 in its corresponding position. Clearly, a bit vector consisting only of zeros is contained in any bit vector.

Using this definition, one can define a precise conflict test between lock requests for read accesses (*read fragment locks*) and lock requests for accesses which update data (*write fragment locks*). (Note that read and write fragment locks are respectively the same as read and update fragment bit vectors as described above.) Let $FR(T_i)$ denote the set

of granted read fragment locks and $FW(T_i)$ the set of granted write fragment locks of transaction T_i. The rules for granting fragment locks can be stated as follows:

1. A read fragment lock l_r for transaction T_r is granted, if there exists no write fragment lock $l_w \in FW(T_w)$ for any transaction T_w, $T_r \neq T_w$, such that l_r is contained in l_w. We say T_r is *in conflict with* T_w (or vice versa) if l_r is contained in l_w.
2. A write fragment lock l_w is granted, if there exists no read fragment lock $l_r \in FR(T_r)$, $T_r \neq T_w$, such that l_r is contained in l_w.
3. Read locks never interfere among one another.
4. Write locks never interfere among one another.

The 4th rule may look surprising at first glance. One must remember, however, that we only talk about *index locks* here, and index locks alone are never sufficient to update a document because other access paths (e.g. relation scan, segment scan) exist as well. In addition to the index lock, an exclusive lock for the document itself has to be obtained before the operation may be performed. The index write locks are compatible for the same reason as the IX locks in System R, used for locking indexes /GLP75,GLP76/.

Queries containing OR-clauses can be represented either by using one fragment lock per OR-clause or by ANDing these vectors to get only one fragment lock. To insert or delete a document one write fragment lock is sufficient. To modify a document either one write fragment lock for the old value and one fragment write lock for the new value has to be obtained or the OR result of both vectors. (Note: The scope of a read fragment lock which is obtained by ANDing single read fragment locks is usually larger (in the best case equal) than the scope defined by the original set of single read fragment locks. The same holds analogously when combining a set of write fragment locks to one write fragment lock by applying the OR operation. In other words, more is usually locked in these cases than by using multiple fragment locks instead.)

Up to now we have discussed how conflicts at the level of *primary data* (documents) can be determined more precisely using the predicate oriented fragment locks as described above. As locks on primary data have usually to be kept until end of transaction to ensure consistency (serializability) /BSW79,EN81,Ull80/, more precise locks on primary data generally enhance the degree of parallelism between transactions. However, no conflict at primary data level does not necessarily mean no conflict at the *secondary data* (pointer list) level. Hence, the update of secondary data must be implemented such that parallel transactions never see inconsistent secondary data (invalid pointers, wrong length information, etc.). This can be achieved by temporarily locking the portion of secondary data for the duration of the update operation, by using 'careful replacement' strategies /GS76,Mul81,Ver78/, or by a combination of both.

3. Consistency Aspects of Deferred Index Update

For simplicity reasons we will assume the following: when a text index is used as access path for a given read transaction, the pointer lists corresponding to the 'on' fragment bits are intersected to obtain the final pointer list, and this list will then be used for accessing the documents themselves. In this case it is easy to see that pointer lists need only to be current with respect to conflicting transactions but can be obsolete otherwise (some updates on index are still pending).

As an example, consider again the query q and the inserter u of document d as introduced in the previous section. The final pointer list for q is obtained by intersecting the pointer lists corresponding to *data*, *base*, *operating*, and *systems*. As q is not in conflict with u, the pointer to document d will not occur in q's final pointer list, regardless whether the pointer lists corresponding to the terms of document d have been completely or partially updated, or even none of them has been updated yet.

Obviously, for a given transaction, updating pointer lists can only be deferred as long as no conflict with any subsequent or parallel transaction occurs. Consequently, transactions, which have logically committed but have not yet performed all their related index updates, must be tracked by the concurrency control mechanism as if they were still active. The problem of detecting conflicts due to pending index updates is, to a large extent, the same as the normal problem of detecting conflicts between concurrent transactions. Hence, lock representation and deferred update representation can be done in the same way to make conflict test simple.

Another question is, how to do *bookkeeping* for the completed and the pending index updates. Assume that bit vectors as described in section 2 are used to represent fragment locks in a dedicated lock table. Could one and the same bit vector be used to represent both the fragment lock and the update status? In other words, whenever a pending index update for a transaction has been performed, could the corresponding bit in the bit vector simply be switched from '1' to '0'.

Let $l_w \in FW(T_w)$ be the write lock of a committed transaction T_w locking the fragments corresponding to the vector (011001). l_w shall be in the lock table because the update of the index is still pending. The corresponding document itself, however, is already included in the database. That is, only the access paths are not up-to-date.

Assume now that transaction T_r with vector (011000) requests a read lock l_r. To compute the pointers to all the documents belonging to its read set, an intersection will be

done on the pointer lists for fragments $frag_2$ and $frag_3$. To ensure that it gets all the po of the committed documents, one must check if these two pointer lists corresponding to $frag_2$ and $frag_3$ are up-to-date. If the vector l_w for transaction T_w is (011001) like above, this conflict is detected and T_r would be blocked until the necessary updates have been performed.

Suppose now the pointer list corresponding to $frag_2$ had been updated and the lock vector would reflect this by changing the pending update vector for transaction T_w to (001001). A test for conflict will show that no conflict exists, even though the pointer list corresponding to $frag_3$ is still obsolete! This is clearly wrong as the new document updated by T_w should be in the read set of T_r.

As one can see from this example, the lock information for a transaction must remain unchanged even if only one pointer list remains to be updated. Bookkeeping on pending and completed updates on the terms has to be done separately. A write fragment lock can be removed from the fragment lock table only when all the terms for that transaction have been updated.

Another aspect of the deferred update approach is that all the lock information must be secure and *survive system crashes* to ensure consistent indexes. This can be achieved by logging this information. How this information can be represented compactly, is discussed in section 5.

4. Strategies for Performing Deferred Updates

In the previous section we have discussed how to recognize conflicts caused by pending index updates. In this section we want to discuss various alternatives to resolve such conflicts. For explanatory purposes we will assume in the following, that bookkeeping of pending updates is again done by using a bit vector representation. That is, a '1' signals that the corresponding pointer list has still to be updated (this point is reconsidered in section 5).

Let us assume that at time t_0 the fragment lock table for committed transactions with pending index updates and the bookkeeping information (update status) for a given attribute is as shown in Figure 1. Assume further, that the read lock depicted in Fig. 2 is requested at time t_1. As one can easily see from Figures 1 and 2, the lock request of transaction no. 6 conflicts with the write locks of transactions no. 1, 3, and 4 which are already in their deferred update phase. The straightforward solution to resolve such a conflict would be to perform all pending updates of the affected transactions. However, this solution would only shift the waiting for updates to some subsequent transaction and, as a consequence, would cause there unexpected as well as unacceptable long response times.

TNR	fragment write lock A B C D E F G H I J	update status A B C D E F G H I J
1	0 0 1 0 1 1 0 0 1 0	0 0 0 0 1 0 0 0 0 0
2	0 1 1 0 0 1 0 0 0 1	0 1 1 0 0 1 0 0 0 1
3	1 1 1 0 1 1 0 0 1 0	1 1 1 0 1 1 0 0 0 0
4	0 0 1 0 0 1 0 0 1 1	0 0 1 0 0 1 0 0 1 1
5	0 0 1 0 0 0 1 0 1 0	0 0 1 0 0 0 1 0 1 0

Fig. 1: Fragment Write Lock Table and Update Status Table at Time t_0

TNR	mode	status	lock request A B C D E F G H I J
6	read	active	0 0 1 0 0 1 0 0 1 0

Fig. 2: Read Fragment Lock Request at Time t_1

The key to a more effective solution is the following observation: When inserting or deleting a document, generally a relatively high number of fragment pointer lists has to be updated. As a consequence, in the vector describing the write fragment lock, many bits are usually 'on'. In contrast, queries or read requests tend to specify only a relatively small number of search terms or fragments. Hence, the vector describing the read fragment lock usually has only a small number of bits 'on'. Assume, for instance, that the fragment lock vector of an update transaction has 100 bits 'on', whereas the conflicting fragment lock vector of a query has 4 bits 'on'. In this case only the 4 bits which are common in both vectors cause the conflict. The other 96 bits are not of interest at this moment. This observation is used in the following approaches to perform index update processing not only in a *deferred* way but also *selectively*.

We will assume in the following that there are write locks FW_1, FW_2, ..., FW_n in the fragment lock table and corresponding update status vectors FU_1, FU_2, ..., FU_n, all belonging to committed transactions T_1, T_2, ..., T_n (only one lock entry assumed). Let FR_q be the read fragment lock request of a running transaction (not in conflict with any other active transaction) which has to be tested against the deferred update information of the committed transactions. Let IC be the index set of all conflicting write locks, i.e. $IC := \{i \mid FR_q \text{ in conflict to } FW_i, i = 1,2,...,n\}$. For all locks specified in IC by their index number one can now compute which updates have to be performed to resolve the existing conflict with FR_q. That is, for each entry i in IC the *conflict fragment vector* $CF_i := FR_q \text{ AND } FU_i$ can be computed. As an example, consider the situation of Figures 1 and 2. That is, $IC := \{1,3,4\}$. In this case, the conflict fragment vectors CF_i, $i = 1,3,4$, look like as depicted in Fig. 3.

In Fig. 3 the bit vector for transaction 1 contains only 0's. That is, there are no pending updates which cause a blocking of the read lock request. Only some pending updates of transaction 3 (fragments C and F) and transaction 4 (fragments C, F, and I) have to be performed. If, say at time t_2,

TNR	conflict vector A B C D E F G H I J
1	0 0 0 0 0 0 0 0 0 0
3	0 0 1 0 0 1 0 0 0 0
4	0 0 1 0 0 1 0 0 1 0

Fig. 3: Conflict Fragment Vector at Time t_1

these updates have been performed, the resulting lock table and update status table will appear as in Fig. 4.

TNR	fragment write lock A B C D E F G H I J	update status A B C D E F G H I J
1	0 0 1 0 1 1 0 0 1 0	0 0 0 0 1 0 0 0 0 0
2	0 1 1 0 0 1 0 0 0 1	0 1 1 0 0 1 0 0 0 1
3	1 1 1 0 1 1 0 0 1 0	1 1 0 0 1 0 0 0 0 0
4	0 0 1 0 0 1 0 0 1 1	0 0 0 0 0 0 0 0 0 1
5	0 0 1 0 0 0 1 0 1 0	0 0 1 0 0 0 1 0 1 0

Fig. 4: Fragment Write Lock Table and Update Status Table at Time t_2

Deferring index update selectively as described above leads usually (see /DLPS85/) to significant improvements in response times for transactions because update transactions commit earlier, and queries do not have to wait until conflicting transactions have performed *all* their index updates. If a pending update does not cause any conflicts, it may not be processed until the end of transaction processing (system shutdown). The above approach reduces response time for active transactions, but obviously, does not reduce the total amount of work with respect to index update; in fact, due to additional logging and bookkeeping the total overhead is slightly increased.

The key for saving also some amount of overall work is the following observation: To fetch a certain pointer list, say, 10 times to perform 10 updates is usually much more expensive than to fetch it only once and to perform the 10 updates in a batch-like manner. Consider once again the situation of Figures 1, 2, and 3. The pointer lists corresponding to fragments C and F have to be updated for transactions T_3 and T_4 to resolve the conflict. Instead of performing first all conflicting updates for T_3 and then all those for T_4 one could also work 'column oriented'. That is, first one fetches the pointer list of fragment C and performs the updates for both T_3 for T_4 before one releases the pointer list again, then the procedure is repeated for pointer list of fragment F etc..

A further improvement along this line is as follows: Instead of performing only the updates of conflicting transactions for a specific fragment, one could perform all pending updates for this fragment at once. Using this approach, the Update Status Table in Fig. 4 would show only 0's in columns C, F, and I. The response time for a given query might be slightly increased compared to the previous approach, however, I/O-overhead is reduced substantially. More investigations are required to see which solution is superior in which cases.

By using *idle times* of the system to perform pending updates, further improvements are possible. In this case some background task (let us call it *indexer*) would be activated to look for pending updates whenever system load is low. The indexer can also be used to control the amount of storage used for bookkeeping and/or to restrict the amount of work to be done after end of transaction processing. In this case, however, the indexer has to run concurrently to the other transactions even when the system is not idle, which may result in increased response time for the other transactions.

5. Implementation Considerations

Throughout the previous sections we have used the bit vector representation for fragment locks as well as for the update status information. This has been done mainly to simplify discussion. In this section we want to analyze some internal structures to see their effect on performance. In the following we will compare the Bit Vector (BV) representation with two list structures: a *Transaction Oriented Fragment List* (TOFL) and a *Fragment Oriented Transaction List* (FOTL). First we discuss briefly the lock representation problem. It is assumed that, besides the fragment lock information itself, some "global" information as Transaction Number (TNR) and Lock and/or Operation Mode (LMO) has to be maintained. Depending on the selected structure, one may also need the fragment number (FNR), the number of list entries (NoE), and a chain field. For simplicity, we will assume that each text attribute, as far as it is indexed, has its own fragment lock table. Thus, index-id and attribute-id need not be concerned in the following representations and discussions.

As alternatives to a bit vector representation (Figure 5), one can represent the fragment conjunctions either in a transaction oriented (row oriented) list structure (Fig. 6) or in a fragment oriented (column oriented) list structure (Fig. 7). In the latter case, for every fragment there exists a (perhaps empty) list of transaction and lock or operation entries. Obviously, the storage space needed for the bit vector representation is dependent only on the number of fragments used in a specific fragment index but not on the fragments actually specified in a read or write fragment lock (bit vector compression is beyond the scope of this paper and is not considered here). In contrast, the storage requirements of both list representations depend on the number of fragments specified in the fragment locks.

Fig. 5: Bit Vector Lock Representation

Fig. 6: Transaction Oriented Fragment List

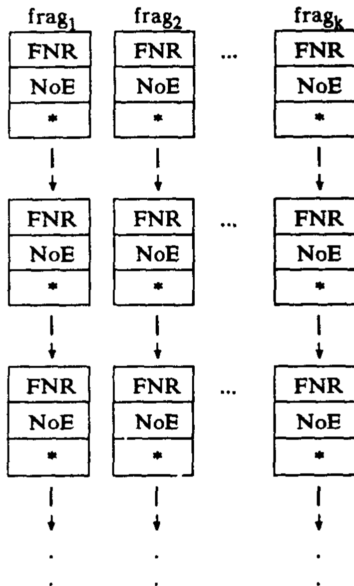

Fig. 7: Fragment Oriented Transaction List

The numbers in Table 1 show the amount of storage space needed for the different approaches in various environments. FRAGMENTS stands for the number of fragments used in the fragment index. READ__LOCKS and WRITE__LOCKS are the number of read lock entries and write lock entries in the lock table. READ__FRAGS and WRITE__FRAGS show how many fragments are specified on the average in a read or a write lock, respectively. USED__FRAGS shows how many *different* fragments are currently used by the read and write locks. That is, how much overlap occurs among the read locks or the write locks. The abbreviations BV, TOFL, and FOTL have been introduced at the beginning of this section. The extensions __R, __W, and __TOTAL stand for number of bytes for read fragment locks, write fragment locks, and the sum of both, respectively. Further, in calculating the figures in this table, we have assumed that transaction and fragment numbers can be represented with two bytes and chain pointers with four.

As one can see from table 1, the bit vector approach behaves quite well when the number of fragments and the number of read locks are relatively small (see columns 1 and 2). If, however, the number of locks increases (see columns 3 and 4), or a larger set of fragments is used (see col. 5), the BV

	1	2	3	4	5
FRAGMENTS	4000	4000	4000	4000	8000
READ__LOCKS	10	10	20	20	20
READ__FRAGS	4	4	4	4	4
WRITE__LOCKS	5	5	10	110	10
WRITE__FRAGS	100	200	100	100	100
USED__FRAGS	400	800	400	2000	400
BV__R	5030	5030	10060	10060	20060
BV__W	2515	2515	5030	55330	10030
BV__TOTAL	7545	7545	15090	65390	30090
TOFL__R	330	330	660	660	660
TOFL__W	3045	6045	6090	66990	6090
TOFL__TOTAL	3375	6375	6750	67650	6750
FOTL__R	520	528	800	672	800
FOTL__W	6460	13152	9960	92888	9960
FOTL__TOTAL	6980	13680	10760	93560	10760

Table 1: Storage Requirements for Lock Table

solution becomes relatively poor. The number of 110 write fragment locks in column 4 might seem somewhat high at first sight. However, having only 10 write locks of "active" transactions and in total 100 write locks due to pending updates would already reach this number. Looking at the storage requirements alone, the Fragment Oriented Transaction List approach also does not look very promising. It is always worse than the Transaction Oriented Fragment List approach and sometimes even worse than the bit vector approach.

The second essential issue is conflict testing. As explained earlier (see section 2), a read fragment lock request has to be tested (inclusion test) against all granted write fragment locks, regardless whether they belong to "active" transactions or pending updates. Assume that we have 100 write fragment lock entries, each having 100 fragments specified on the average, and a read fragment lock request having 4 fragments specified. In the bit vector case we would have to perform 100 bit vector AND operations. In the Transaction Oriented Fragment List case, we have to run through 100 lists and to touch 50 list elements on the average to find out whether there is a conflict. In the Fragment Oriented Transaction List case, however, we would simply intersect 4 lists to find out whether there is any transaction having all these 4 fragments specified. In our example, in the worst case, these lists would have 100 entries each. Furthermore, as the lengths of the fragment transaction lists are generally not the same, we can perform list intersection starting with the shortest ones first. This would reduce the operation time to find an intersection. Thus, as one can easily see this method of conflict testing is in general by far the fastest one.

There is also another aspect not considered so far. If the lock table becomes too large to be kept completely in main memory, one has to think about how to swap parts of it efficiently in and out of memory (to save I/O's). Using the BV or TOFL methods to store information, one can hardly

avoid piecewise fetching the whole lock table into main memory to perform the necessary comparisons. On the other hand, with the fragment oriented (FOTL) method, one has to fetch only the few (and relatively short) affected lists.

Our analysis leads to the following conclusion: In a general environment, where a large number of pending updates may occur, the write fragment locks should be organized as a Fragment Oriented Transaction List to enable fast conflict testing and swapping. The read locks should be organized in transaction oriented manner as in the Transaction Oriented Fragment List (alternatively, one can use a compressed bit vector representation). As these lists are usually very short (see table 1), one should be able to keep them in main memory. If the main memory is too small to capture all write fragment locks in Fragment Oriented Transaction List representation but would allow to store it either in bit vector or Transaction Oriented Fragment List representation, the bit vector representation would be preferred over the Transaction Oriented Fragment List due to its faster conflict test.

The bookkeeping information should be organized in the fragment oriented fashion, as already pointed out in section 4. Hence it should be organized in a Fragment Oriented Transaction List, too. In addition to the lock representation, the list elements have to carry now also the object's address to be inserted or deleted in the corresponding pointer list.

6. Extension to Unstable Fragment Sets

We have discussed so far fragment lock generation and conflict testing based on a stable fragment set. In the following we want to outline how this approach can be extended to work with 'unstable' fragment sets as well. An example for 'unstable' fragment sets is standard full word indexing where every new word contained in a document and not being a so-called "stop word"[1] will be inserted into the set of fragments. We will assume throughout this discussion that the fragment set used as reference basis for fragment lock computation is only allowed to *grow* during normal transaction processing. Shrinking shall only be allowed when there are no locks in the lock table. When allowing the fragment set to grow dynamically one has to show that no conflicts get "lost". For explanatory purposes we will assume that new fragments are always appended to the fragment list. To understand the problem, consider the following example.

Suppose that at time t_0 $FL_0 :=$ (*data,base,systems*) is the current fragment list. Assume, at time t_1, $t_0 < t_1$, a transaction T_r wants to read all documents containing *information* and *systems*. As T_r's read fragment lock l_r is computed based on FL_0 it looks like $l_r :=$ (001). Assume further, that "information" is simply ignored for bit vector computation. Consider now the case that at time t_2, $t_2 > t_1$, while T_r is still active, an update transaction T_w wants to insert a document d containing the fragments *information* and *systems*. This causes FL_0 to be extended first to FL_2 by adding "information". Obviously d belongs to T_r's read set. The question is, however, whether the conflict will be detected regardless whether T_w's write fragment lock request is computed based on FL_0 or on FL_2 assuming that l_r remains unchange

Let $l_{w,0} :=$ (001) and $l_{w,2} :=$ (0011) denote the write fragment lock computed based on FL_0 and on FL_2, respectively. As one can easily see, l_r is in conflict with both $l_{w,0}$ and $l_{w,2}$. But would it have caused any problem if T_r had asked only for "information" but not for "systems" as done above? - No, because in this case l_r would have been $l_r :=$ (000). Such an "empty" read fragment lock, however, conflicts with all write fragment locks (even with empty ones), except write locks belonging to the same transaction. Thus, in all these cases the conflict would be properly detected causing T_w to be blocked until T_r has released its locks.

The reason why the method can be applied for dynamically growing fragment sets as outlined above is based on the following fact: Let A, B, and C denote three arbitrary sets (including empty sets). If A is a subset of B, then A will always be a subset of $B \cup C$, too. As the conflict test described in this section is essentially a subset test, one can conclude that an existing lock conflict between a read fragment lock and a write fragment lock is not affected by "extending" the write fragment lock.

7. Summary and Conclusion

This paper concentrated on the issue of how to integrate text index maintenance into an on-line concurrent update environment such that the resulting transaction processing times remain acceptable. First, we discussed a dedicated concurrency control method which reduces the probability of lock conflicts by using a predicate-oriented locking scheme. Second, we analyzed how text index maintenance can be accomplished such that it can be done on-line without causing excessive transaction time. The key idea of this new approach is to selectively perform those index updates within a transaction which are relevant to ensure consistency, and to perform others after end of transaction processing using intermediate idle times of the system.

It has been shown that this approach of selective deferred update can be integrated into the concurrency control method described. As we have shown, the concept of selective deferred update reduces not only the transactions' response times but also the total amount of work by packaging several updates together such that the total number of necessary I/O's is decreased. We have also analyzed three practical implementation techniques to see the effects of different environment parameters on performance. In

[1] Stop words are those words that are excluded to be key terms (e.g. 'the', 'of', 'is')

addition, we have performed some simulations and the results clearly show the strength of the new approach compared to normal index maintenance techniques. Because of lack of space, these results are not included here but reported in /DLPS85/.

Acknowledgement

The authors would like to thank the IBM management, particularly Dr. A. Blaser, for their support, and their colleagues F. Andersen, K. Kuespert, and P. Pistor for their suggestions to improve this manuscript.

References

BS77 Bayer, R.; Schkolnick, M.: Concurrency of Operations on B-Trees. Acta Informatica, Vol. 9, 1977, pp. 1-21

BSW79 Bernstein, Ph.A.; Shipman, D.W.; Wong, W.S.: Formal Aspects of Serializability in Database Concurrency Control. IEEE Trans. on Software Eng., Vol. SE-5, No. 3, May 1979, pp. 203-216

DLPS85 Dadam, P.; Lum, V.; Praedel, U.; Schlageter, G.: Selective Deferred Index Maintenance & Concurrency Control in Integrated Information Systems: Concepts and Performance Evaluation. FernUniversitaet Hagen, Informatik Berichte Nr. 54, 1985

DPS82 Dadam, P.; Pistor, P.; Schek, H.-J.: Praedikat-Sperren mittels Textfragmenten, in: Informatik-Fachberichte 57 (Proc. GI-12. Jahrestagung, Kaiserslautern, October 1982), Springer-Verlag, pp. 648-668

DPS83 Dadam, P.; Pistor, P.; Schek, H.-J.: A Predicate Oriented Locking Approach for Integrated Information Systems, in: Information Processing 83 (Proc. IFIP'83, Paris, 1983), North-Holland Publ. Comp., 1983, pp. 763-768

EN81 Ekanadham, K.; Nigam, A.: On Serializability. IBM Research Report RC 9257, 1981

GLP75 Gray, J.N.; Lorie, R.A.; Putzolu, G.R.: Granularity of Locks in a Shared Data Base. Proc. VLDB 75, New York, September 1975, pp. 428-451

GLP76 Gray, J.N.; Lorie, R.A.; Putzolu, G.R.: Granularity of Locks and Degrees of Consistency in a Large Shared Data Base, in: Modelling in Data Base Management Systems, North-Holland Publ. Comp., 1976, pp. 365-394

Gra78 Gray, J.N.: Notes on Database Operating Systems, in: Lecture Notes in Computer Science, No. 60, Springer-Verlag, 1978, pp. 393-481

GS76 Giordano, N.J.; Schwartz, M.S.: Data Base Recovery at CMIC. Proc. ACM-SIGMOD Conf., Washington, D.C., June 1976, pp. 33-42

KSW79 Kropp, D.; Schek, H.-J.; Walch, G.: Text Field Indexing, in: Database Technology (Proc. ACM German Chapter, Seminar on Data Base Technology, 1979), Teubner-Verlag, 1979, pp. 101-115

KW81 Kropp, D.; Walch, G.: A Graph Structured Text Field Index Based on Word Fragments. Information Processing and Management, Vol. 17, No. 6, 1981, pp. 363-376

KW84 Kwong, Y.; Wood, D.: A New Method for Concurrency in B-Trees. IEEE Trans. on Software Eng., Vol. SE-8, No. 3, May 1982, pp. 211-222

Lau84 Lausen, G.: Integrated Concurrency Control in Shared B-Trees. Computing, Vol. 33, 1984, pp. 13-26

Lum85 Lum, V.; Dadam, P.; Erbe, R.; Guenauer, J.; Pistor, P.; Walch, G.; Werner, H.; Woodfill, J.: Design of an Integrated DBMS to Support Advanced Applications. Invited paper, Proc. International Conference on Foundations of Data Organization, Kyoto University, Japan, May 1985; also in: Proc. Fachtagung GI Datenbanksysteme in Buero,Technik und Wissenschaft March 1985, Karlsruhe, Germany, pp. 362-381

LY81 Lehman, P.L.; Yao, S.B.: Efficient Locking for Concurrent Operations on B-Trees. ACM Trans. on Database Systems, Vol. 6, No. 4, Dec. 1981, pp. 650-670

ML84 Manber, U.; Ladner, R.E.: Concurrency Control in a Dynamic Search Structure. ACM Trans. on Database Systems, Vol. 9, No. 3, Sept. 1984, pp. 439-455

Mul81 Mullin, J.K.: Change Area B-Trees: A Technique to Aid Error Recovery. The Computer Journal, Vol. 24, No. 4, 1981, pp. 367-373

Sche78 Schek, H.-J.: The Reference String Indexing Method, in Lecture Notes in Computer Science 65 (Proc. Informations Systems Methodology, Venice, Italy, 1978), Springer-Verlag, 1978, pp. 432-459

Sche81 Schek, H.-J.: Methods for the Administration of Textual Data in DB Systems, in: Information Retrieval Research (Proc. of Research and Development in IR, Cambridge, May 1980), Butterworths, London, 1981, pp. 218-235

SP84 Schwarz, P.M.; Spector, A.Z.: Synchronizing Shared Abstract Types. ACM Trans. on Computer Systems, Vol. 2, No. 3, August 1984, pp. 223-250

Ull80 Ullman, J.D.: Principles of Database Systems. Pitman Publ. Ltd., London, 1980

Ver78 Verhofstad, J.S.M.: Recovery Techniques for Database Systems. ACM Computing Surveys, Vol. 10, No. 2, June 1978, pp. 168-195