To appear in 19th ACM SIGIR International Conference on Research and Development in Information Retrieval (SIGIR 96), Zurich, Switzerland, August 1996

Efficient Transaction Support for Dynamic Information Retrieval Systems

Mohan Kamath and *Krithi Ramamritham*
{*kamath,krithi*} *@cs.umass.edu*
Department of Computer Science
University of Massachusetts
Amherst MA 01003-4610 USA

Abstract

To properly handle concurrent accesses to documents by updates and queries in information retrieval (IR) systems, efforts are on to integrate IR features with database management system (DBMS) features. However, initial research has revealed that DBMS features optimized for traditional databases, display degraded performance while handling text databases. Since efficiency is critical in IR systems, infrastructural extensions are necessary for several DBMS features, transaction support being one of them. This paper focuses on developing efficient transaction support for IR systems where updates and queries arrive *dynamically*, by exploiting the data characteristics of the indexes as well as of the queries and updates that access the indexes. Results of performance tests on a prototype system demonstrate the superior performance of our algorithms.

1 Introduction

Recently there has been an explosion in the size of text databases and the number of on-line sites containing text databases. To handle this growth and to cater to more sophisticated queries, information retrieval (IR) systems have been evolving rapidly. Providing transaction semantics for concurrent accesses by updates and queries to documents has been a long recognized goal. Hence recently there have been some research efforts to integrate information retrieval (IR) features with database management system (DBMS) features [Fuh93, BCCM94, DDS+95]. However, initial findings have revealed that DBMS features optimized for large traditional databases show performance degradation while handling text databases [DDS+95]. Efficiency is a critical factor that governs the success of large IR systems and hence successful integration of IR and DBMS features requires careful analysis and redesign of some of the DBMS features. One of them is transaction support for IR and it has received little attention both from the IR and DBMS research communities.

This paper proposes efficient transaction techniques for IR systems, focusing on index management. Specifically, to achieve high performance, our concurrency control and recovery algorithms exploit the data characteristics of the indexes as well as of the queries and updates that access the indexes. In this paper, we primarily focus on *dynamic* IR systems (DIRS) where queries and updates arriving dynamically are processed. Thus the articles of interest to a user are known only when a query arrives. This is unlike systems like SIFT [YGM95a] and Tapestry [TGNO92] which are geared to continuously respond to *statically* specified queries or *filters*. In these systems, a new document that arrives is routed to a user if it passes the filtering criterion specified by that user. Neverthless the techniques we describe for dynamic queries are applicable to these static queries as well. An example of a system that handles dynamic queries is an on-line search system for web pages and news articles (like Infoseek search). Here documents arriving over the network with rates sometimes as high as 100 MB/hour [Cro95] are continuously analyzed and added to the system, while several thousand queries are processed every hour.

In many IR systems, for example, news retrieval systems, the most recently added items are often the most relevant ones. Also for any general query, the best document that satisfies the query could be among the latest documents added to the system. Thus, all efforts should be made to consider the most latest documents for processing a query. A similar situation exists in distributed IR systems, comprised of thousands of independent text databases. In the *text-database discovery* problem [YGM95b], before answering a query, the most appropriate site which contains documents that best match a query is to be determined. This is achieved by accessing a local database that is continuously updated using meta-information arriving from other sites. If concurrent updates (latest updates in particular) to this local database are not handled properly, the query could be directed to a less appropriate site when in fact there are better sites. Our concurrency control and recovery algorithms are specifically geared to handle such scenarios where changing information must be accessed in dynamic decision making scenarios.

Since text databases store huge collections of documents, they can be efficiently retrieved only with the use of indexes, which are usually referred to as *inverted-lists*. An inverted-list exists for each keyword (term) that exists in one or more documents in the text database. It contains the document IDs of documents that contain the keyword. Optionally, other information like the number of documents that contain the keyword and the location information about the keyword (within the documents) will also be stored. Since the document themselves are immaterial to the processing of query/update transactions, in this paper, we concentrate mainly on index management.

In DIRS, when new documents are added, updates to the indexes can be modeled as transactions. If updates to indexes are done as database transactions, traditional concurrency control schemes would acquire long term locks on the data items for updates. However, an analysis of the data characteristics of the indexes and transactions that access the indexes reveals that there are no tight inter-relationships between the index entries as is the case between data items in traditional database systems. Consider the addition of two independent documents (we will return to dependent documents later). This will result in the addition of the IDs of the two documents to the index entries of the keywords that appear in the documents. Thus the updates are actually appends and hence there is no dependency between the different data items being updated. Thus, there is no need to perform inplace updates as far as document additions are concerned. This has tremendous implications on the design of concurrency control and recovery schemes. We have designed a concurrency control scheme that uses latches (short term locks) in combination with a novel technique called *operation reordering* to achieve the correctness of updates while also satisfying the recency requirement. Because update transactions are of long duration, use of latches with operation reordering significantly improves the efficiency of the system since the locking overheads incurred for reading indexes to process queries and for updating indexes after document analysis are reduced. We have also incorporated special schemes to efficiently handle deletions and modifications to documents. To handle recovery, simple but effective schemes have been designed since there is no need to undo changes[1] that have already been made by updates. Because update transactions are of long duration, transaction splitting techniques have been used to reduce restart time while recovering from system failures. From the point of view of information service providers, the high performance resulting from the use of our schemes will be very attractive in order to remain competitive.

The rest of the paper is organized as follows. Section 2 discusses the data and transaction characteristics in IR systems. Section 3 presents the architecture of system. Section 4 describes our concurrency control and locking schemes. Section 5 discusses our logging/recovery schemes. In section 6 we describe our prototype system, performance tests and present the results of our performance tests. Section 7 concludes the paper.

2 Data and Transaction Characteristics

In this section we present our data model and then analyze the performance and correctness requirements of transactions in IR environments.

2.1 Data Model

Documents added to the system can be structured or unstructured. Irrespective of the structure of the documents, in this paper we will assume that they are manipulated as a complete entity. As they are added to the system, they are assigned monotonically increasing IDs that serve as logical identifiers. Since inverted-lists have to be accessed quickly, normally B-trees (or some other index structure) are used to maintain a keyword-index (primary index). There are several ways in which the inverted-lists (secondary index) themselves can be linked to a keyword in a B-tree node [GR93], especially for non-unique keyword additions. The choice reflects the tradeoff in the number and type of accesses. The type of structure used for maintaining the keyword-index or the inverted-list is not relevant to us since the techniques we are proposing in this paper are *independent* of the structure used.

In the case of static queries (filters), user-profiles are stored in the database. Users can modify them at any point in time, even while they are being continuously evaluated as inverted-lists are updated. The user's new profile should go into effect as early as possible. Both static profiles and dynamic queries consist of a set of keywords and the operators connecting (logical - AND/OR, proximity - NEXT, etc.) them. The query goes through an optional query expansion phase by referring to the context dictionary, *i.e.*, a user specified query is transformed to have more semantic context (by changing/adding new keywords) before it reads the inverted-lists [LK94, BGAS94]. Any changes made to the context dictionary should also be reflected in the system at the earliest.

2.2 Query and Update Processing

When a document is added to the database, the indexes must be updated to reflect the addition of the new document to the database. Suppose a new document has fifty keywords, say, "summer", "olympics", "Atlanta" and so on. When this document is added to the database, the fifty keywords must be added to the keyword-index, if they are not already there. Then the ID of the document must be added to the inverted-lists associated with the keywords. Thus, these updates are typically in the form of appends, *i.e.*, the document ID is appended to the inverted-list associated with each of the keywords.

Although it is possible to access and update these indexes on the fly as the documents are analyzed, it is not done so for two reasons — the cost of (i) accessing the indexes and (ii) acquiring locks several times is high. Hence the new documents are analyzed first in a batch and the keywords extracted, and all the inverted-lists are updated at the end in an incremental fashion as shown in figure 1. The incremental strategy also obviates the need to rebuild indexes[BCC94, TGMS94].

Given the keywords relevant to a query, the set of inverted-lists corresponding to these keywords is identified. After reading the inverted-lists, the document IDs and other required information are extracted from the inverted-lists. Further query processing is done based on the type of the query

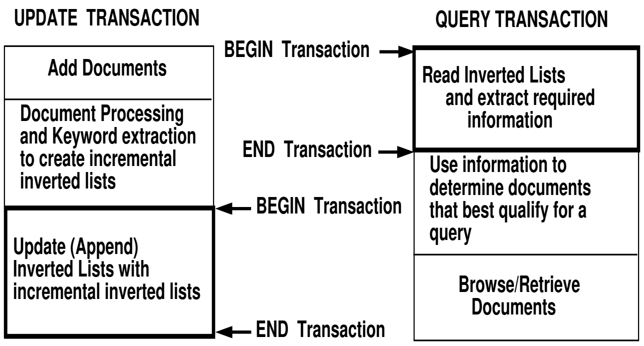

Figure 1: *Transactions in DIRS environments*

[1]deletions are handled in a different manner.

and other operators specified in the query. Once the document IDs are determined, query processing is complete and the user is presented with the basic information about the documents like the abstract or the first few lines. Several optimizations can be performed to truncate the search process [Bro95].

The highlighted phases in figure 1 show the scope of the transactions during query and update processing. Accesses to inverted-lists in this phase are to be made only after acquiring locks.

2.3 Performance and Correctness Requirements

To amortize the cost of accessing and updating the indexes, a group of documents is analyzed as a batch. Thus updates are performed on a large number of inverted-lists, increasing the duration of such updates. When updates are of long duration, queries will also have longer response times and the probability of transactions deadlocks also increase [GR93]. We now focus on performance and correctness considerations, so that we can exploit them to design high performance concurrency control and recovery schemes tailored for DIRSs.

There are a number of distinctions between the requirements of DIRS environments and that of traditional applications like banking:

- In banking, an account's value depends on other subaccounts or related accounts and hence there exist dependencies between the individual data items. In DIRS applications, the index entries for two documents are not tightly related. Of course, the indexes must correctly reflect the contents of a document and this leads to consistency requirements relating a document and its corresponding index entries.
- Updates often add entries to the indexes[2], *i.e.*, a particular index entry is not written based on the content/value of the same or some other index entry and thus the contents of one data item are independent of other data items. Thus, in DIRSs, as opposed to arbitrary operations which update the data in-place, appends/additions are the norm. These operations commute.

These distinctions lead to differences in the way updates and queries are structured as transactions in DIRS and in the *atomicity, consistency,* and *durability* properties associated with them:

- The atomicity requirement is that *all* the changes to the document database and the indexes must exist at the end of the update transaction.
- The consistency requirement is that at the end of the transaction *both* the documents and the indexes must be mutually consistent.
- The durability requirement is that at the end of an update transaction all the documents and changes to their index entries must be durable in the database. If there are system failures, it is unacceptable to start the transaction all over again from the beginning. Hence we perform forward recovery by continuing the transaction from the point where the failure occurred.

Because update transactions can be of long duration we should be concerned about the *isolation* properties of these updates especially since queries are typically of short duration. Isolation requirements can be usually specified based on the dependencies between the read (R) and write (W) operations. The three dependencies we need to consider are W-W, W-R and R-W [BHG87].

- *W-W dependencies:* Since updates are in the form of appends and appends to a set can be considered to be idempotent[3]. Appends also commute and hence these dependencies need not be tracked. If the index also stores a count of documents that contain the keyword, it will have to be updated based on the previous value. However additions of documents entail incrementing this counter and increment operations are commutative. For these reasons, W-W dependency can be ignored.
- *W-R and R-W dependencies:* These correspond to a query and an update executing concurrently. Since (1) an update transaction updates the index entries associated with the documents one keyword at a time and at most once, (2) a query reads the index entries associated with a keyword only once, and (3) there are no integrity constraints between the keywords, not tracking these dependencies simply means that a query and a concurrent update are unaware of each other; the index entries read by the query will reflect all the updates done until then to the indexes and so the query results will be correct in the sense that it will not return any extraneous documents. Thus, from a correctness point of view, W-R and R-W dependencies need not be tracked.

Unfortunately, if W-R and R-W dependencies are ignored, the query results will *not* reflect updates to index-entries that are yet to be done by updating transactions which are still in progress. The net effect is that a query may not consider the most recent articles. Suppose new documents on "relational database products" are being added to the database and the update transaction has only updated the index entries for keyword "relational" and is yet to update the keyword "database". If a query needs articles that contain "relational" and "database" in it, it would read the updated metadata for "relational" and nonupdated metadata for "database" and miss out on all the new documents that would have really qualified. Thus tracking updates, and more specifically, the needs of queries vis. a vis. the concurrent updates, is very important since they affect the result of the query. If a query does not consider the most recent additions to the database, its results will be outdated. Hence to improve the recency metric, query processing must be cognizant of concurrent updates.

To summarize, the isolation requirements of queries and updates justify use of latches, which is a major advantage – since updates and queries need to latch only one data item at a time, index accesses involve only short waits and no deadlocks occur. This indicates that long terms locks used in several traditional applications can unnecessarily degrade performance in IR systems. Because latches do not help track W-R or R-W dependencies on specific metadata items, we need some other means to satisfy the recency criterion. Towards this end we have introduced the notion of operation reordering as discussed in section 4. Because dependencies need not be tracked, the update can be programmed as a

[2] We will consider the issue of deletes and modifications later in section 4.

[3] The append operation can be implemented in an idempotent manner to avoid duplicates.

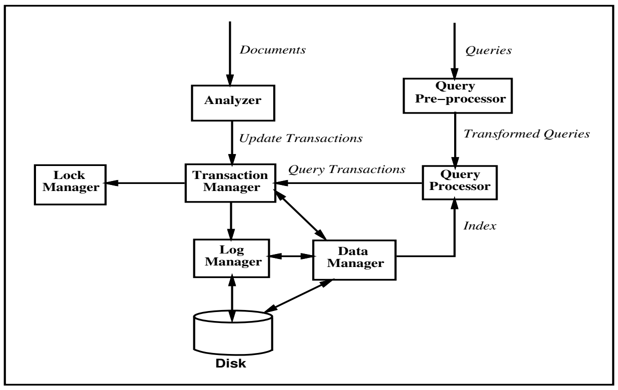

Figure 2: *Software Architecture of DIRS Server*

mini-batch[4] [GR93]. Since there no dependencies between index entries and there are no operations that can lead to logical errors, logical failures are rare. This allows us to realize efficient forward recovery as discussed in section 5.

3 System Architecture

The software architecture for the DIRS server is shown in figure 2. It contains the essential IR components and DBMS components necessary for our study.

New documents are processed by the *analyzer* which forms the incremental inverted-lists. The analyzer then submits an update transaction to the *transaction manager* whose data items are the documents and the incremental inverted-lists. Queries that arrive from various clients go through the *query pre-processor* where queries are expanded using the help of a context/semantic dictionary. The *query processor* then submits query transactions to the transaction manager whose data items are the inverted lists for the keywords in the transformed query. The rest of the interactions take place in the DBMS components.

The transaction manager keeps track of the queries and updates concurrently in progress. The *lock manager* implements the concurrency control scheme. The transaction manager gets the required locks on each of the inverted-lists by contacting the lock manager. Once the locks have been acquired the transaction manager informs the data manager about the data items. The *data manager* performs the required read and write operations on the documents and indexes. Index entries requested by the query processor is obtained from the data manager. The *log manager* oversees the operations of the data manager and logs the occurrence of significant events during updates. The log manager is responsible for ensuring the durability of the updates, *i.e.*, both the newly added documents and their index entries are made persistent on the disk.

4 Concurrency Control

Having analyzed the correctness requirements, in this section we will first present our scheme to handle queries concurrently with document additions. Then we describe extensions to this scheme to handle document deletions/modifications and handle other situations.

[4]Short transactions that perform part of the update. The atomicity granularity is still that of the long update transaction.

4.1 Scheme

Our concurrency control scheme uses *latches*[5], *i.e.*, short duration locks[6], for reads and writes. A latch is the cheapest form of lock that can be used to ensure consistency of data in case of physical rearrangement of data on pages. If we are interested only in correctness and "recency" of documents is not a concern then just using latches is sufficient. However, since recency is of interest to us, we introduce a new locking technique called *latching with operation-reordering.* This technique exploits knowledge about data and transactions. The algorithms are shown in figures 3 and 4.

Recall that after the documents have been analyzed, the inverted-lists to be read and written by the queries and updates respectively are known and an inverted-list is accessed only once. When a transaction arrives, the transaction manager informs the lock manager about all the inverted-lists that will be accessed by the transaction. The lock manager also keeps track of all the active update transactions. When a query arrives the lock manager will perform a conflict check with all active update transactions to see if there are any common inverted-lists (keywords). If there are inverted-lists common to a query and a concurrently executing update transaction, then the ID of that update transaction is added to the query's *conflict list* and "hints" are passed to the update transaction to update these common inverted-lists at the earliest. The transaction manager in turn reorders the operations of the update transaction and submits updates on the common inverted-lists first. Once a common inverted-list is updated and the write latch is released, the query can latch the item and read it. Our mechanism ensures that the query does not latch the item before the update write latches it and that the updates write to these items first. This way the lock waiting time for queries is drastically reduced and we can ensure that the query sees the most recent documents without paying a high price.

The locking mechanisms and data structures are similar to the ones described in [GR93]. The main data structure is a hash table and each hash chain contains records identifying inverted-lists and other relevant information like lock status. The lock header for each inverted-list contains a list of transactions to whom locks have been granted and a waiting queue for transactions that need a lock on that inverted-list. When transactions to whom locks have been granted finish their work, the next item from the waiting queue is granted a lock. Accesses to the lock table are serialized. We have made some enhancements to the lock hash table to maintain additional information required for operation-reordering. For each inverted-list, there is a *processed list* apart from the usual *granted list* and *waiting queue.* The processed list contains transaction IDs of active updates that have already done an update to the inverted-list after obtaining a write lock. A lock for a inverted-list is available if there is no entry corresponding to that inverted-list in the lock table. The lock requests corresponding to inverted-lists of four keywords are shown in figure 5. We now explain how locking is done for queries and updates and in the process show how the entries in the table are utilized.

When an update requests a lock, it is handled as follows: A write lock is granted if it is available else it is put on the wait queue. After a lock is granted and the write has been performed, the transaction ID of the update is added to the

[5]In the rest of our discussion, we will use the term latch and lock inter-changeably.

[6]Locks acquired prior to an access are released immediately after the access.

```
Add inverted-list IDs (keyword) to access-set of Transaction;
If (Transaction == Update)
   Add Transaction-ID to active-update-transactions list;
else /* Transaction type is Query*/
   Check for conflicts with each transaction in active-update-transactions list;
   For each transaction that conflicts, add that Transaction-ID to the query's conflict-list;
   Pass "hints" to the transaction manager to reorder the operations of the conflicting transaction
        such that the conflicting operations are performed at the earliest;
```

Figure 3: *Establishing conflict set at lock manager & passing hints to transaction manager*

```
1. Read Lock/Unlock Request from Query:
     If (Operation == Lock)
        Perform safety check - ensure transactions in conflict-list have completed or written the inverted-list;
        (the above is done in conjunction with the processed-list of the inverted-list);
        If safe and lock not granted to write
           issue read lock;
        else
           place request on wait queue for inverted-list;
     else /* Operation is Unlock */
        If nobody is in the wait queue
           release read lock & delete entry from lock table;
        else /* somebody is in wait queue */
           grant lock to inverted-list on wait queue;


2. Write Lock/Unlock Request from Update:
     If (Operation == Lock)
        If lock not granted
           issue read lock;
        else /* lock has been granted */
           place request on wait queue for inverted-list;
     else /* Operation is Unlock */
        Enter Transaction-ID in processed-list of inverted-list
        If nobody is in the wait queue
           release read lock & deleted entry from lock table;
        else /* somebody is in wait queue */
           grant lock(s) to inverted-list(s) on wait queue; /* multiple grants only for reads*/
           (locks granted to queries only after safety check)
```

Figure 4: *Algorithms for reordering write operations*

processed list. When all the operations of the update complete, the entries corresponding to this update are removed from the respective processed lists. For this purpose all lock table entries for an update transaction are chained. When a query requests a lock, the lock is granted only if it is safe (see figure 4) and available else the request is inserted into the wait queue. Then when the lock is available it is granted and after the operation is performed the entry is removed from the lock table. Note that only transaction IDs of updates are inserted into the processed list.

Note that the above scheme reorders the operations such that the query sees the inverted-lists after the updates from concurrent update transactions have taken place, thus ensuring that the query results reflect all documents that have been or would have been added to the database by all active update transactions present in the system when the query begins. The other case, where a update begins when a query is in progress is also worthy of consideration. However, experimental results have shown that the payoffs are negligible, especially since queries are short compared to the update transactions. We discuss this further in section 6.

4.2 Discussion

Deletions and modifications are not as frequent as additions in IR environments. Deletion of a document involves removing the document ID from the various inverted-lists and removing the document itself from the database. Modification can be handled in two ways. One option is to add the modified version as a new document and delete the old version. The other option is to determine the deleted and added keywords by determining the difference between the documents[7] and removing the document ID from the inverted-lists corresponding to deleted keywords and adding the document ID to inverted-lists corresponding to added keywords. Irrespective of how the deletion or modification is achieved, they have to be done via an update transaction. Hence our concurrency control scheme does not need modifications to ensure that the deletion/modifications are immediately reflected in query results without much overheads.

Keyword	**Locks**		
	Processed	**Granted**	**Waiting**
..	..	..	..
olympics	U_1, U_2	Q_3	U_5
..	..	..	..
library	U_3	Q_4, Q_6	U_6
..	..	..	..
digital	U_3	U_4	Q_5
..	..	..	..
summer	U_2, U_3	Q_4	U_6
..	..	..	..

Figure 5: *Lock Table (U_i — update i, Q_i — query i)*

It should be noted however that deletions and modifications are costly operations since (i) documents are to be reanalyzed and (ii) locks are to be acquired. Hence we propose the following approach to reduce the amortized locking costs. In the case of deletion, the document IDs for the

[7]Using a Unix *diff* like program.

deleted articles is maintained in a *purged-documents* list with a bound on the maximum size of the list. Thus, when the list of qualifying documents is determined for any query, a check is performed against this list and the IDs of the purged documents removed from the qualifying list. For modifications, the first option mentioned above is used in conjunction with this approach. When the bound is exceeded or the load is light, the purged-documents are analyzed as a *batch* to determine all the keywords and the document IDs are removed from the corresponding inverted lists. This approach gives very good performance, especially since deletions or modifications are rare in DIRS.

Now we discuss the addition of *related* documents. Although relationships exist for example in hypertext documents, they do not carry over to their respective index entries. This is because the documents are first added to the database and only then are the index entries updated. When the hypertext documents are added, all the documents exist in the database and consistency must be ensured only in terms of having a link in a document point to the correct document. Hence our concurrency control schemes do not need any modifications to handle the updates of related documents. A document which is pointed to by a link is not permitted to be deleted unless the IR administrator permits this.

We now discuss how our latching with operation reordering scheme applies to scenarios other than the inverted-list updates.

1. The use of our concurrency control scheme for updating the context dictionary is simple. The updates to the dictionary are done before queries can access the keyword contexts from the dictionary.
2. The application of our concurrency control scheme to the text-database discovery problem mentioned earlier is also straightforward. The latest keyword meta-information updates arriving from other sites are seen by the query posed against the local database before routing a request to a site. Thus the queries will be routed to the best site based on the latest available information without incurring much overheads.
3. In the context of static continuous queries, when a user submits a new profile or makes a profile modification, the profile inverted-lists [YGM95a] corresponding to these profiles are to be updated. The document analyzer reads the profile inverted-lists only after the profile inverted-lists have been updated to reflect the profile modification performed by the user. The documents received by a user from the IR system will thus immediately reflect the profile changes.

To summarize, we have presented a concurrency control scheme that guarantees that the queries see the most recent documents without paying a huge cost. Another advantage of our scheme is that it is independent of the index structure used for the keyword-index and the inverted-lists. Hence it can be used in the integration of IR with any type of DBMS — RDBMS or an OODBMS and with any index structure like hashing or B-tree and its variants. Since our schemes are independent of the index structure used, applications can store and interpret the indexes [LS88, DDS+95]. We did not discuss the details of concurrency control scheme for the different type of index structure since they have been thoroughly investigated [Moh90, ML92, MN92, SC92, LS92].

5 Logging and Recovery

In this section we present logging techniques for making document changes durable on the disk and recovery techniques to ensure consistency in the event of failures. These are implemented by the log manager (figure 2). Recall from section 2 that the document analysis phase by itself is outside the scope of the update transaction and the inverted lists for each keyword is updated all at once inside the update transaction.

5.1 Scheme

First we discuss operations outside the scope of the update transaction, which are not logged. Since document analysis is a CPU intensive operation, it is unacceptable to start analyzing a whole batch from scratch in the case of failures. Hence *persistent savepoints* are taken periodically to flush the partial results (incremental inverted-lists) onto the disk. Note that these partial results are not stored in the database itself but in a temporary area on the disk. Since there are no logs, persistent savepoints are much simpler than checkpointing — the buffers containing the incremental inverted-lists are flushed to disk and a savepoint record that contains the document ID of the last document analyzed is saved persistently on the disk. If there is a system failure, only a little work is lost since the analysis can continue from the next document after the last savepoint, *i.e.*, the document next to the one which has been analyzed and whose incremental inverted-lists have been flushed to the temporary area on the disk.

Next we discuss operations inside the scope of the update transaction. Since data items (inverted-lists) in the database are not written based on the value of other data-items in the database, logical failures do not occur. Also since inverted-lists are just appended to, there is no need to keep track of what the previous value of a data item was before writing to it. These significantly simplify logging in terms of what is logged. Specifically there is no need to log the before values of the data items. Since we already store the incremental inverted-lists in a temporary area on the disk, there is also no need to store the new value that is written. Hence a log record just contains the *(transaction-ID, keyword)* pair. If there is a system failure, for each ongoing update transaction it is known as to which of the keyword inverted-lists have been updated and which are yet to be and thus forward recovery is possible. Since the value of the data-items are not logged, log records are much smaller and hence logging an update is much quicker.

An elegant solution to handle failure situations for long transactions in database systems[8] is the use of a mini-batch and this idea is applicable for update transactions in IR system. For our concurrency control scheme, a mini-batch would be the updates to inverted-lists corresponding to a small set of keywords. The updates within the mini-batches are performed by fetching the disk resident database pages of the required inverted-lists (bringing the last page of each list is sufficient) into the buffer and appending the new document IDs from the incremental inverted-list. After the set of updates within a mini-batch have been performed, the transaction is committed. But this does not mean that the buffer changes have been made persistent on the disk. Hence we force the log records of the committed mini-batch to the disk at commit time. A checkpoint is to be taken periodically. Several factors have to be considered before determin-

[8] For *e.g.*, interest computation for all accounts in a bank

ing the frequency of checkpointing [TR84]. When a checkpoint is taken, all system inverted-list database buffers are flushed to the disk and a checkpoint record that contains the LSN (log sequence number) of the last log record processed is also flushed to the disk. For system inverted-lists whose database pages have been updated and flushed to the disk, incremental inverted-lists stored in the temporary area on disk removed. Note that there is no need to maintain and update a page LSN as in traditional transaction processing systems.

Recovery from system failures that can occur between checkpoints is to be handled carefully since we do not store the after values in the log records for the updates. It is possible that a few mini-batches commit between a checkpoint and the failure. For all such committed mini-batches we have flushed the log records to the disk. The log record from which the redo pass is to begin is identified based on the LSN of the last checkpoint record persistent on the disk. Hence for all the keywords in such log records, we have to fetch the incremental inverted-lists saved in the temporary area on disk into the buffer. Then these changes are to be applied by bringing in the disk resident database pages corresponding to the system inverted-lists on the disk. When such updates have been performed, the system inverted-list buffers are flushed to the disk and their copy is removed from the temporary area. A new checkpoint record with the LSN of the last log record processed is then flushed to the disk. Recall that since none of the update transactions have to be aborted, an undo pass is not required. Recovery is now complete and the system can start the next mini-batch update and process queries.

5.2 Discussion

To get additional performance, more optimizations are possible. Document analysis and the system inverted-list updates can be pipelined, *i.e.*, as the savepoints occur in the document analysis phase, the system inverted-lists can be updated with the incremental inverted-lists created during that savepoint. If system failures are not frequent, then the two stages of the pipeline can be collapsed into one, *i.e.*, document analysis and the system inverted-lists update can be done together. If this is the case then the incremental inverted-lists need not be stored in a temporary area, instead directly used to update the system inverted list. This also implies that the log records must now store the after value of an update, *i.e.*, the incremental inverted-list. Note that if the updates are of long duration and documents are arriving at a high rate, more than 2 update transactions can be executing concurrently in the system. Our logging and concurrency control scheme are flexible enough to handle such situations.

To summarize, we presented logging and recovery techniques for IR systems that are much simpler than techniques needed for traditional databases and yet guarantee the correctness required by IR systems. Specifically, the log records and the logging process have been made as simple as possible based on the characteristics of IR systems. These simpler techniques reduce the overheads and hence the system can perform very efficiently.

6 Performance Tests

In this section we briefly discuss the details of the prototype system, nature of the tests and the performance results. We have only concentrated on the performance of the concurrency control schemes. The benefits of the logging/recovery scheme is clear since we perform much less work compared to traditional recovery schemes and hence performance tests are not needed to verify them.

6.1 Prototype System

We have built a prototype by extending our bibliography search system following the architecture presented in section 3. The bibliography search system has been in use on our world wide web site[9] for a few months. The system has been built on a SunSparc 20 multiprocessor workstation with 96 MB of main memory, 2 GB SCSI disk and running Solaris 2.3 version of Unix. Most of the components shown in figure 2 have been implemented as independent Unix processes communicating using IPC messages. Our prototype system does not contain a query pre-processor or a log manager. We use a variant of the B-tree to manage the keyword-index.

6.2 Nature of tests

To create an environment where documents arrive over the network, documents were pre-analyzed and the incremental inverted-lists are continuously added to the system. In order to evaluate the system in real world situations, several queries have been taken from the access log of the web server. They range from 2 to 4 keywords and the keywords are well distributed among the spectrum of words present in the documents. The rest of the important characteristics of document collections for updates are presented in the table below:

Characteristic	Value
Avg. no. of unique keywords in a batch	1,694
Total no. of word occurrences in a batch	17,326
Avg. number of documents in a batch	1,000

The above data set reflects short documents arriving in large quantities over the network and hence we decided to process 1000 documents in a batch, although this is a tradeoff based on several parameters. The total size of a batch is about 0.45 MB, which is close to that used in [DDS+95]. Notice that updates are long since on an average 1694 inverted-lists are to be updated. To avoid deadlocks arising from long duration updates in the DB-locking scheme, we rearranged the keywords whose inverted-lists are to be updated in an alphabetical order.

We use two performance metrics to compare the different schemes we are testing. The *response time* for update and query transactions is the wall clock time elapsed in seconds. The response time for an update is the time taken to update the inverted-lists while for the query it is the time taken to read the inverted-lists and determine the documents that satisfy the query. *Recency* is measured in terms of the percentage of query transactions that failed to read the documents that are being added by the transactions that are concurrently performing updates.

We compare our scheme denoted by 'Latch with Operation-Reordering' with two other schemes. *Latching* refers to the scheme that uses latches and satisfies the correctness requirement that the query returns only the correct documents, *i.e.*, those in the database. *DB-locking* provides isolation level 2, *i.e.*, cursor stability. This was implemented to study the effect of using an off-the-shelf transaction processing approach, one most suitable for our needs. This is the popular locking mode used by most traditional DBMS

[9] (at URL http://www-ccs.cs.umass.edu/db/bib-search.html).

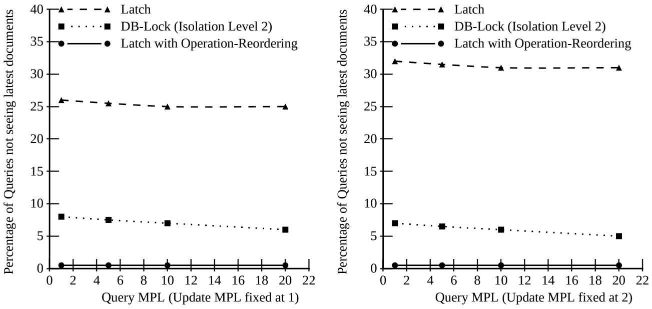

Figure 6: *Percentage of queries missing the latest documents*

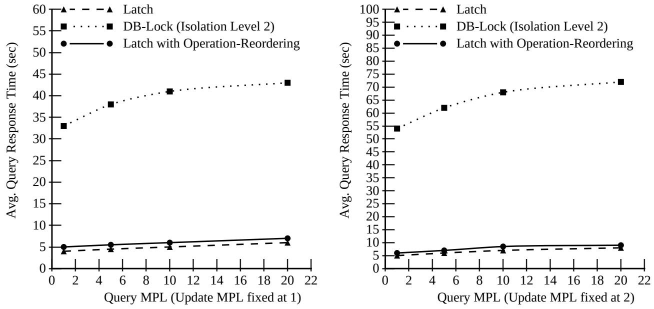

Figure 7: *Average response time of queries*

applications. Here queries use short term locks and updates use long term locks.

The tests have been performed by varying the multiprogramming level (MPL) of query and update transactions. To maintain the desired query and update MPL, a query and an update spawner *fork* the required number of threads respectively. The threads in turn invoke the update and query transactions. In our tests, we fixed the update MPL to a particular value and then varied the query MPL over a range.

6.3 Results

To study the recency properties of the queries executed by the three schemes, we first examined the percentage of queries that failed to fetch the latest documents being added by the concurrent update transactions. This is determined by comparing the (1) timestamps of the queries and updates and (2) document IDs returned by the queries with those added to the inverted-lists by the updates. Specifically, if we know query Q executed concurrently with update U and Q did not return the document IDs of qualifying documents added by U, then query Q failed to fetch the latest documents.

As shown in figure 6, it can be seen that queries using our scheme fetch the latest documents (99.5 % of the time). Upon close examination of the system we found that when a query requests a set of latches, the hints from the lock manager to the transaction manager and the subsequent write operation on the common keywords from the updates are done almost immediately in quick succession. Thus our idea of operation reordering has a huge pay off and contributes to good recency. The 0.5 % we miss in terms of latest documents can be attributed to update transactions that start while query transactions are in progress. Similar to the compensation based on-line query processing technique [SC91], we could force the query to check if there are new updates before it completes and if so re-execute the query. The fact that only 0.5 % of the queries miss the latest documents suggests that such a situation is very infrequent because queries are short and updates are long. For this reason, the rare query that does benefit from this extra check may not warrant the extra costs that are incurred by the additional check at the end of *each* query. However if an increase in query response time is tolerable, this technique can be adopted to ensure that all the queries see the latest data.

The latching scheme performs badly since it misses a fairly large number of updates. This behavior worsens further when the update MPL is increased to 2 since it now misses appends from both updates.

The DB-locking scheme performs moderately. Even here, while on an average most queries fetch the recent documents there are certain queries that fail to get the latest articles. These are queries that acquired read locks on the items before the update could get write locks on them. In our implementation of DB-locking, write locks are acquired in a growing phase and then released as a batch at the end. If the locks were acquired and released the other way, *i.e.*, all locks are acquired at the beginning and released in a slow shrinking phase, the queries would have seen the latest articles but the average response time will also increase substantially.

The excellent behavior of our technique incurs a small overhead — seen in the form of a slight increase in the response time. Response time is the time that elapses between query submission time and the time when the query finishes reading the required inverted-lists and determining the qualifying documents. The results are shown in figure 7 (note the two graphs have a different scale along the Y axis).

The response time of DB-locking is very high compared to that of latching and our scheme. The difference between our scheme and latching is in the order of a second and can be attributed to (1) the cost of checking common keywords between queries and concurrent updates and (2) the time spent by read operations from queries waiting for updates to happen on the inverted-lists of those common keywords. For an update MPL of 1, we also measured the average time for an update transaction as 378 seconds for the latching and DB-locking scheme. Due to the additional checks performed by our scheme, the updates take about 4% more time. As the update MPL increases to 2, the update transaction time for the DB-locking scheme shoots up rapidly due to write lock contention while for the other two schemes it increases only marginally.

In summary we see that our latching with operation reordering considers the latest articles without paying a high price. If there are no concurrent updates, since there are no checks to be performed, our scheme performs the same as the latching scheme.

7 Conclusions

Transaction processing techniques optimized for traditional database system are unsuitable for efficiently handling queries and updates in text databases. Hence new schemes are needed to successfully integrate IR features with DBMS features. In this paper we focused on developing efficient concurrency control and recovery schemes that exploit the characteristics of data and transactions in IR systems. Our focus was on systems where queries and updates arrive dynamically. A significant advantage of our techniques is that they are independent of the scheme used for indexing.

Our specific contributions are as follows:

- We analyzed the characteristics of transactions in dynamic IR environments by studying queries and updates on indexes and determined the correctness properties of interest. Our observation is that traditional transactions and transaction processing techniques are unnecessarily restrictive for these environments and can degrade performance.

- To satisfy the requirements of dynamic queries, we proposed a concurrency control scheme which — (a) exploits data and transaction knowledge, (b) uses short term locks for both reading and writing, and (c) allows dynamic reordering of accesses to achieve high performance.
- We proposed simple logging and recovery techniques to reduce restart time while recovering from system failures.
- To quantify the benefits of our schemes, we implemented the schemes on a prototype system and compared the performance of competing approaches. Our schemes showed substantial improvement in performance.

While it may be possible to incorporate the ideas presented in this paper into a text search engine that uses an off-the-shelf database product as a repository for the inverted-lists and documents, the desired performance and correctness may not result. Hence the ideas we describe here are to be utilized by database system developers to cater to the needs of emerging database applications like information retrieval.

By concentrating on correctness and system related performance issues, our work complements the work done thus far in information retrieval for building correct, scalable, high performance text database systems and digital libraries. Each of our schemes is based on simple ideas and is founded on sound observations about the nature of the data, of the queries and updates. Nevertheless the practical impact of these ideas has been shown to be significant.

Acknowledgement

This work has been supported by NSF grant IRI-9314376 and a grant from Sun Microsystems Labs.

References

[BCC94] E.W. Brown, J.P. Callan, and W.B. Croft. Fast incremental indexing for full-text information retrieval. In *Proc. of Intl. Conference on Very Large Databases (VLDB)*, Santiago, Chile, 1994.

[BCCM94] E.W. Brown, J.P. Callan, W.B. Croft, and J.E.B Moss. Supporting Full-Text Information Retrieval with a Persistent Object Store. In *Proc. of Intl. Conference on Extending Database Technology*, Cambridge, UK, 1994.

[BGAS94] C. Buckley, G.Salton, J. Allan, and A. Singhal. Automatic Query Expansion Using SMART: TREC3. In *Third Text REtrieval Conference (TREC-3)*, pages 69–80, Gaithersburg, Maryland, 1994.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[Bro95] E.W. Brown. Fast Evaluation of Structured Queries for Information Retrieval. In *Proc. of SIGIR Intl. Conf. on Research and Development in Information Retrieval*, 1995.

[Cro95] W. B. Croft. What do People Want from Information Retrieval? In *D-Lib Magazine. Available at http://www.dlib.org/dlib/november95/11croft.html*, 1995.

[DDS+95] S. DeFazio, A. Daoud, L. Smith, J. Srinivasan, B. Croft, and J. Callan. Integrating IR and RDBMS Using Cooperative Indexing. In *Proc. of SIGIR Intl. Conf. on Research and Development in Information Retrieval*, pages 84–92, 1995.

[Fuh93] N. Fuhr. A probabilistic relational model for the integration of IR and databases. In *Proc. of SIGIR Intl. Conf. on Research and Development in Information Retrieval*, pages 309–317, 1993.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[LK94] X. A. Lu and R. B. Keefer. Query Expansion/Reduction and its Impact on Retrieval Effectiveness. In *Third Text REtrieval Conference (TREC-3)*, pages 231–240, Gaithersburg, Maryland, 1994.

[LS88] C. Lynch and M. Stonebraker. Extended user-defined indexing with application to textual databases. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 306, Los Angeles, CA, August 1988.

[LS92] D. Lomet and B. Salzberg. Access method concurrency with recovery. In *Proc. ACM SIGMOD Conf.*, page 351, San Diego, CA, June 1992.

[ML92] C. Mohan and F. Levine. ARIES/IM: An efficient and high-concurrency index management method using write-ahead logging. In *Proc. ACM SIGMOD Conf.*, page 371, San Diego, CA, June 1992.

[MN92] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *ACM SIGMOD Conf. on the Management of Data, San Diego*, June 1992.

[Moh90] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on BTree indexes. In *Proceedings of the 16th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Brisbane*, August 1990.

[SC91] V. Srinivasan and M. J. Carey. Performance of B-tree concurrency control algorithms. In *ACM SIGMOD Conf. on the Management of Data, Boulder*, May 1991.

[SC92] V. Srinivasan and M. J. Carey. Compensation-based on-line query processing. In *Proc. ACM SIGMOD Conf.*, page 331, San Diego, CA, June 1992.

[TGMS94] A. Thomasic, H. Garcia-Molina, and K. Shoens. Incremental Updates of Inverted Lists for Text Document Retrieval. In *Proc. of SIGMOD Intl. Conference on Management of Data*, May 1994.

[TGNO92] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proc ACM SIGMOD Conf.*, pages 321–330, San Diego, California, 1992.

[TR84] A. Tantawi and M. Ruschitzka. Performance analysis of checkpointing strategies. *ACM Transactions on Computer Systems*, 2(2):123–144, May 1984.

[YGM95a] T. W. Yan and H. Garcia-Molina. SIFT - A Tool for Wide-Area Information Dissemination. In *In Proc. of 1995 USENIX Technical Conference*, pages 177–186, 1995.

[YGM95b] T.W. Yan and H. Garcia-Molina. Information Finding in a Digital Library: the Stanford Perspective. *SIGMOD Record*, 24(3), September 1995.