# Efficient Transaction Management & Query Processing in Massive Digital Databases †

*Mohan Kamath* and *Krithi Ramamritham*
Computer Science Technical Report 95-93
Department of Computer Science
University of Massachusetts
Amherst MA 01003
{*kamath,krithi*}*@cs.umass.edu*

**Abstract**

We address several important issues that arise in the development of Massive Digital Database Systems (MDDSs) in which data is being added continuously and on which users pose queries on the fly. *News-on-demand* and document retrieval systems are examples of systems that have these characteristics. Given the size of data, metadata such as index structures become even more important in these systems — data is accessed only after processing the metadata, both of which will reside on tertiary storage. The focus of this paper is on *query and transaction processing* in such systems, with emphasis on *metadata management.*

The performance in these systems can be measured in terms of the *response time* for the queries and the *recency* or age of the items retrieved. Both need to be minimized. The key to satisfying the performance requirements is to exploit the characteristics of the metadata as well as of the queries and updates that access the metadata. After analyzing the functionality and correctness properties of updates, we develop an efficient scheme for executing queries concurrently with updates such that the queries have short response times and are guaranteed to return the most recent articles. Secondly, we address logging and recovery issues and propose techniques for efficiently migrating metadata updates from disk to tape. Thirdly, considering the tape access needs of queries, we develop new tape scheduling techniques for multiple queries such that the response time of queries is reduced.

Results of the performance tests on a prototype system show the superior performance of the developed algorithms and reveal that to build high performance MDDSs it is imperative that we adopt approaches that exploit the data and transaction characteristics.

**Keywords:** Concurrency Control, Transaction Management, Query Processing, Digital Libraries, Multimedia Databases, Information Retrieval, Hierarchical Storage, Optimization, Performance

† Supported by the National Science Foundation under grant IRI - 9314376 and a grant from Sun Microsystems Lab

# Contents

# 1 Introduction

Massive Digital Database Systems (MDDSs) store peta-bytes of data with tera-bytes being added every day [Wor94]. Examples of such applications include digital libraries for news, office article management systems and earth observation satellite systems. To store, retrieve and manage such massive amounts of digital data, there is a need to develop efficient MDDSs. MDDSs use hierarchical storage systems [Wor94] consisting of primary, secondary and tertiary storage devices to handle the huge amounts of data while achieving a better price-performance ratio. In such a hierarchical storage system, the tertiary device holds all the data and metadata while the secondary and primary act as a two level cache. While disk farms[1] provide secondary storage, tape libraries provide tertiary storage.

In MDDSs, data is typically retrieved based on contents. The most popular form of retrieval is based on keywords that occur in articles. To retrieve data efficiently, metadata, in the form of indexes, is required. Typically the size of the metadata is of the same order of magnitude as the data and hence metadata will also reside on tertiary storage. The disk will only contain metadata that is *currently* needed by queries or additions made by updates. Given this, metadata such as index structures become even more important in these systems — data is accessed only after processing the metadata.

The focus of this paper is on high-performance *query and transaction processing* techniques for MDDSs, with emphasis on *metadata management*. The performance of a MDDS can be measured in terms of the *response time* for the queries and the *recency* or age of the items retrieved. Both need to be minimized. The special characteristics of data and metadata, the high latency of tapes and the desired performance criteria demand the development of novel approaches to query and transaction processing in MDDSs. The key to satisfying the performance requirements is to exploit the characteristics of the metadata as well as of the queries and updates to the metadata.

We are interested in MDDSs in which data is being added continuously and wherein users enter the MDDS dynamically and pose queries on-the-fly. *News-on-demand*, *i.e.*, digital library for news, and on-line information retrieval systems have these characteristics. Here new articles are constantly added to the database and the articles retrieved by the dynamic queries must ideally include the most recent additions. 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 user is informed about a new article if it passes the filtering criterion. Even though we are interested in on-the-fly queries, our techniques do find applicability in such situations also.

Our contributions are:

- We analyze the functionality & correctness properties of updates and then develop an efficient scheme for executing queries concurrently with updates. The queries have short response times and are guaranteed to return the most recent articles.

  Our concurrency control technique uses just latches, *i.e.*, short term locks, but still satisfies atomicity, consistency, and durability of updates. Also, a query is executed so as to return the most recent articles satisfying the query predicate – including those articles being added by the concurrent update transaction(s).

- We develop logging and recovery techniques for efficiently migrating metadata updates from disk to tape. They are designed to minimize the disruption experienced by tape accesses entailed by the queries and caused by the need to migrate the updates to tape.

[1] disk farms will be hereafter referred to as disks.

Changes to metadata are migrated to tapes in a lazy manner. However, these are stable logged and maintained in such a way that query results are based on the most recent state of the database.

- Since the time to mount a tape and seek data within a tape is in the order of few tens of seconds, access to tapes must be even better optimized than to disks [SSU90, CHL93]. We develop novel approaches to scheduling the tape access requests of dynamically arriving concurrent queries such that the average response time of queries is minimized.

  Our approaches minimize the number of tape mounts by reading/writing data from/to a mounted tape opportunistically. Which tape to mount next is based on the lengths of the queries, measured in terms of the number of data items accessed, as well as on the specific tapes on which they reside. The tape scheduling techniques have the nice property that response times are minimized while fairness to queries of different lengths is maintained.

Results of the performance tests on a prototype system show the superior performance of the developed algorithms and reveal that to build high performance MDDSs it is imperative that we adopt approaches that exploit the data and transaction characteristics. For information service providers, the resulting high performance will be very attractive in order to remain competitive.

It is important to point out that while the traditional transaction model and associated transaction processing approaches may be able to solve some of the problems in MDDSs, they cater to more restrictive database situations and hence are likely to be wanting when satisfying the recency and response time performance criteria. This is because, data components in MDDSs do not have the tight interrelationships that are the norm in typical database applications. For instance, in a news database, each news item can be considered independent of another. The metadata about two items are also not tightly related. Of course, the metadata about a news item must correctly reflect the contents of that item and this leads to consistency requirements relating a data item and its metadata. These characteristics have implications for how we design the queries and updates as transactions, in particular, how we achieve the atomicity, consistency, and durability of updates and the correctness of query results. Hence issues addressed in this paper to achieve good performance become important given the current trend to used DBMSs for managing metadata [VCC95].

The rest of the paper is organized as follows. Section 2 discusses the data and transaction characteristics. Section 3 presents the system architecture and some of the details of our prototype system. Section 4 describes our concurrency control technique and locking scheme. Our schemes for logging/recovery and migrating data from disk to tape are discussed in section 5. Section 6 presents our optimization technique for scheduling multiple queries. Related work is discussed in section 7. Scope for future work is enumerated in section 8 and section 9 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 MDDS environments.

## 2.1 Data Model

MDDSs store data of various types including text, image, audio and video. Metadata in MDDSs primarily consists of indexes. Textual articles are typically indexed based on keywords. An article can then be retrieved from a huge collection by specifying a set of keywords and the predicates joining them

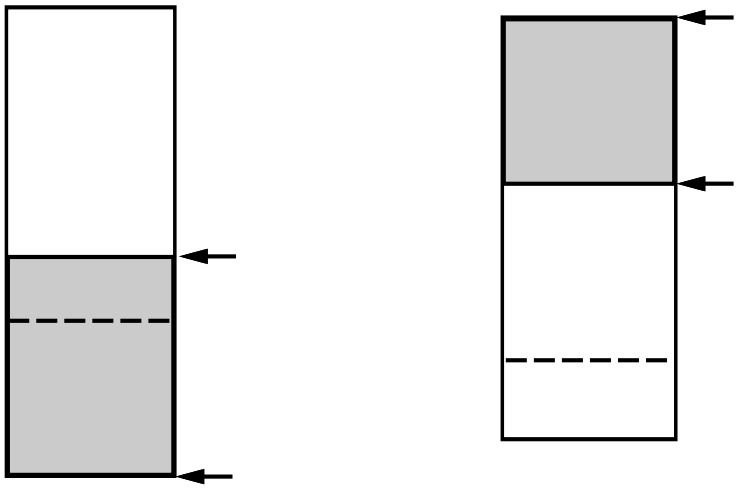

Figure 1: *Transactions in MDDS environments*

[Fal85, Cro89]. An image can be retrieved by specifying various properties like color, texture, shapes and sizes [ACF+94, Chi94, CLP94]. The query processor in both these cases refers to several indexes and determines the articles/images that are appropriate for a query. Thus, metadata is accessed during initial processing to determine the articles/images that qualify for a query while the article/image as such is retrieved only based on the matches found during initial processing and subsequent browsing/filtering done by the user. The techniques we describe in this paper are generic even though we use a text retrieval system as the motivating example.

The metadata consists of a set of keywords and associated with each keyword is an *article ID-set*[2], the set of IDs[3] of articles which contain that keyword. Optionally, other information like the number of articles that contain the keyword (which is essentially the size of the article ID-set) and the location information about the keyword (within the articles) will also be stored. Making sure that the indices are updated and maintained in such a way that queries return the most recently added documents requires careful concurrency control of accesses to the metadata. Modifications and deletions to data are very rare in such environments [SB94].

## 2.2 Query and Update Processing

When an article is added to the database, the metadata must be modified to reflect the addition of this article to the database. For example, consider a new article. It is first analyzed to determine the keywords in it. Suppose this new article has two keywords, say, "baseball" and "playoffs". When this article is added to the database, the two keywords must be added to the metadata, if they are not already there, and the ID of the article must be added to the article ID-sets associated with them. Thus, these updates are typically in the form of appends: the article is appended to the set of documents in the database and the article ID is appended to the article ID-sets associated with each of the keywords.

[2] which is actually an inverted list

[3] ID refers to the logical identifier

Although it is possible to access and update these indexes on the fly as the articles are analyzed, it is not done so for two reasons. The cost of accessing the indexes are high, and the locking costs will be high. Hence the new articles are analyzed first in a batch and the metadata extracted, and all the article ID-sets are updated at the end in an incremental fashion [BCC94, TGMS94] as shown in figure 1.

A query consists of a set of keywords signifying the articles of interest. To process such queries, first, the list of metadata to be read is identified. The article IDs corresponding to these keywords are then extracted from this metadata. To minimize the time a query holds read locks on the metadata, the required metadata is copied into the query area and the locks released before processing can start as shown in figure 1. Further query processing is done based on the type of query, for example, to process an AND query given two keywords, the common article IDs – in the metadata pertaining to the two keywords – are determined. Once the article IDs are determined, the query is essentially processed and the user is presented with the basic information about the articles like the abstract or the first few lines. Several optimizations can be performed to truncate the search process, but we do not go further into this issue.

Since the articles themselves are immaterial to the query/update transactions, in this paper, we concentrate mainly on metadata management.

### 2.3 Performance and Correctness Requirements

To amortize the cost of accessing and updating the metadata, a group of articles are added in a single update transaction. Thus updates are performed on a large number of metadata objects, increasing the duration of such updates. Identification of the performance and correctness considerations helps us design concurrency control schemes that are tailored for MDDSs in the context of such updates.

There are a number of distinctions between the requirements in MDDS 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 MDDS applications, each article can be considered independent of another. The metadata about two items are also not tightly related. Of course, the metadata about a news item must correctly reflect the contents of that item and this leads to consistency requirements relating a data item and its metadata.

- Updates only add to the metadata, *i.e.* a particular metadata is not written based on the content/value of the same or some other metadata and thus the contents of one metadata is independent of other metadata. Thus, in MDDSs, as opposed to arbitrary operations which update the data in-place, appends/additions are the norm. These changes commute.

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

- Atomicity requirement is that *all* the changes to the article database and the metadatabase must exist at the end of the update transaction.

- Consistency requirement is that at the end of the transaction *both* the articles and the metadata must be mutually consistent.

- The durability requirement is that at the end of an update transaction all the articles and changes to their metadata must be durable in the database. If there are system failures, the transaction must continue from the point where the failure occurred since in unacceptable to start the transaction all over again from the beginning. Since updates are in the form of appends and appends to a set can be considered to be idempotent, we can perform forward recovery upon a failure. To handle failures that may occur during a long update, the update can be programmed as a mini-batch [GR93]. Since there no dependencies between data items and no operations that can lead to logical errors, logical failures are rare.

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.

- *W-W dependencies:* Since appends from multiple updates commute, these need not be tracked. If the metadata also stores a count of articles that contain the keyword, it will have to be updated based on the previous value. However additions of articles 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 metadata associated with the data one keyword at a time and at most once, (2) a query reads the metadata 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 metadata read by the query will reflect all the updates done until then to the metadata and so the query results will be correct in the sense that it will not return any extraneous articles. Thus, from a correctness point of view, W-R and R-W dependencies need not be tracked.

Unfortunately, the query results will not reflect updates that are yet to be done by updating transactions which are still in progress. The net effect is that a query may not return the most recent articles. Suppose a new article on "relational database products" is being added to the database and the update transaction has only updated the metadata 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 articles 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 completeness of the result of the query. This is called *recall* in information retrieval. The *number* of articles retrieved by a query is an *indication* of the recall metric. Another consideration is the *recency* or *currency* of the results. If a query does not return the most recent additions to the database, its results will be outdated. The *age* of the articles retrieved is an indication of the recency metric. To improve performance of a query with respect to both metrics, query processing must be cognizant of concurrent updates.

Another situation where concurrent updates must be tracked occurs during *text-database discovery* [YGM95b]. Here, each site maintains metadata related to articles at other sites such that a query can be directed to a site that has the relevant articles. If concurrent updates are not tracked, then the query could be directed to an inappropriate site when there are better sites in reality.

Not having to track dependencies allows us to use just latches, or short term locks, which is a major advantage – since updates and queries need to latch only one metadata item at a time, metadata

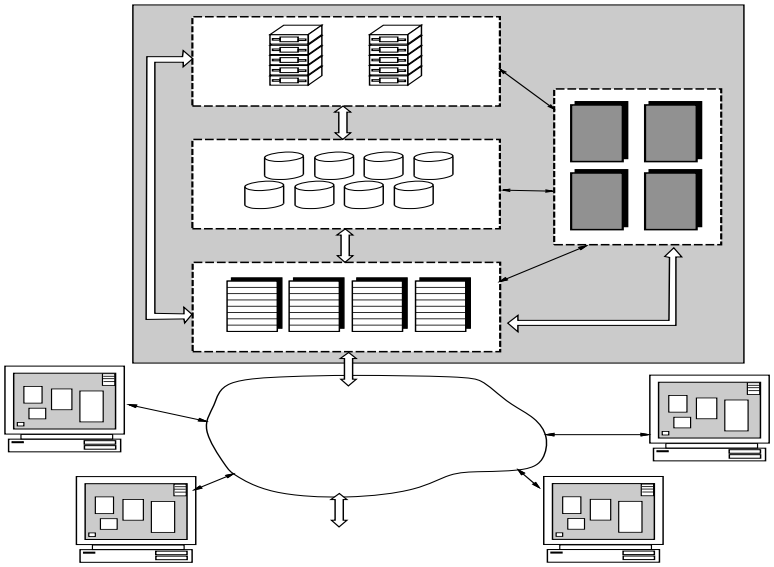

Figure 2: *Storage Hierarchy of MDDS Server*

accesses involve only short waits and no deadlocks occur. 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.

To summarize, the isolation requirements of queries and updates justifies use of latches. But some mechanism is needed to satisfy the recency requirement. This is the subject of Section 4. Logging and recovery techniques needed to satisfy atomicity and durability are discussed in Section 5. Finally, scheduling tape accesses to minimize query response times is the subject of Section 6.

In the above discussion we assumed that there are no deletions or revisions to articles. This is the case most of the time [SB94]. However if for some reason deletions/revisions occur then additional steps must be taken. For revising a article, the metadata corresponding to the article to be deleted is to be removed from all the relevant metadata and new entries are to be added to the metadata to correspond to the new version. This is a fairly complicated process since the old version of the article may have to be analyzed to determine the keywords. Since the space consumed by articles that are occasionally deleted is comparatively negligible, it might be better to use the following approach. The article IDs for the deleted articles can be maintained in a *purged-articles* list on the disk and when the qualifying list of articles is retrieved for any query, a check can be performed against this list and the IDs of the purged articles deleted from the qualifying list.

## 3 System Architecture

The storage hierarchy for a MDDS server, shown in figure 2, consists of a primary (RAM-volatile memory), secondary (disk farm) and tertiary storage (tape library)[4]. The tape library holds several tapes and any required tape can be loaded into the reader. The time for mounting a tape is of the order of a few seconds, sometimes as high as 40 seconds. Once a tape has been loaded a seek is to be

[4]Instead a read-write optical jukebox can also be used and we return to this issue later in section 6

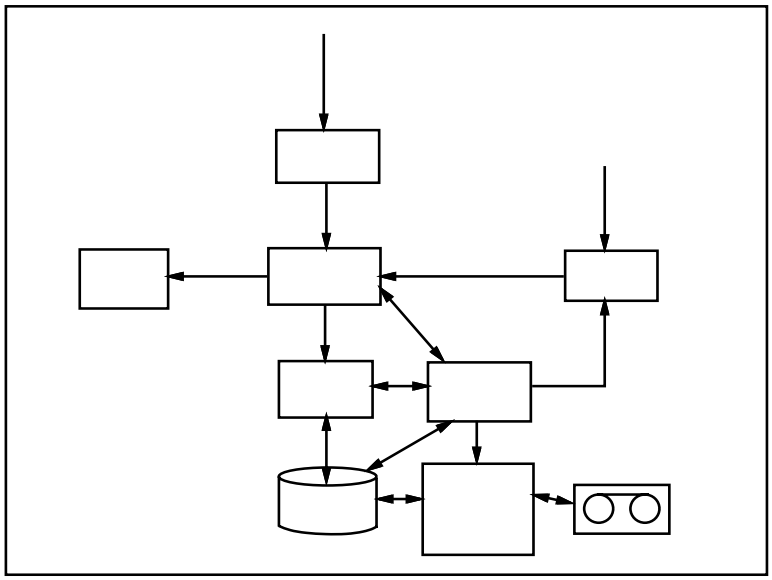

Figure 3: *Software Architecture of MDDS Server*

performed to fetch the required data from the tape. Since the seek is linear and not random like disks, the seek time is also high — of the order of a few tens of seconds. Because of the high mount plus seek times, accesses to data located on tapes must be carefully planned to minimize I/O delays. The secondary and the primary media function as a two level cache with one difference — the secondary cache is non-volatile while the primary cache is volatile. This has implications for logging and recovery. Traditional strategies can be used for data transfer between secondary and primary storage and for data/transaction management in the primary. Hence new strategies are to be developed mainly for data transfer between tertiary and secondary storage and for data/transaction management on the secondary.

The software architecture for the MDDS server is shown in figure 3. New articles are first analyzed by the analyzer that extracts the required metadata, *i.e.*, the keywords, and forms the article ID-sets and then submits an update transaction to the transaction manager. Queries that arrive from various clients are examined by the query processor and it submits query transactions to the transaction manager.

The lock manager implements the concurrency control scheme. The data manager performs the required read and write operations on the metadata. Metadata requested by the query processor is obtained from the data manager. If the required data is not on the disk, then the data manager requests the tape scheduler to transfer it from the tape to the disk. The tape scheduler schedules transfers of the requested data between the tape and the disk. The log manager oversees the operations of the data manager and logs the occurrence of significant events during updates. Both the log manager and the data manager operate on data in the disk and the buffer (primary).

We have built a prototype based on our bibliography search system following the above architecture. The bibliography search system has been in use on our world wide web site[5] for a few months. The starting size of our metadatabase is about 1GB. In order to evaluate the system in a real world situation,

[5](at URL `http://www-ccs.cs.umass.edu/db/bib-search.html`). This system has been used for querying over the WWW more than 5000 times over the last 2 months.

```
Add items 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 4: *Establishing conflict set for a query and passing hints to the transaction manager*

most of the queries have been taken from the access log of the web server. The system has been built on a SunSparc 20 multiprocessor workstation with 96 MB of main memory[6] running Solaris 2.3 version of Unix. All the components shown in figure 3 have been implemented as independent Unix processes communicating using IPC messages. The total disk space for the system was restricted to about 200 MB. To make the implementation very efficient, hash tables have been used for several functions — in the lock manager for tracking granted and waiting requests and in the transaction manager to track completed operations, to name just two.

# 4 Concurrency Control

Our concurrency control scheme uses *latches*, *i.e.*, short duration locks, for reads and writes. If we are interested only in correctness and "recency" of article is not a concern then just using latches is sufficient. However, since recency is of interest to us, we could do the following: When locks are requested for a query transaction, the lock manager performs a check to see if any other concurrent update transaction has already updated any data items needed by the query (transaction timestamp can be used to determine this precedence order). If there are such items, then latches for the rest of the items needed by a the query are also granted to the update transaction first. That is, for data items common to an update and a query, the update write locks the metadata first and performs the write before the query can read lock the metadata and read it. Since the data items are not dependent on one another, the performance of this scheme can be further improved by using a read-past and write-past technique. This allows a transaction to perform the next operation even if the current operation has to wait for a lock. The operations waiting for a lock can be performed later when the lock is granted. However this still means that the query has to wait for an unspecified amount of time till the last latch is granted.

Hence to reduce the response time of a query, by exploiting knowledge about data and transactions we go one step further. Recall that after the articles have been analyzed, the read sets and write sets for queries and updates are known and an item in accessed only once. Hence we introduce a new locking technique called *latching with operation-reordering*. The algorithms are shown in figures 4 and 5. and works as follows: When a transaction arrives, the transaction manager informs the lock manager about all the items that will be accessed by the transaction. The lock manager also keeps track of all the active update transactions. Thus, when a query arrives the lock manager will perform a conflict check with all active update transactions. If there are items common to a query and a concurrently

[6]The total size of virtual memory is 226 MB although most of it is occupied by the system processes, X Window Server and other daemons.

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


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

Figure 5: *Algorithms for reordering write operations*

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 items at the earliest. The transaction manager in turn reorders the operations of the update transaction and submits operations on the common items first. Once one of these common items 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 waiting time for queries is minimized and they also see the most recent articles.

While several read latches can be issued concurrently, only one write latch is issued at a time. The latching mechanisms and data structures are similar to the locking data structures described in [GR93]. The main data structure is a hash table and each hash chain contains a number of latched items. The latch header for each item contains a list of transactions for whom a latches have been granted and a waiting queue for transactions that need a latch on that item. When transactions to whom latches have been granted finish their work, the next item from the waiting queue is granted a latch. The operations on the latch table are serialized. We have made some enhancements to the latch hash table to maintain additional information required for operation-reordering. For each latchable item, 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 data item after obtaining a write latch. A latch for a keyword is available if there is no entry corresponding to that keyword in the table. The latch requests corresponding to four keywords are shown in figure 6. We now explain how latching is

| Keyword | Latches | | |
|---|---|---|---|
| | Processed | Granted | Waiting |
| .. | .. | .. | .. |
| Heterogeneous | U1, U2 | Q3 | U5 |
| .. | .. | .. | .. |
| Legacy | U3 | Q4, Q6 | U6 |
| .. | .. | .. | .. |
| Multidatabase | U3 | U4 | Q5 |
| .. | .. | .. | .. |
| Optimization | U2, U3 | Q4 | U6 |
| .. | .. | .. | .. |

Figure 6: *Latch Table*

done for queries and updates and in the process show how the entries in the table are utilized.

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

When a query requests a latch, the latch is granted only if it is safe 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, if nobody else is waiting the entry is removed from the latch table. Note that only updates are inserted into the processed list.

Note that the above scheme ensures that the query results reflect all articles that have been or will be 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, as our experimental results show, the payoffs are negligible, especially since queries are short compared to the update transactions.

Our concurrency control and locking scheme can also be used for static continuous queries in systems like SIFT and Tapestry. When a new user submits a profile or a user changes her/his profile, our techniques can be used to ensure that the user does not miss any article during or immediately after the profile submission/modification. Our techniques can also used for long duration transactions in other environments where the read and write sets are known and there are no dependencies between data items.

## 4.1 Performance Results

Using our prototype system (see section 3 for details) we measured the performance of the schemes with different multi-programming levels (MPLs). 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 have set the update MPL to a particular value and then varied the query MPL over a range. We have a query intensive environment and hence

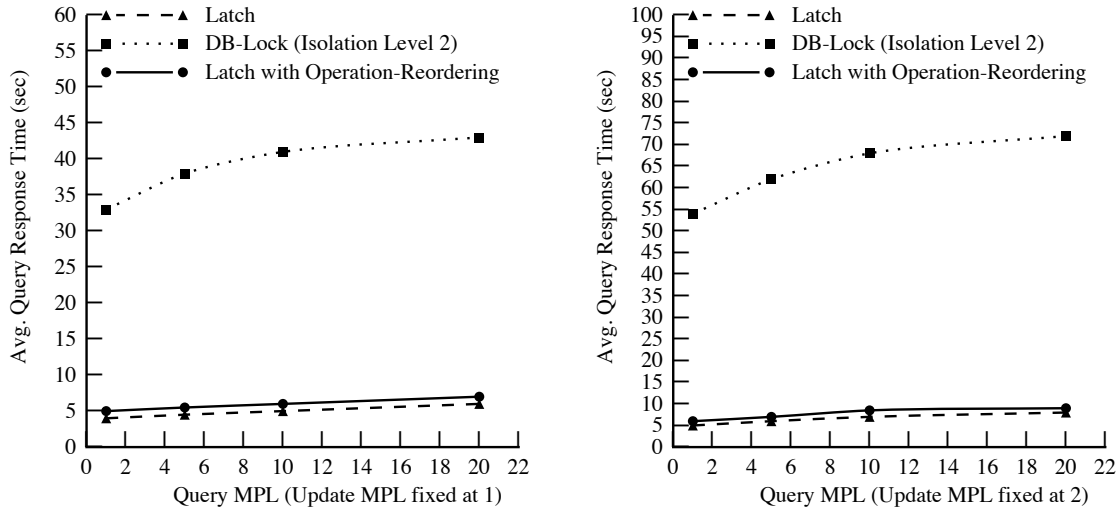


Figure 7: *Avg. response time of queries*

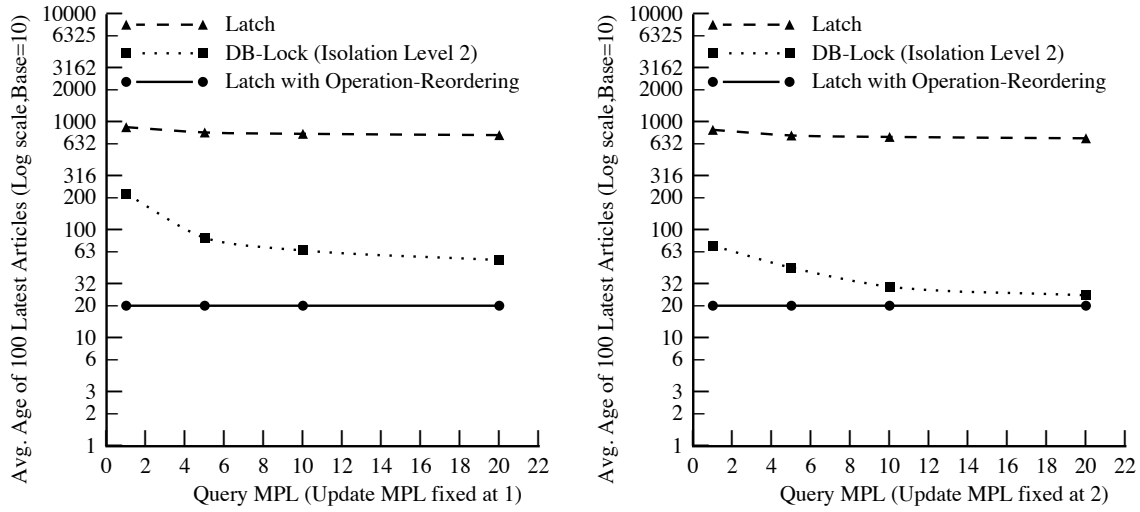


Figure 8: *Avg. age of latest 100 articles returned by queries*

there are no gaps between successive queries within a query thread. The update thread waits for 10 seconds between the completion time of one update transactions and the start of the next. So as to focus on the performance of our concurrency control scheme, for these tests, the objects (metadata) reside on the disk.

We first study the average response time of queries. Response time is the time that elapses between when the query is submitted and when the system determines the article IDs that satisfy the query. The results are shown in figure 7 (note the two graphs have a different scale along the Y axis).

- *Latching* refers to the scheme that uses latches and satisfies just the correctness requirement.
- *DB-locking* provides isolation level 2, *i.e.*, cursor stability. This was implemented to see the effect of using an off-the-shelf transaction processing approach, one most suitable for our needs.
- *Latching with operation reordering* is our scheme.

As expected we see that the response time with DB-locking is very high compared to that of latching and our scheme. The difference between our scheme and latching is small and is the order of a second. This slightly larger response time is due to (1) the cost of checking common items between queries and concurrent updates and (2) the time spent by read operations from queries waiting for write operations on those common items to be completed by the concurrent updates.

To study recency properties of the queries executed by the three schemes, we examine the average age of the top 100 articles returned by the queries. Age of an article corresponds to the difference in the arrival time of the query and the time when the metadata for the article was written. As shown in figure 8 (age shown in logarithmic scale) it can be seen that our scheme retrieves the latest articles. 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 items 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 latching scheme performs the worst since it misses most of the updates and hence retrieves less articles from the updates currently active in the system.

The DB-locking scheme performs moderately. Though on average several queries get the most recent articles 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. They are typically the objects that are written towards the end of the update transaction. Since the update transactions are typically long duration transactions the possibility of deadlocks are very high. Hence we designed the updates to access the objects in alphabetical order so that deadlocks can be avoided. 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 acquired at the beginning and released in a slow shrinking phase as the operations are completed, then the results would be slightly different. The queries will have to wait for the write locks to be released by updates almost all the time and this will definitely retrieve the latest articles but the average response time will also increase substantially.

Note that there is a slight increase in the average response time and drop in the average age of the 100 latest articles as the query MPL inreases. This is because on average queries stay slightly longer in the system. This is especially noticeable in the case of DB-locking due to higher contention (queries waiting for updates and updates waiting for queries). Also, this phenomenon is more pronounced as the update MPL increases and DB-locking experiences conflicts for write locks as well.

Our experiments show that forcing in-progress updaters to go ahead of new queries on the metadata items of interest to the query has a good payoff. However, if an update begins when a query is already in progress, with our current scheme the query does not benefit from the updates. So we tested an enhanced scheme wherein when a query is about to complete, it checks whether an update started after it began and if so, the query is retried. Our experiments indicate that such a situation is very infrequent compared to the case of a query starting after an update begins because queries are short and updates are long. As a result, average recency remains almost the same as without the extra check when a query finishes. That is, 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.

In summary we see that our latching with operation reordering can retrieve 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.

# 5 Data Migration, Logging and Recovery

In this section we present techniques for logging/recovery and migrating updates of metadata to tapes. We concentrate mainly on issues that arise due to the use of a hierarchical storage system, specifically, tapes. Since the tape mount and seek time is high, to perform these functions efficiently, data accesses to tapes should be minimized.

Updates to metadata migrate to tapes in two stages. The update is first made persistent on disk. Then it has to be transferred to the tape since the disk size is not sufficient to hold all the update. Since there are no in-place updates, the entire metadata need not be fetched from the tape before a write is performed, instead metadata is just appended to the tapes. In traditional database systems, the logs are first written to the disk and a checkpoint process runs in the background to "install" the changes on the actual data pages. Hence the changes are typically installed in the order in which they were made. If a similar scheme is used for migrating updated data onto tapes, there would be a continuous flow of data from disk to tapes. Specifically, while queries are trying to access the data they require from tapes, there will be a lot of unnecessary intervening tape accesses to store the updates onto the tapes. This will hurt the performance and hence the average response time of queries will increase considerably. Instead of taking the above *immediate migration* approach we take the following *lazy migration* approach. The updates on the disk are flushed to tape only when the corresponding tape is mounted on the tape drive of the tape library. This way, unnecessary tape accesses just for transferring the updates is avoided. However it may so happen that some of these tapes are never scheduled since none of the queries need metadata stored on that tape. To handle this situation we establish a *bound* on the size of the pending updates that need to be made to a particular tape. Hence when the size of the pending updates increases beyond this bound, a tape access is scheduled automatically. This lazy migration approach reduces the number of tape accesses for updates considerably. Hence queries would be able to get the desired data from the tapes faster, thereby improving the response time of the queries.

While most of the metadata resides on tape, some of it will be on disk and updates to some of them will also be residing on the disk. Hence it is important to track where the metadata is currently resident. For this purpose we use a data structure called the *mapping-table* which is very similar in functionality to the page-table used in memory management. The mapping-table essentially stores some metadata about the metadata and is shown in figure 9. It always resides on the disk (not on tapes). This table is hashed on the keyword. For each keyword, the mapping table contains information about the tape on

| Keyword | Tape-ID | On-Disk | Disk-Loc | Dirty | Size | Last Used |
|---|---|---|---|---|---|---|
| .. | .. | .. | .. | .. | .. | .. |
| Heterogeneous | 3 | 1 | 2235 | 1 | 34,342 | 11:24 |
| .. | .. | .. | .. | .. | .. | .. |
| Legacy | 5 | 0 | 0 | 0 | 12,957 | 11:10 |
| .. | .. | .. | .. | .. | .. | .. |
| Multidatabase | 2 | 0 | 0 | 0 | 44,813 | 11:28 |
| .. | .. | .. | .. | .. | .. | .. |
| Optimization | 8 | 1 | 2972 | 0 | 98,268 | 11:18 |
| .. | .. | .. | .. | .. | .. | .. |

Figure 9: *Mapping-Table*

which the keyword's metadata is stored, if a copy of it is on disk (bit field) and if so the disk address is also stored. It also contains information about the size of the metadata (which may be used in query optimization, *e.g.*, for finding the common items across two article ID-sets) and the time of last usage for data replacement (eviction) purpose on disk. There is also a dirty-bit which indicates that there has been an update to that metadata on the disk and the update has not yet been flushed to the tape. If an entry does not exist for a keyword then no instances of that keyword currently exist in any of the articles in the database.

Now we discuss how updates and queries are processed using this mapping-table. When the metadata is updated, the incremental update is stored on the disk in an update list (which is again hashed) and the corresponding dirty-bit in the mapping table is set to 1. It is also possible that a copy of the metadata is cached on the disk due to access from a query. In such cases, if a new query arrives, it is important to ensure that a new query sees the latest version of the metadata. Hence before a query reads the metadata, the mapping-table is checked to see if the dirty-bit is set. If it is set, then both the cached metadata for the keyword and the incremental update from the update list has to be read. Consider another situation wherein a query arrives when the update resides on the disk but the metadata resides on the tape. When the tape that holds the metadata for the keyword is mounted for access by a query, the required metadata is first copied from the tape to the disk. If the dirty-bit is set, the query also reads the incremental update on disk. Only then the incremental update is flushed to the tape and the dirty-bit is reset.

Next we discuss logging/recovery requirements. Recall that in order to handle failures that may occur during a long update, the update can be programmed as a mini-batch. The primary and the secondary act as a two-level cache. Traditional logging techniques deal with ensuring consistency and durability when data is transferred from primary to the secondary. Here we concentrate on such issues for data transfers from secondary to tertiary. Since the secondary cache is non-volatile, logging requirements are quite different — the data values need not be logged since it resides on the disk and only the actions need to be logged. The logs are minimal in size and are typically stored on the disk. The log space can be reclaimed periodically.

The main event that is of importance is the transfer of data between the disk and the tape. After the "transfer completed" message is received from the tape-drive, an action completed message is written to the log. Only then the corresponding bit is changed in the mapping table. If a system failure occurs after the initiation but before the action complete message is written to the log, then the action has to be redone at restart time. Since the mapping-table is the only data structure which knows the updates that have percolated to tape, changes to it (made in memory) need to be logged to handle system failures. Traditional database logging techniques are used to ensure the persistence of the mapping-table.

Persistent savepoints may be necessary even for computations done outside the transactions. Examples of these include incremental metadata extracted from the analysis phase, partial results of compute intensive queries (queries that require processing several metadata items). Logical failures do not occur at any stage during updates in memory and hence there is no need to store before images of data.

### 5.1 Performance Results

Though we do not have a tape library in our prototype system but only a tape drive, we have been able to simulate accesses to a tape library. We do this by having a virtual tapeID attached to every object ID and by creating a random delay (in the tape-scheduler) with a mean of 10 seconds that simulates

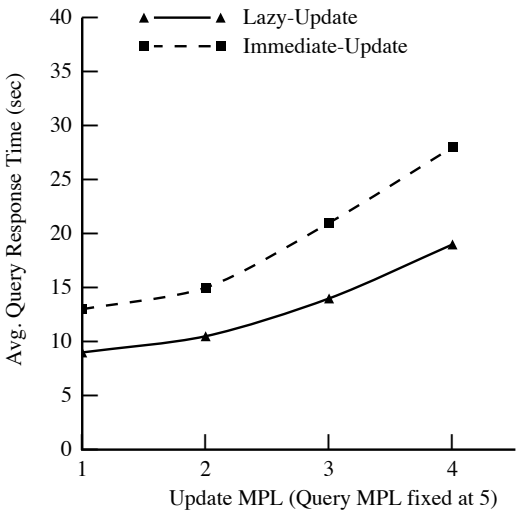


Figure 10: *Avg. response time of queries*

tape mount delays. Since the access characteristics of metadata and the articles are different, MDDSs should be equipped with two tape libraries — one for the articles and the other for metadata. Because we are concerned only with metadata management, we focus only on the tape library that holds the metadata.

We studied the average response time of queries for the two migration techniques we discussed earlier — immediate and lazy. These tests were performed at a fixed query MPL of 5. The results are shown in figure 10. The lazy migration scheme clearly performs better than the immediate scheme. We can also observe that the response time of queries increases non-linearly for both the schemes with the increase in update MPL. The increase however is more pronounced for the immediate update technique. This is because of the very large number of intervening tape accesses from updates. Thus it is clear that to improve response time of queries in a hierarchical storage environment, updates have to be migrated lazily from disk to tape.

It is not difficult to see that if the tape mount delays are larger than the assumed average of 10 secs then the response times will also increase for both schemes. Further, the performance gap between the two schemes will also increase.

# 6 Multiple-Query Optimization and Scheduling Tape Accesses

In this section we discuss query optimization techniques to reduce the response time of queries. We primarily focus on multiple-query optimization techniques with the goal of minimizing the number of tape swaps and through that reducing the average query response time. We do this by considering 4 different aspects — arrival time of queries, metadata needed by queries, metadata to tape mapping and the tape currently mounted in the tape reader.

Query optimization in hierarchical storage systems consist of two parts — one to optimize data transfers from disk to buffer (primary optimizer) and the other from tape to disk (secondary optimizer). Traditional query optimization schemes have addressed the primary optimizer where due to buffer size constraints, information about the size of the data and the buffer size is used to fetch the data/metadata

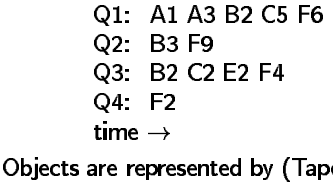


Figure 11: *Objects Accessed by each query (Objects cached on disk not shown)*

in a certain order for processing the queries. We concentrate only on the secondary optimizer. Since in a MDDS, only specific metadata has to be brought to the disk (and not the articles themselves), there are hardly any ordering constraints imposed due to disk space limitations. Thus we can reorder object accesses dynamically by using information about the query arrival time and the ID of the tape mounted currently. Once the metadata is on the disk, they can be fetched from the disk to the buffer in the order specified by the primary optimizer. This is done when for example common articles have to be identified from a set of metadata. Since some of the metadata might already exist on the disk, only metadata that is to be fetched from the tape is considered in the secondary optimizer. Transfer time for metadata from tape to disk is negligible compares to the tape mount/seek time and hence is ignored for our present discussion. The effect of data compression is also ignored since it only affects the transfer time. For all the examples we present in this section, we assume an average tape exchange time of 10 seconds and an average tape seek time of 20 seconds. In the figures, LD refers to a tape load (mount) and SE refers to a seek.

Consider for example the four different queries shown in figure 11 that need to access objects from tape. The objects required by the queries are shown as a concatenation of the tapeId and the objectID on the tape. Figure 12(a) shows a simple strategy where objects required by queries are fetched on a first-come first-server basis. This is improved in figure 12(b), by rearranging the list of objects in the access plan such that all objects required from a tape are fetched at a stretch once the tape is mounted. This will prevent unnecessary tape swaps and seeks on the tape (figure 12(a)) and thus optimizes tape I/O time. Also, if two queries need the same object, in this second case a single retrieval of the object is sufficient. The average query access time in the first case is 250 and the second case is 240 seconds and hence there is not much of an improvement. Thus such naive strategies are not sufficient to reduce

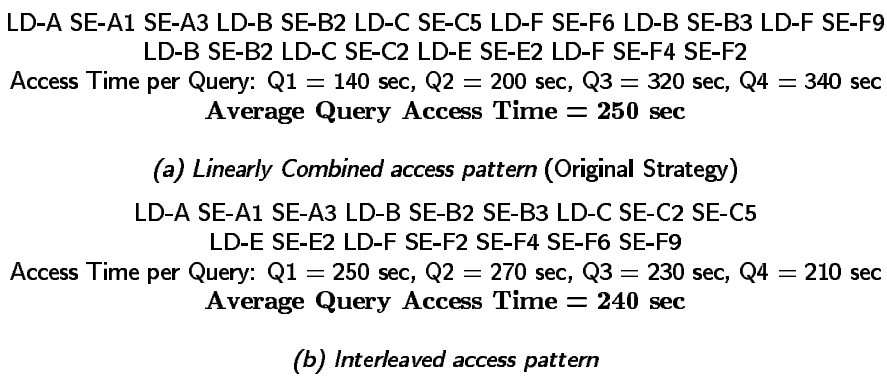


Figure 12: *Combined access patterns (LD indicate Load, SE indicates seek)*

LD-F SE-F2 SE-F4 SE-F6 SE-F9 LD-B SE-B2 SE-B3
LD-E SE-E2 LD-C SE-C2 SE-C5 LD-A SE-A1 SE-A3
Access Time per Query: Q1 = 270 sec, Q2 = 140 sec, Q3 = 200 sec, Q4 = 30 sec
**Average Query Access Time = 160 sec (1.56 times faster than original)**

Figure 13: *Optimized Access Pattern after rearranging objects*

```
Q1:    A1  A3  B2  C5  F6
Q2:        B3  F9
Q3:            B2  C2  E2  F4
Q4:                F2
time →
```

Figure 14: *Dynamic Arrival Pattern of Queries*

the response time of queries and we need more efficient strategies for multiple-query optimization.

Now we present our strategy of rearranging the object accesses such that the average query access time is reduced. The key idea here is to rearrange the objects in the access plan such that both the objectives of reducing the average query access time and reducing the number of tape exchanges (swaps) are met. We select a query that requires the least number of object accesses and retrieve those objects first. While retrieving these objects, objects required by other queries from the same tape are also retrieved. Then the next shortest query is considered and so on. The actual object access pattern for the queries is shown in figure 13. Unlike previous strategies, queries which require fewer object accesses have shorter access times. Hence as shown in the figure the average access time of queries is 160 seconds, an improvement by a factor of 1.56 over the original strategy.

However in reality queries arrive in a dynamic fashion as shown in figure 14. Hence we modify our optimization strategy such that information about the the *arrival sequence* of queries and the tape currently mounted is made use of. Before a tape is dismounted a check is performed to see if any of the newly arrived queries access the data on the tape being dismounted. If so the required objects are retrieved before the tape is dismounted. The decision to mount the next tape is based on the number of objects required by queries. We select a query which has the least number of objects to be accessed yet. Then we schedule a tape that contains objects required by the selected query. In case of a tie, the tape which is in more demand is selected, *i.e.*, contains objects required by other queries as well. This is called the *shortest query first* algorithm and is shown in figure 15.

Our new strategy as applied to dynamically arriving queries is shown in figure 16. As shown in the figure the average access time of queries is 125 seconds an improvement by a factor of 2 over the original strategies (of course in the present case queries arrive dynamically). This algorithm is "fair"

```
do forever:
  mount the next tape after making a selection as follows:
    select query with least number of objects yet to be retrieved;
    If several tapes qualify, select the one which is needed most by other queries as well;
    before unmounting a tape check if any other newly arrived query needs data from that tape;
    if so transfer the required data from tape to disk;
```

Figure 15: *Shortest-Query-First Algorithm*

LD-A SE-A1 SE-A3 LD-B SE-B3 SE-B2 LD-F SE-F9 SE-F2 SE-F4 SE-F6
LD-C SE-C5 SE-C2 LD-E SE-E2
Access Time per Query: Q1 = 220 - 0 = 220 sec, Q2 = 130 - 60 = 70 sec,
Q3 = 270 - 80 = 190 sec, Q4 = 150 - 130 = 20 sec
**Average Query Access Time = 125 sec (2 times faster than original)**

Figure 16: *Access Pattern during Dynamic Optimization*

```
do forever:
  mount the next tape after making a selection as follows:
    select query with least value for (arrival time + estimated time to retrieve the objects required by the query);
    If several tapes qualify, select the one which is needed most by other queries as well;
    before unmounting a tape check if any other newly arrived query needs data from that tape;
    if so transfer the required data from tape to disk;
```

Figure 17: *Earliest-Expected-Completion-Time-First Algorithm*

since the response time actually depends on the number of objects accessed. However consider a long query which requires tape access to several objects, none of which are accessed by the short queries that keep coming. If the shortest-query-first algorithm is used, the long query will starve since the tape which have objects needed by the long query may be mounted very late. Hence it is important to consider the *arrival time* of the queries as well. We enhance the shortest-query-first algorithm to include the arrival time of query. A query which has the least value for the sum of the [arrival time + estimated time to retrieve the rest of the objects of the query (as of the current instant)] is selected and a tape which contains the objects needed by that query is mounted and the required objects retrieved. We called this the *earliest expected completion time first* algorithm and is shown in figure 17. The retrieval time is estimated based on the average mount and seek time of tapes on the specific system.

Note that the above scheme handles the tape accesses arising from the need to migrate updates as well. Basically, the migration process sends its requests to the tape scheduler and the tape scheduler handles these in conjunction with requests of queries.

It is very important to decide when to stop accessing objects from a tape and exchange tapes so that queries requiring access to other tapes do not starve. For this purpose a threshold limit is established and if the number of objects accessed from a tape is more than the threshold, the tape is exchanged and mounted again later after object requests for other pending queries have been satisfied.

Our algorithm has been developed assuming there is a single tape reader to read the tapes. Recent trends indicate that such large tape libraries may be equipped with multiple tape readers. Excepting for the fact that several tapes can be accessed concurrently thus reducing tape I/O time, there are no other special issues and our algorithm can be easily extended to handle such systems.

Our query optimization strategies are general and can be adapted to other types of tertiary devices as well. For example in the case of optical disks, disk exchange time (8 sec) is of a large magnitude compared to the seek time (0.1 sec). Hence our strategies will prove to be very beneficial for optical disc jukeboxes. Our optimization strategies can also be used in RDBMS/ODBMS environments that use hierarchical storage systems to answer queries that just need accesses to indexes (queries requiring access only to index fields).

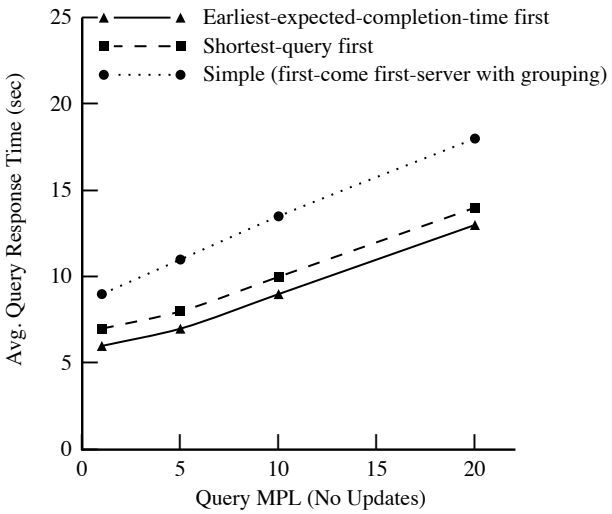


Figure 18: *Results of tape scheduling tests*

### 6.1 Performance Results

The average response time of queries was studied for three different schemes — simple (first-come first-serve with grouping), shortest-query first and the earliest-expected-completion-time first. The results are shown in figure 18. The average response time of the queries increases almost linearly with the query MPL. The earliest-expected-completion-time first scheme performs the best followed by shortest-query first scheme and the simple scheme performs the worst. The simple scheme performs badly because it just mounts a tape from which data is needed and fetches from the tape objects required by other queries as well and does not consider the number of objects needed by queries or their arrival time. Thus it does not use "information" regarding which queries should finish first and which can finish later. The shortest-query first scheme performs better; it uses information about the number of objects required by the queries and hence queries which need fewer object finish quicker. The earliest-expected-completion-time first scheme performs the best; it considers the arrival time of queries and the objects needed by them. It attempts to be fair to both short and long queries. Hence it wins over the shortest-query first scheme where some long queries can starve. Thus we see that by carefully planning accesses using knowledge about the tertiary device and the queries, the average response time of queries can be reduced considerably.

## 7 Related Work

In this section we relate our work to previously proposed concurrency control enhancements, recovery in hierarchical storage systems, and multiple query optimization.

Recall that our concurrency control scheme uses latching, exploits information about the read sets of queries and write sets of updates, makes use of the commutativity of appends and increments, and benefits from the ability to reorder update operations. Thus, whereas in static locking (conservative 2PL) [BHG87] a transaction obtains *all* locks before it submits any of its operations and as soon as the last operations is done all the locks may be released, our locks are of short duration and our scheme is non two-phase. Our scheme is different from altruistic locking [SGMS94] since we do not

use any long-term locks and we exploit update and query semantics to reorder update operations. This allows the updates to perform the operations and release the write locks early. The read-past and write-past capabilities allow queries to proceed faster because they can obtain other locks that are available while waiting for a particular lock. Even though we are exploiting the structure and semantics of metadata which are akin to index structures, it should be clear that simpler solutions than those developed for concurrency control of B-tree based index structures [Moh90, ML92] suffice because of the independence of different metadata items. The compensation based techniques discussed in [SC92] to handle queries or make changes to access structures concurrently with ongoing updates is not suited for our environment. Applying the technique will be very inefficient for MDDS metadata since updates hold long term write locks and queries will have to wait till the conflicting updates complete to determine the final outcome of the query. As we mentioned in section 4 the additional payoffs from adding such a compensation to our latching with reordering approach is negligible. The online index building technique described in [MN92] concentrates mainly on consistently handling duplicates and deletions of keys from the index as the index is being built. However in our environment such situations do not arise.

Issues related to hierarchical storage systems are discussed in [FR94, G+95] but they mainly focus on storage design for supporting continuous delivery of multimedia data. Recently database technology, specifically transaction management, has been applied for mass storage management in the context of ADSM (IBM's ADSTAR Distributed Storage Manager [CRH95]). The nature of their applications and requirements are quite different from ours. While we focus on exploiting metadata and update semantics for metadata management, they try to achieve transactional semantics for archiving and transferring files between storage devices.

In the context of the POSTGRES system, [SS95] discusses multiple query optimization for data residing on tertiary storage. It primarily concentrates on caching and scheduling strategies for efficiently processing relational two-way joins. Their main concern is the limited size of the disk-cache since not just the indexes but entire relations may have to be fetched to the secondary storage. In an MDDS, processing queries requires accesses just to specific metadata and not the data itself and hence the size of the disk-cache is not a major consideration especially for the typically short queries. Only the response time is critical.

# 8 Scope for Future Work

In this section we discuss some of the issues which came up during our discussion but were not addressed since they are orthogonal to the main focus of this paper. They relate to data management on disk/tape and system integration. Problems presented here need to be addressed to further improve the functionality and performance of MDDSs.

**Data Management on Disk:**

In a hierarchical storage system, deciding what data to cache on the disk is still an open problem. Periodic schemes that use heuristics and history are needed to cache hot data on disk to reduce tape I/Os. Defining "hot" data is difficult since it may relate to the latest data (*e.g.*, data pertaining to last 2 weeks) or most commonly asked data (*e.g.*, information about the internet) or seasonal data (*e.g.*, facts about Olympics). This definition changes over time and the periodic caching schemes are responsible for automatic migration of hot data between tapes and disks. This type of caching will have a significant impact on the performance of the system. Some caching strategies for information retrieval systems are discussed in [?].

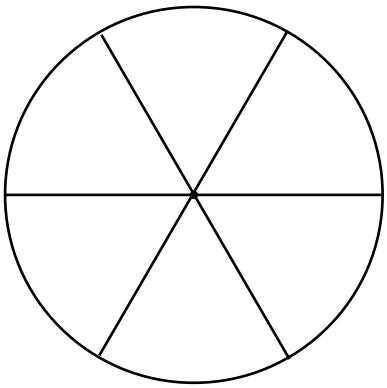

Figure 19: *Disk Space partitioning*

New schemes are needed for disk space partioning as well. The disk space has to be partitioned to hold the different types of data as shown in figure 19. The amount of space allocated for each is a critical factor and can largely dictate the performance of the system. The space requirements have to be determined after proper modeling of a number of parameters. The query area stores metadata read for queries and the partial results. Hence query area for example is approximately the product of (no-of-queries · no-of-keywords · size-of-inverted-list). The disk cache holds data cached from tape. An LRU algorithm has be used for data replacement (eviction) from the disk cache. Further studies are needed to see if any other better strategy is possible. Note that the disk cache is a read only area and none of the data from the disk cache is to be flushed to the tape. The update lists hold updates to be applied to data on tape. A threshold value is used to flush data to tape to prevent update lists from growing too big. The document processing area holds all the arriving documents and incremental inverted lists extracted after analyzing the documents before they are inserted into the database. The area allocated will depend on the arrival rate of documents and their sizes. The log files store the sequence of significant actions performed on data in the memory, on disk and on the buffer. Also the performance of queries is determined largely by quick accesses to the mapping-table and hence enough space is to be allocated such that the entire mapping table always stays on the disk (as opposed to migrating parts of it to the tape).

**Data Management on Tape:**

Due to their sequential nature, an important problem to be addressed is the location of updates on tapes. If data is to be layed out contiguously on tapes, then additional space is to be allocated for each metadata as shown in figure 20. Since metadata does not grow uniformly (some grow faster than others), it is a difficult task to decide how much space is to be allocated for each metadata. At retrieval time the seek delays will be the same irrespective of the size of the metadata, *i.e.*, smaller size metadata will incur same overheads are larger size metadata. As shown in the figure, another option is to just append new the metadata on tape in the order in which they come. This implies each metadata will be scattered throughout the length of the tape. Hence at access time, the entire tape might have to be searched. The tradeoffs involved here are to be studied and efficient schemes can be designed.

Clustering and placement strategies are also needed for data on tapes, *i.e.*, which tape a particular metadata should go to. This will depend on the access frequencies of the different metadata. Data layout issues on tertiary storage in the context of scientific databases is discussed in [?]. Since metadata keeps growing, and there are multiple metadata on a single tape, strategies are needed to handle spills,

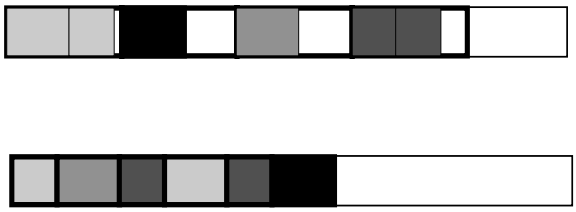

Figure 20: *Data Scattering on Tape*

*i.e.*, updates to metadata that cannot fit onto the same tape. Although some simple strategy is discussed in [CRH95] additional study is needed. If the updates are stored on a new tape then query optimization will become more complicated. An alternative is to reorganize the metdata such that the whole metadata fits on a single tape. The tradeoffs here have to be investigated thoroughly. Another problem has to do with how the metadata are organized themselves. Although B-trees are ideal for disk based systems they are difficult to maintain on tapes. For example it is very complicated to handle a node split of a B-tree on tape. Hence new access methods may have to be developed for metadata on tapes. Special algorithms like the elevator scheduling algorithms for disk scheduling can be developed to reduce seek time on tapes (by knowing current position of head on the tape). Such schemes can also improve the performance of the system as it can avoid unecessary rewinding to the beiginning of tape. Finally in our technique, we append data at the end while some information retrieval systems try to keep the document IDs sorted based on occurence frequency. Hence special schemes might have to be developed to store, manage and retrieve sorted data from tapes.

**System Integration Issues:**

One of the important issues that need to be addressed is how can the techniques we proposed be put into practice. They can be developed as a new class of DBMS on integrated with current commerical DBMS. To integrate them into current commercial DBMS, several componenets have to be modified, including the lock manager, transaction manager, index manager, resource manager, query optimizer and the log manager. In the query optimizer, even if a query requires access to metadata of several keywords, in our scheme the metadata was fetched to disk at a stretch without considering the possibility of double buffering, *i.e.*, pipelining data from tape to disk and disk to memory. Queries can be processed even faster if double buffering is done since they can be processed partially even before all the metadata is retrieved.

Another key retrieval aspect that has an impact on system performance is direct transfer of data from tape to memory. If data is transferred directly from tape to memory, then there is considerable savings in time since additional copying to and reading from the disk is avoided. This can be used in our system for example in the case of the document (as opposed to metadata) tape library since no additional processing is involved in retrieving documents given the document ID. It is clear that to process queries, updating metadata and storing partial results, metadata for queries have to be brought to the disk. However the documents themselves need not be brought to the disk and instead can be directly transferred from tape to memory. Direct transfer can be exploited when the results of the query contain only static objects and the size of the objects is within a certain limit. However this is not possible in cases where the result of a query contains large objets containing dynamic media. The

tape library will become a bottleneck since no other objects can be retrieved while this transfer takes place. Since the time taken for the transfer will be large, several other queries needing access to object on other tapes will have to wait. Hence in such cases it is better to transfer data from tape to disk and from disk to memory. An incentive to bring data from tape to disk before is to cache on the disk some of the objects which can be shared between concurrent queries. Hence these tradeoffs have to be studied. In the case if audio and video objects, admission control and bandwidth allocation may also needed to transfer the objects to the client.

# 9 Conclusions

MDDSs of the future will have to manage huge amounts of growing data and hence they have to use hierarchical systems to store data in a cost-effective manner. The size of the data, the delay characteristics of the tertiary storage devices and the performance requirements render traditional techniques inappropriate for several data management functions. In this paper we focused on developing efficient transaction management and query processing techniques for MDDSs. Our specific contributions are as follows:

- We analyzed the characteristics of transactions in MDDS environments by studying queries and updates on metadata and determined the ACID properties. Our observation is that traditional transactions and transaction processing techniques are unnecessarily restrictive for these environments and can degrade performance.
- We proposed a concurrency control scheme which exploits data and transaction knowledge, uses short term locks for both reading and writing, and allows dynamic reordering of accesses to achieve high performance.
- We analyzed how data can be made persistent on tapes along with the logging requirements for recovery from system failures. By migrating updates from disks to tape in a lazy fashion, our scheme avoids a lot on costly tape I/Os that interfere with tape I/O requests from queries. Thus the response time of queries is improved.
- We also developed a new multiple-query optimization technique for queries needing access to metadata on tapes. By considering the arrival time and data requirements of queries in addition to the possibility of tape swaps, our query optimization algorithm is capable of considerably reducing the average response time of queries.
- 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. Due to limitations in our computing resources, metadata size was confined to atmost 1.25 GB. However we believe that similar performance improvements will be observed even in larger metadatabases.

By concentrating on correctness and system related performance issues, our work complements the work done thus far in information retrieval for building efficient MDDSs. All of our techniques are general enough to be used independently, however maximum benefits can be obtained by combining them. During the course of our work, several other new issues surfaced. These relate to the management of data within the disks and the tapes as well as system integration. We have presented them separately in the previous section thus providing some directions for future work.

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, and of the data storage devices. Nevertheless the practical impact of these ideas has been shown to be significant.

## References


[ACF+94] M. Arya, W. Cody, C. Faloutsos, J. Richardson, and A. Toga. QBISM: Extending a DBMS to support 3D medical images. In *Proc. IEEE Int'l. Conf. on Data Eng.*, page 314, Houston, TX, February 1994.

[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.

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

[Chi94] T. Chiueh. Content-Based Image Indexing. In *Proc. of Intl. Conference on Very Large Databases (VLDB)*, pages 583–593, Santiago, Chile, 1994.

[CHL93] M. Carey, L. Haas, and M. Livny. Tapes Hold Data, Too: Challenges of Tuples on Tertiary Store. In *Proc. of SIGMOD Intl. Conference on Management of Data*, pages 413–419, 1993.

[CLP94] T. Chua, S. Lim, and H. Pung. Content Based Retrieval of Segmented Images. In *Proc. of ACM Multimedia*, pages 211–218, 1994.

[CRH95] L.F. Cabrera, R. Rees, and W. Hineman. Applying Database Technology in the ADSM Mass Storage System. In *Proc. of Intl. Conference on Very Large Databases (VLDB)*, Zurich, 1995.

[Cro89] W. B. Croft. Research and development in information retrieval. *ACM Trans. on Inf. Sys.*, 7(3):181, 1989.

[Fal85] C. Faloutsos. Access Methods for Text. *ACM Computing Survey*, 17:50–74, 1985.

[FR94] C. Federighi and L. Rowe. A distributed hierarchical storage manager for a video-on-demand system. In *IS&T/SPIE Symposium on Electronic Imaging Science and Technology*, San Jose, CA, February 1994. SPIE.

[G+95] S. Ghandeharizadeh et al. On configuring hierarchical storage managers. Technical Report 95-601, Computer Science Department, University of Southern California, Los Angeles, CA, 1995.

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

[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 concurrencty 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. Also published in/as: IBM Almaden Res.Ctr, Res.R. No.RJ7008, Mar.1990, 27pp.

[SB94] A. Strawman and F. Bretherton. A Reference Model for Metadata. In *http://www.llnl.gov/liv_comp/metadata/papers/whitepaper-bretherton.html*, University of Wisconsin, March 1994.

[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.

[SGMS94] K. Salem, H. Garcia-Molina, and J. Shands. Altruistic locking. *ACM Trans. on Database Sys.*, 19(1):117, March 1994.

[SS95] S. Sarawagi and M. Stonebraker. Query Processing in Tertiary Memory Databases. In *Proc. of Intl. Conference on Very Large Databases (VLDB)*, Zurich, 1995.

[SSU90] A. Silberschatz, M. Stonebraker, and J. Ullman. Database systems: Achievements and opportunities. *SIGMOD Record*, 19(4):6–22, December 1990.

[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.

[VCC95] S.R. Vasanthakumar, J. Callan, and W.B. Croft. Integrating INQUERY with an RDBMS to Support Text Retrieval. In *ACM SIGIR Conference on Research and Development in Information Retrieval Post-Conference Workshop on Information Retrieval and Databases*, Seattle, WA, 1995.

[Wor94] MDDS Workgroup. Proceedings of the Massive Digital Data Systems Workshop, February 1994.

[YGM95a] T. W. Yan and H. Garcia-Molina. SIFT - A Toll 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.