Archiving Techniques for Temporal Databases

Vram Kouramajian
Department of Computer Science
Wichita State University
Wichita, Kansas 67260-0083
vram@cs.twsu.edu

Ramez Elmasri
Computer Science Engineering Department
The University of Texas at Arlington
Arlington, Texas 76019-0015
elmasri@cse.uta.edu

Abstract
This paper describes archiving strategies for append-only temporal databases. We present a storage architecture where optical disks work in tandem with magnetic disks. Magnetic disks are used for storing current versions and recent past versions, whereas optical disks are dedicated for archiving older past versions. Similarly, temporal access structures are stored on both magnetic and optical disks.

Our migration techniques: (1) allow temporal data and access structures to span magnetic disks and optical disks; (2) minimize the overhead of the migration process by taking advantage of append-only nature of temporal databases; (3) gracefully handle object versions with very long time intervals so that the delay in the migration process is kept to minimum; and (4) ensure that no false magnetic or optical disk address lookup is performed during search operations by duplicating some closed versions on both magnetic and optical disks.

To validate our claims for the efficiency of migration techniques, we analyze the performance of temporal access structures partitioned between magnetic and optical disks. We show that the migration process has a minimal effect on the search time. Our simulation identifies important parameters, and shows how they affect the performance of the temporal access structures. These include mean of version lifespan, block size, query time interval length, and total number of versions.

1 Introduction
The main characteristic of temporal databases is that they preserve the complete history of the Universe of Discourse; that is, they follow the non deletion rule of data [Snod93, TCGJ93, JCEG94]. Thus, temporal databases require physical storage medium that can accommodate huge volumes of temporal data. This suggests that it is not economically feasible to store the entire temporal data on magnetic disks. One of the prime reasons is that unlike optical disks, magnetic disks do not have a long life and their storage capacity is limited. Another reason is that unlike current data, historical data is less likely to be referenced, and therefore a relatively slower medium such as optical disks is suitable for on-line archiving of historical data.

Write-Once Read Many (WORM) optical disks provide a cost-effective, efficient solution for organizations requiring permanent storage for large amounts of temporal data. They offer the ability to store massive quantities of data in a relatively small amount of space because of lower cost of storage per byte. They have the advantage of not suffering from deterioration unlike magnetic media; a 100 year lifespan has been predicted for WORM disks. Although the access time of optical disks is higher than that of magnetic disks, WORM disks are still considered on-line random access devices, and they offer the best choice for archiving immutable data.

Database systems take periodic backups to provide recovery from disk failures. These backups reflect some past state of the application. Whenever a failure occurs, the backups together with the log are used to restore the database to a consistent state. In temporal databases, past states of the application are kept to provide on-line access. Although this capability can be achieved by keeping full backups of the application, the performance of the resulting system will definitely be very poor. To overcome this deficiency, the issues of automatic archiving and searching of both object versions and temporal access structures must be successfully addressed, which is the topic of this paper.

In this paper, we present a storage architecture where optical disks work in tandem with magnetic disks. Magnetic disks are used for storing current versions and recent past versions, whereas optical disks are dedicated for archiving older past versions. Similarly, temporal index structures are stored on both magnetic and optical disks. The temporal index structure considered is the Monotonic B+-Tree (MBT) [ElJK92] which uses an efficient temporal access method called the time index [ElWK90, Kour94]. Although our techniques are shown to work with the MBT, they may apply as well, to other append-only structures such as the AP-tree [SeGu93].

The advantages of our migration techniques are:

1. Our storage structure allows indices for temporal data to span magnetic disks and optical disks. This results in substantial performance improvement for insert and search operations.
2. Our archiving techniques minimize the overhead of the migration process by taking advantage of append-only nature of temporal databases.

3. Our archiving techniques gracefully handle object versions with very long time intervals so that the delay in the migration process is kept to minimum. Our previous archiving solution for object versions with very long time intervals slows the migration process and adversely affects search operations over magnetic-optical storage [EIJK92].

4. Our archiving techniques ensure that no false magnetic or optical disk address lookup is performed during search operations by duplicating some closed versions on both magnetic and optical disks.

To validate our claims for the efficiency of migration techniques, we analyze the performance of temporal access structures partitioned between magnetic and optical disks. We show that the migration process has a minimal effect on the search time. Our simulation identifies important parameters, and shows how they affect the performance of the temporal access structures. These include mean of version lifespan, block size, query time interval length, and total number of versions.

This work differs from that of Kolovson and Stonebraker [KoS89] and Lomet and Salzberg [LoSa93]. Our work allows indices for historical data to span magnetic disks and optical disks, as opposed to being exclusively restricted to either medium [KoS89]. Lomet and Salzberg [LoSa93] discuss how the TSB-tree access structure [LoSa89] can provide the backup function of data recovery, a topic which is not addressed in this paper. Our goal is to keep past, current, and future states of a database on-line and readily accessible; to concurrently support normal database activity with migration process; and to reduce migration cost by taking advantage of append-only characteristics of the MBT [EIJK92].

The rest of this paper is organized as follows. Section 2 starts with a discussion of the time representation and the logical storage model used in our work; it then reviews the time index and the MBT access structures. Section 3 defines a number of concepts that will be useful in our discussion of migration techniques. Section 4 introduces the main memory data structures used to facilitate the migration process. Section 5 presents in detail the strategy for migration of object versions and access structures to WORM disk. Section 6 describes a technique for gracefully handling very long lived versions that may delay the migration process. Section 7 reports on the simulation studies. Finally, Section 8 concludes the paper.

2 Background

Research in time notation has centered around two basic symbolic models of time. One is the description of the state of the real world as continuous points in time; and the second as intervals of time. In Section 2.1 we discuss the representation of time used in this paper. The time dimension can be associated either with tuples or attributes. Section 2.2 describes the temporal representation model. Sections 2.3 and 2.4 review the time index [EiWK90, Kour94] and the MBT [EIJK92]; two access structures used in our archiving techniques.

2.1 Time Representation

Let \( T = \{t_{-\infty}, t_1, \ldots, t_{n_{\text{now}}}, \ldots, t_{+\infty} \} \) be a countably infinite set of totally ordered discrete time points, where \( t_{\text{now}} \) represents the current time point which is continuously increasing. A time point is really the smallest possible interval, which is considered atomic or indivisible. This has been called a chronon [JCEG94]. We will use the terms point and chronon interchangeably in this paper.

A time interval, denoted by \([t_1, t_2]\), is defined to be a set of consecutive chronones; that is, the totally ordered set \( \{t_1, t_1+1, \ldots, t_2-1\} \subset T \), where \( t_1 \) is the \textit{start} time and \( t_2 \) is the \textit{end} time of the interval. In general, it is preferable to use an interval with an open upper bound \([t_1, t_\infty)\), especially when dealing with an application that uses different time granularities [WI91].

2.2 Storage Model

Our access structure is defined over a storage model, called TRDB (Temporal Relational Database), which is based on object (tuple) versioning [NaAh89]. A TRDB consists of a collection of object versions; that is, \( \text{TRDB} = \{e_{11}, e_{12}, \ldots, e_{1k}, \ldots, e_{m1}, e_{m2}, \ldots, e_{mt}\} \), where \( e_{ij} \) is a version of an object \( e_i \). Each object version \( e_{ij} \) is augmented with two additional attributes \( t_s \) (valid start time) and \( t_e \) (valid end time), which define the version time interval \( [t_s, t_e] \). This represents the fact that the object version \( e_{ij} \) is valid from time point \( t_s \) till the time point before \( t_e \). An object version \( e_{ij} \) with \( e_{ij}.t_s = \text{now} \) is considered to be the current version of some object \( e_i \). An object version \( e_{ij} \) is immutable if the value assigned to its \( e_{ij}.t_e \) is less than \( \text{now} \), and it is called a closed version. A version whose \( e_{ij}.t_e \geq \text{now} \) is called an open version. (We will assume that there is at most one open version for each object.)

2.3 Review of the Time Index

The time index [Kour94] is defined over a TRDB. The basic idea behind the time index is to maintain a set of linearly ordered indexing points on the time dimension. An indexing point is created at the time points where an object version interval either starts or terminates\(^1\). The Set of all Indexing Points SIP is formally defined as follows:

\[
\text{SIP} = \{t \mid (\exists e_{ij}) \left( (e_{ij} \in \text{TRDB}) \land ((t = e_{ij}.t_s) \lor (t = e_{ij}.t_e)) \right) \}
\]

Since all the indexing points \( t \in \text{SIP} \) can be totally ordered, we can use a regular \( B^+ \)-tree\(^2\) [Corne79] to index these time points. In a temporal database, there may be a large number of version pointers associated with an indexing point, and many of them will be repeated from the previous indexing point. To reduce this redundancy and make the time index practical in terms of storage requirements, an incremental scheme

\(^1\text{Note that an interval associated with an object version } e_{ij} \text{ terminates at } t_e \text{ if the object version terminates at } t_{e-1}, \text{ since we are using open upper bounds for intervals (see Section 2.1).}

\(^2\text{The MBT, reviewed in Section 2.4, can be used as the index structure for the time index.}\)
3 Migration Strategy Concepts

This section defines a number of concepts that will be useful in our discussion of migration techniques. Section 3.1 introduces the concept of $T_m$, a time point used as the dividing criterion between magnetic and optical disks. Section 3.2 classifies closed versions based on the relative positioning of their start and end time points with respect to $T_m$; this classification is used in deciding on whether closed versions are stored on optical disk, or on magnetic disk, or on both disks. Section 3.3 classifies open versions based on whether they hold the migration of few disk blocks or a large number of disk blocks. (In Section 6 we discuss techniques that handle open versions which hold the migration of a large number of disk blocks.) Finally, Section 3.4 classifies leaf nodes based on whether they allow or hold the migration of internal nodes.

3.1 Time Point $T_m$

We choose a time point $T_m$ selected based on the least start time $\min_t$ of open versions, to be the dividing criterion between the two media partitions: magnetic and optical. We call the former media the $M$-Partition and the latter media the $O$-Partition. We differentiate between two types of addresses (pointers): $\text{ma}$ denotes a magnetic disk address, and $\text{oa}$ denotes an optical disk address.

The value of $\min_t$ is defined as:

$$\min_t = \min \{ e_t | e_t \geq \text{now} \}$$

The time point $T_m$ is given the value of the leading entry $t_l$ of the leaf node in which the indexing point $\min_t$ is located. Hence, $T_m$ is defined as follows:

$$T_m = t_l (\min_t \in \text{leaf node} n)$$

The justification behind selecting the time value $T_m$ as the dividing criterion is that all search operations on time values less than $T_m$ will only lead to closed versions. The value of $T_m$ can be either queried from the database or computed during the migration process. We choose the latter approach since querying the value of $T_m$ is a very expensive operation. (The data structure, called the $T_m$ queue, used in the computation is introduced in Section 4.)

3.2 Different Types of Closed Versions

There are three types of closed versions:

1. History Version: A history version $e_{\text{history}}$ is a closed version that satisfies the property:

$$e_{\text{history}}.t_s \leq T_m.$$ 

A history version is only stored on and referenced from the $O$-Partition.

2. Spanned Version: A spanned version $e_{\text{span}}$ is a closed version that satisfies the property:

$$(e_{\text{span}}.t_s < T_m) \land (e_{\text{span}}.t_e > T_m) \land (e_{\text{span}}.t_e < \text{now}).$$

A spanned version is stored on and referenced from both the $O$-Partition and the $M$-Partition.

3. Non-Spanned Version: A non-spanned version $e_{\text{non-span}}$ is a closed version that satisfies the property:

$$e_{\text{non-span}}.t_s \geq T_m \land e_{\text{non-span}}.t_e < \text{now}.$$ 

A non-spanned version is stored on both the $O$-Partition and the $M$-Partition but it is only referenced from the $M$-Partition.
History versions are not duplicated, whereas spanned versions and non-spanned versions are duplicated on both optical and magnetic disks. Duplicating spanned versions reduces search time by limiting certain queries to exclusively access either optical disk or magnetic disk. Duplicating non-spanned versions speeds up the migration of version pointers considerably. For each non-spanned version, an entry is appended to a mapping table specifying its magnetic-optical address correspondence. (The table, called the MOAMT, is described in Section 4.) By using this table, converting version pointers from magnetic to optical addresses is quite efficient, since it only requires a simple in-memory table lookup as described in Section 5.

### 3.3 Different Types of Open Versions

There are two types of open versions:

1. **Long Lived Version (LLV):** A long lived version is an open version that holds the migration of buckets that refer to non-spanned versions.

2. **Very Long Lived Version (VLLV):** A very long lived version is an open version that holds the migration of a (large) number of buckets.

The specific number of buckets used in the classification of open versions as VLLVs is application-dependent and can be adjusted based on a number of policy considerations. An example of a policy consideration is the wish to keep object versions for longer periods of time on magnetic disk by choosing a large value for n.

The presence of LLVs is a desirable feature in our model since these open versions hold the migration of non-spanned versions, which are more likely to be referenced than (older) spanned and history versions. However, the presence of VLLVs complicates the migration process since these open versions hold the migration of spanned versions and subsequently MBTs to optical disk. Two techniques for identifying VLLVs are presented in Section 4 and a graceful method for handling VLLVs is introduced in Section 6.

### 3.4 Closing and Non-Closing Leaf Nodes

There are two types of leaf nodes:

1. **Closing Node:** A closing node is a leaf node that serves as the last (rightmost) node of a subtree.

2. **Non-Closing Node:** A non-closing node is a leaf node which is not the last (rightmost) node of any subtree.

The migration process distinguishes between these two types of leaf nodes since an internal node is migrated only after its closing leaf node is migrated.

### 4 Data Structures

In this section, we introduce the main memory data structures used to facilitate the migration process. Section 4.1 describes a data structure, called the T, queue, which keeps track of open versions. Section 4.2 presents the mirror sector, which is used during transfer of closed versions from magnetic disks to optical disks. Finally, Section 4.3 introduces a mapping table, called the MOAMT, which keeps track of closed versions.

#### 4.1 The T Queue

We define a queue, called the T queue (see Figure 1), which keeps track of the number of open versions referenced (through SP buckets) from each leaf node. Each element of the queue is of the form:

\[ q_e = (t_l, ma, cnt) \]

where

- \( t_l \) is the value of a leading entry of a leaf node \( ln \);
- \( ma \) is a pointer to the leaf node \( ln \);
- \( cnt \) is the number of open versions referenced from the leaf node \( ln \) through SP buckets.

The head of the queue always satisfies \( q_e.t_l = T_m \) because leaf nodes whose indexing points are less than \( T_m \) refer only to closed versions. The tail of the queue corresponds to the rightmost leaf node of the MBT. There may be elements in the \( T_m \) queue that correspond to leaf nodes whose leading entry \( t_l \) is greater than \( T_m \) and who only refer to closed versions (i.e., \( cnt \) equal to zero). These elements are kept in the queue since a long \( T_m \) queue indicates the existence of very long lived versions (VLLVs). If the length of the \( T_m \) queue keeps increasing, then the head of the queue will definitely point to a leaf node that refers to VLLVs.

#### 4.2 Optical and Mirror Sectors

Since the smallest writable unit on an optical disk is a sector, it is not viable to transfer an object version from magnetic disk to optical disk as soon as it becomes closed. A closed version is assigned space on optical disk, but it will be moved only when a whole sector of closed versions is ready for migration. Our approach is:

1. to reserve an entire optical disk sector, called the optical sector, whose address is \( ossa \) and
2. to create a mirror image, called the mirror sector, of that sector on magnetic disk whose address is \( msa \) (see Figure 1).

Hence, a closed version stored in the mirror sector has a known optical disk address. The storage requirements for optical and mirror sectors are minimal since only one optical sector and one mirror sector are needed.

#### 4.3 Magnetic to Optical Address Mapping Table

To speed up the migration process, we keep a Magnetic to Optical Address Mapping Table (MOAMT) (see Figure 1), which stores the magnetic-to-optical correspondence for closed versions that are being migrated, together with their end time points; that is, each entry in the MOAMT is of the form \( (ma, oa, t_e) \), where \( ma \) is the magnetic address of a closed version, \( oa \) is the assigned optical address, and \( t_e \) is the end time of the closed version.

Whenever an open version becomes a spanned (or non-spanned) version, a new entry is reserved in the MOAMT that keeps the magnetic address, optical address, and end time point of the spanned version. Once
the spanned version becomes a history version, its entry is removed from the MOAMT. The memory requirements for MOAMT is in general small unless there are a large number of spanned and non-spanned versions. If the size of MOAMT keeps increasing, then this implies that spanned and non-spanned versions are still kept on both partitions due to the presence of VLLVs.

5 Migration of Versions, Buckets, and MBT Nodes

To minimize disk access operations during archiving, we establish a well defined order for migration. The order of migration is: closed versions, SP buckets, SC buckets, SM buckets, leaf nodes, and internal nodes. In the following subsections, we present in detail the migration process.

5.1 Object Versions

Whenever an object version is closed, it is moved to the mirror sector, declared ready for migration, and assigned a space on optical disk. Then, an entry of the form \((ma, oa, te)\) is inserted into the MOAMT, where \(ma\) is the magnetic address of the closed version, \(oa\) is the assigned optical address, and \(te\) is the end time of the closed version.

Initially, a closed version becomes a non-spanned version and is duplicated on both optical and magnetic partitions but referenced only from the \(M\).\text{Partition}. Later, it becomes a spanned version and therefore it is accessed from both partitions. Finally, it becomes a history version and it is removed from the \(M\).\text{Partition}; thus, it is only accessed from the \(O\).\text{Partition}.

5.2 The Mirror Sector

The migration process only needs to keep one mirror sector and one optical sector. The mirror sector is pinned in main memory since it is accessed whenever an object version is closed. A closed version is first moved to mirror sector and is assigned a space on the reserved optical sector. Once the mirror sector becomes full, it is written to the optical sector and a new optical sector is reserved.

5.3 The \(T_m\) Queue

The \(T_m\) queue must be maintained in main memory since it is frequently referenced and often updated. The following operations are performed on the \(T_m\) queue:

- Whenever an object version \(e_{ij}\) is closed, the \(T_m\) queue is fetched and the element \(qe_m = (t_l, ma, cnt)\) which satisfies the condition \((ge_m.t_l < e_{ij}.t_e \land e_{ij}.t_e < ge_{(m+1)}.t_l)\) is returned. Then, \(qe_m.cnt\) is decremented by one.
- Whenever a new object is inserted or an object version is updated, the tail of the \(T_m\) queue is updated by incrementing its counter by one. Unlike closing an object, updating or inserting an object does not require a traversal of the queue since new indexing points in the MBT are always added to the rightmost leaf node.
- Whenever a new leaf node is inserted, a new element is appended to the \(T_m\) queue.
- Whenever the count of the first element in the \(T_m\) queue becomes zero, all leading elements in the queue with \(cnt\) equal to zero are removed; and \(T_m\) is assigned a new value, denoted as updated\(T_m\), which is equal to the time value of the new head.

The entire migration process is triggered whenever the value of \(T_m\) is updated. Each time an element of the head of the \(T_m\) queue is removed, the MOAMT table is updated, and SP, SC, SM buckets (referenced through the removed element) are transferred to optical disks.

5.4 The MOAMT

Similar to the \(T_m\) queue, the MOAMT must be maintained in main memory since it is frequently referenced and often updated. The MOAMT speeds up the migration process because it is used for converting version pointers from magnetic address to optical address through a simple lookup of a MOAMT entry.

The MOAMT stores an entry of the form \(ent = (ma, oa, te)\) for each non-spanned or spanned version (see Figure 1). A new entry is appended into MOAMT whenever an object is closed. Old entries are removed from the MOAMT when \(T_m\) is updated. These removed entries correspond to closed objects who just became history versions; that is, their end time points are less than or equal to updated\(T_m\). Removing entries from the MOAMT is a very fast operation since these entries are sorted in increasing order of their end time values\(^5\). Thus, starting from the top of the MOAMT, those entries \(ent_l = (ma, oa, te)\) whose time values \(ent_l.t_e\) are less than or equal to updated\(T_m\) are deleted.

\(^5\)Sorting entries of the MOAMT is free since the end time point of any new appended entry is greater or equal to the end time point of the previously appended entry; this property is the result of the append-only nature of temporal databases.
5.5 Buckets of Version Pointers

To speed up the migration process, a well defined order of migration for buckets is established: SP, SC, and SM. The reasoning behind this order is based on the following observation: assume that \( \{p_1, p_2, \ldots, p_n\} \) is the set of version pointers pointing to the closed version \( e_i j \), \( p_1 \) is an incremental plus pointer, each \( p_k \) \((2 \leq k \leq n)\) is a continuous pointer, and \( p_n \) is an incremental minus pointer. This implies that (1) the indexing point (i.e., time point) corresponding to \( p_1 \) is less than the indexing points corresponding to \( p_k \) \((2 \leq k \leq n)\) and (2) the indexing point corresponding to \( p_n \) is greater than the indexing points corresponding to \( p_k \) \((1 \leq k \leq n - 1)\). Hence, once the version pointer \( p_n \) is migrated (which is stored in a SM bucket), there is no need to keep track of the version \( e_i j \) on the P-Partition; that is, there is no need to keep an entry for \( e_i j \) in the MOAMT table.

A block on magnetic disk of incremental plus or continuous version pointers is migrated if each version pointer refers to a closed version whose start time is less than \( \text{updated}_T \). A block of incremental minus pointers is migrated if each version pointer refers to a closed version whose end time is less than or equal to \( \text{updated}_T \). Locating such blocks is an expensive operation because a comprehensive search of the SP, SC, and SM buckets on the P-Partition is required. In our case, we identify buckets ready for migration by using the \( T_m \) queue. For each \( q_{ei} = (t_i, ma, cnt) \), where \( q_{ei}.t_i \) is less than \( \text{updated}_T \), the leaf node at address \( q_{ei}.ma \) is fetched, and then the SP, SC, and SM buckets associated with that leaf node are fetched. Afterwards, the magnetic address of each version pointer in the buckets is converted to an optical address by using the MOAMT table. Finally, the buckets are migrated to optical disk.

The last block of a SP, SC, or SM bucket \( lb \) declared ready for migration may be only partially filled by the version pointers of that bucket \( lb \); the rest of the block may be filled with the version pointers of the next bucket \( nb \) (see Figure 2). Our solution is to duplicate such blocks on the M-Partition and the O-Partition. However, in the case of SP or SC buckets \( lb \), it is also possible that some version pointers of the next bucket \( nb \) refer to open versions. One solution is to force close all such open versions and force create identical new open versions, similar to the technique proposed for VLLVs (see Section 6).

5.6 Leaf Nodes

A leaf node is migrated whenever the value of \( \text{updated}_T \) becomes greater than the last indexing point of the leaf node. Similar to migration of buckets, the \( T_m \) queue is used to identify leaf nodes that are ready for archiving. For each \( q_{ei} = (t_i, ma, cnt) \), where \( q_{ei}.t_i \) is less than the value of \( \text{updated}_T \), the leaf node at address \( q_{ei}.ma \) is fetched and it is subsequently declared ready for migration.

Before migrating a leaf node, all its magnetic pointers (i.e., \( ma \)) must be converted to optical pointers (i.e., \( oa \)); that is, the pointers to SP, SC, and SM buckets as well as the next leaf node. Pointers to SP, SC, and SM buckets are updated during the migration of buckets, whereas the optical disk address of the next leaf node is computed during the migration of the leaf node. The computation is done as follows:

- For a non-closing node, the optical disk address of the next leaf is the address of the next sector. This is because all non-closing leaf nodes between two closing nodes are assigned contiguous spaces on the O-Partition.

- For a closing node, the number of ancestor internal nodes belonging to the same subtree must be considered. Let \( A \) be the number of internal nodes of a closing node \( cn \), \( s_n \) be the address of the next free sector reserved for the MBT on the O-Partition, \( S \) be the size of an internal node\(^6\), and \( s_1 \) be the address of the next leaf node. The value of \( s_1 \) is determined based on the following formula:
  \[
  s_1 = s_n + A \times S
  \]
  The value of \( A \) is equal to \((h_i - 1)\), where \( h_i \) is the height of the largest full subtree that has \( cn \) as its closing node.

5.7 Internal Nodes

Migration of internal nodes occurs one node at a time in a bottom-up process. An internal node is declared ready for migration if all its time point values are less than the value of \( \text{updated}_T \). The migration of a leaf node may ripple up the tree as far as the level just below the root node. The current root node is always on the magnetic disk up to the point when a new root node is created; then, it becomes a regular internal node. Similarly, the rightmost node on each level is always on magnetic disk since it references the current state of the database.

\(^6\)We assume that the size of a node is a multiple of sectors.
6 Migration of Very Long Lived Versions (VLLVs)

Before discussing techniques for handling VLLVs, we highlight some important characteristics of the migration process:

- VLLVs hold the migration of spanned versions and MBT nodes.
- The presence of VLLVs can be confirmed in two ways: (1) when the length of the \( T_m \) queue keeps increasing and/or (2) when the size of the \( MOAMT \) table keeps increasing.
- During some time intervals only new objects may be created and no changes may occur in existing object versions; this characteristic of the application has a negative impact on the migration process.
- Some time intervals may show an increase in the number of closed versions as well as a decrease in the number of new object versions; this characteristic of the application has a positive impact on the migration process.

To achieve a timely migration, a VLLV should be assigned to optical disk. However, since an optical disk is a write-once medium, future changes to the VLLV \( t_e \) inhibits this immediate assignment. One solution proposed in [ElJK92] is to use an approach similar to that of mirror sectors. Optical disk sectors can be reserved for VLLVs, and corresponding mirror sectors can be reserved on magnetic disk. Thus a VLLV can be moved to a mirror sector, and treated as a history version. A mirror sector can be copied to optical sector whenever it becomes full and all its VLLVs become closed.

The above solution requires a large number of mirror sectors and adversely affects search operations on magnetic-optical storage. This is because each time the address of a history version is needed, the mirror sector on magnetic disk is first searched in case the object version is a VLLV. For most object versions, which are not VLLVs, a subsequent search in the \( OPartition \) is needed. This means that the search often spans both magnetic and optical disks, thus slowing the search process.

Our solution takes advantage of the append-only nature of temporal databases. The idea is to force close VLLVs and force create new open versions for VLLVs. First, we search for all VLLVs by using the \( T_m \) queue, and then assign \( (t_{last} + 1) \) to VLLV \( t_e \), where \( t_{last} \) is the largest indexing point in the MBT; that is, force close all VLLVs at \( (t_{last} + 1) \). This is possible because the end time for any VLLV cannot be less than \( (t_{last} + 1) \). Afterwards, we force create new open versions of new VLLVs for all the VLLVs with the same attribute values but with different time attributes (new \( VLLV \) \( t_e = (t_{last} + 1) \) and new \( VLLV \) \( t_e = t_{+\infty} \)). The force close operation causes the value of \( T_m \) to be modified and thus triggers the migration process. In this solution, no false address lookup is performed since any search before \( T_m \) yields an optical disk address, whereas any search after \( T_m \) yields a magnetic disk address.

7 Performance Evaluation of Partitioned Access Structures

In this section, we report on the performance evaluation of the MBT partitioned between magnetic and optical disks. To evaluate the performance of partitioned MBT, we chose two main performance criteria, namely storage requirements and search performance. Our simulation measured storage cost in terms of the number of blocks required to store the MBT and its buckets on both optical and magnetic media. The search performance was measured in number of blocks accessed (on both optical and magnetic disk) to retrieve all object versions that satisfy a certain time point or interval query. The main parameters put to study include the mean of object version lifespan, the total number of object versions stored, block size, and the time length for an interval query.

A temporal relational database was assumed to be implemented with a record–based storage system which supports tuple versioning (discussed in Section 2.2). The temporal database was initialized with 5,000 objects in some simulation runs and with 10,000 objects in others. Two block sizes of 1,024 and 4,096 bytes were studied. The lifespan of an object version was generated using a normal distribution. We used an exponential distribution to generate the events such as close an object and insert a new object. Whenever an additional 25,000 versions were created, we generated 100 point and interval queries. The interval size for interval queries was randomly generated using a normal distribution. Below, we discuss some useful results of the simulation.

Figure 3 shows the effect on storage requirements for a MBT when the mean lifespan of an object version is increased (for a block size equal to 1,024 bytes). With a large mean lifespan and a small inter–arrival rate of new objects the database will contain more number of open versions, leading to more time points inserted into the MBT. Thus object versions continue to be valid across consecutive leading entries which leads to overall increase in total storage. We observed similar results for a block size of 4,096 bytes, except that the total storage is reduced by a large margin since an object version now spans across fewer leaf nodes resulting in fewer 3C buckets and hence reduced total storage. We also observed that the total storage for the MBT and the buckets dropped form 110 Mbytes to 33 Mbytes by changing the block size from 1,024 to 4,096 bytes with 500,000 object versions in the database.

Figure 4 shows average number of blocks (optical and magnetic) accessed for a point query with a block size equal to 4,096 bytes. If we assume that each object version occupies 100 bytes on average and 500,000 object versions exist in the database, we get a file of 12,500 blocks to store all the object versions. An index search requiring only 13 blocks (for indices, buckets, and object versions access) as shown in Figure 4 is quite an improvement over a linear search assuming that the point query is focused on optical or magnetic disk. A similar behavior was observed when we simulated for a block size of 1,024 bytes. However, the average number of blocks accessed was 38, caused by the increasing rate of blocks accessed through leading entries.
Figure 3: Storage for Different Mean of Object Version Lifespan

Figure 4: Block Accesses for Temporal Point Query

Figure 5 shows the average distribution of blocks accessed separately from optical and magnetic disks for an interval query. We notice that most historical queries result in optical disk accesses, whereas most queries over recent data in magnetic disk accesses.

Figure 6 shows the distribution of number of index blocks on magnetic and optical disk. The number of index blocks on magnetic disk that reference the current versions and some recent past versions does not vary and is much less than those on optical disk. This difference keeps increasing as the size of database grows with time.

8 Conclusions

This paper described a migration strategy that keeps: (1) the current data and recent past data along with their buckets and index nodes on magnetic media and (2) older past data along with its buckets and index nodes on optical media. We presented in detail the strategy for archiving of object versions, buckets of version pointers, and index nodes to WORM disk. We also described a technique for handling very long lived versions that may delay the migration process. Finally, we evaluated the performance of temporal access structures partitioned between magnetic and optical disks.

Acknowledgment

We would like to thank Ramana Guntor for his various contributions.

References


Figure 6: Storage for Monotonic $B^+$-tree Index Blocks


