Query Pre-Execution and Batching in Paradise: A Two-Pronged Approach to the Efficient Processing of Queries on Tape-Resident Raster Images[1]

JieBing Yu David J. DeWitt
Department of Computer Sciences
University of Wisconsin – Madison
{jiebing, dewitt@cs.wisc.edu}

Abstract

The focus of the Paradise project [1,2] is to design and implement a scalable database system capable of storing and processing massive data sets such as those produced by NASA's EOSDIS project. This paper describes extensions to Paradise to handle the execution of queries involving collections of satellite images stored on tertiary storage. Several modifications were made to Paradise in order to make the execution of such queries both transparent to the user and efficient. First, the Paradise storage engine (the SHORE storage manager) was extended to support tertiary storage using a log-structured organization for tape volumes. Second, the Paradise query processing engine was modified to incorporate a number of novel mechanisms including query pre-execution, object abstraction, cache-conscious tape scheduling, and query batching. A performance evaluation on a working prototype demonstrates that, together, these techniques can provide a dramatic improvement over more traditional approaches to the management of data stored on tape.

1. Introduction

In July 1997, NASA will begin to launch a series of 10 satellites as part of its *Mission to Planet Earth,* more popularly known as EOSDIS (for Earth Observing System, Data Information System). When fully deployed, these satellites will have an aggregate data rate of about 2 megabytes a second. While this rate is, in itself, not that impressive, it adds up to a couple of terabytes a day and 10 petabytes over the 10 year lifetime of the satellites [3]. Given today's mass storage technology, the data will almost certainly be stored on tape. The latest tape technology offers media that is both very dense and reliable, as well as drives with "reasonable" transfer rates. For example, Quantum's DLT-7000 drive has a transfer rate of approximately 5.0 MB/sec (compressed). The cartridges for this drive have a capacity of 70 GB (compressed), a shelf life of 10 years, and are rated for 500,000 passes [4]**.** However, since tertiary storage systems are much better suited for sequential access, their use as the primary medium for database storage is limited. Efficiently processing data on tape presents a number of challenges [5]. While the cost/capacity gap [5] between tapes and disks has narrowed, there is still a factor of 3.5 in density between the best commodity tape technology (35 GB uncompressed) and the best commodity disk technology (10 GB uncompressed) and a factor of 7 in total cost ($2,000 for a 10 GB disk and $14,000 for a 350 GB tape library). In addition, storage systems using removable media are easier to manage and are more expandable than disk based-systems for large volume data management.

There are two different approaches for handling tape-based data sets in database systems. The first is to use a *Hierarchical Storage Manager (HSM)* such as the one marketed by EMASS [6] to store large objects externally. Such systems almost always operate at the granularity of a file. That is, a whole file is the unit of migration from tertiary storage (i.e. tape) to secondary storage (disk) or memory. When such a system is used

[1] This work is supported by NASA under contracts #USRA-5555-17, #NAGW-3895, and #NAGW-4229, ARPA through ARPA Order number 018 monitored by the U.S. Army Research Laboratory under contract DAAB07-92-C-Q508, IBM, Intel, Sun Microsystems, Microsoft, and Legato.

to store satellite images, each image is typically stored as a separate file. Before an image can be processed, it must be transferred in its entirety from tape to disk or memory. While this approach will work well for certain applications, when only a portion of each image is needed, it wastes tape bandwidth and staging disk capacity transferring entire images.

An alternative to the use of an HSM is to incorporate tertiary storage directly into the database system. This approach is being pursued by the Postgres [8,9] and Paradise [1,2] projects, which extend tertiary storage beyond its normal role as an archive mechanism. With an integrated approach, the database query optimizer and execution engine can optimize accesses to tape so that complicated ad-hoc requests for data on tertiary storage can be served efficiently. In addition, with increasingly powerful object-relational features of systems such as Illustra (Postgres) and Paradise, complicated tasks like analyzing clipped portions of interest on a large number of satellite images can be performed as a single query [10].

In this paper, we describe the extensions that were made to Paradise [1,2] to handle query processing on image data sets stored on magnetic tape. Unfortunately, it is not just as simple as adding support for tape-based storage volumes. While modern tape technology such as the Quantum DLT (Digital Linear Tape) 7000 is dense and relatively fast, a typical tape seek still takes almost a minute! Our solution is two pronged. First, we employ a novel query execution paradigm that we term *query pre-execution.* The idea of pre-execution grew from the experimental observation[2] that queries which accessed data on tape were so slow that we could actually afford to execute the query twice! As we will describe in more detail in Section 4.2, during the pre-execution phase, Paradise executes the query normally except when a reference is made to a block of data residing on tape. When such a reference occurs, Paradise simply collects the reference without fetching the data and proceeds with the execution of the query. Once the entire query has been "pre-executed", Paradise has a very accurate reference string of the tape blocks that the query needs. Then, after using a *cache-conscious tape scheduling algorithm,* which reorders the tape references to minimize the number of seeks performed, the query is executed normally. While the idea of query pre-execution sounds impractical, we demonstrate that it actually works very effectively when dealing with large raster images on tape.

[2] Using the first version of the Paradise tertiary storage manager which did not employ query pre-execution.

Paradise also uses *query batching* to make query processing on tape efficient. Query batching is a variant of traditional tape-based batch processing from the 1970s and what Gray terms a *data pump* [11]. The idea of query batching is simple: dynamically collect a set of queries from users, group them into batches such that each batch uses the same set of tapes[3], pre-execute each query in the batch to obtain its reference string, merge the reference strings, and then execute the queries in the batch together (concurrently). The processing of a batch is done essentially in a "multiple instruction stream, single data stream" (MISD) mode. The ultimate goal is to scan each tape once sequentially, "pumping" tape blocks through the queries that constitute the batch as the blocks are read from tape.

To illustrate some of the issues associated with accessing a tertiary-resident data set in system like Paradise, consider an example in which we need to process a year's worth of weekly satellite imagery data. Figure 1 shows such a data set stored in Paradise: the entire data set appears to the user as a single relation with 'Week' and 'Channel' as integer attributes and Image as an attribute of type Raster ADT. The bodies of the images are stored sequentially on a tape volume in time order. The disk - resident portion of the Raster ADT contains metadata that includes OIDs linking the metadata with the actual images. Consider the following query: Find all image pairs from week 1 and week 26 which bear similarities over a specified region on each channel.

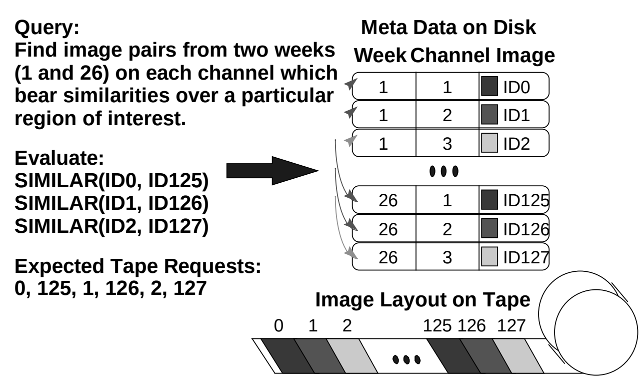

Figure 1: Motivating Example

Executing this query requires evaluating the *SIMILAR*() function on image pairs (i.e. channel 1 with channel 1, channel 2 with channel 2,...) from weeks 1 and 26. It is clear from Figure 1 that evaluating this function naively will cause excessive tape head movement between the two sets of images. To eliminate these random tape ac-

[3] We assume that there are enough tape readers to mount all the tapes needed by a batch simultaneously.

cesses, the relevant portions of all images from week 1 must be cached on disk before the first image from week 26 is accessed. While using techniques from executing pointer joins [12] or assembling complex objects [13] to reorder object accesses may help reduce the number of random accesses, in a multi-user environment, even if each query is executed using its best plan, the aggregate effect can still result in a large number of random tape accesses. The limited size of the disk cache can make matters even worse. It is not sufficient to rely solely on the query optimizer to generate optimal plans for tertiary query processing

The remainder of this paper is organized as follows. In Section 2, we summarize research related to the problem of adding tertiary storage support to database systems. The mechanisms used to extend Paradise to handle tertiary storage volumes are described in Section 3. Section 4 describes the design and implementation of query pre-execution and query batching inside Paradise. Section 5 contains a performance evaluation of these techniques. Our conclusions and future research directions are contained in Section 6.

2. Related Work

Tertiary Storage Management

The focus of the Highlight [14] and LTS [15] projects is the application of log-structured file system techniques [16] to the management of tertiary storage. Highlight integrates LFS with tertiary storage by allowing the automatic migration of LFS file segments (containing user data, index nodes, and directory files) between secondary and tertiary storage. The partial-file migration mechanisms in Highlight were the first attempt to provide an alternative to the whole-file migration techniques that have been widely employed by HSM (Hierarchical Storage Management) systems. Highlight's approach is closely integrated with LFS and treats tertiary storage primarily as a backing store. LTS has a more flexible design whose objective is to provide a general-purpose block-oriented tertiary storage manager. Extensions to Postgres to manage data on optical juke box are described in [8]. Our design for Paradise's tertiary storage volume manager borrows a number of techniques from LTS, but focuses on the use of tape devices instead of optical devices.

A multi-layered caching and migration architecture to manage persistent objects on tertiary storage is proposed in [17]. Their preliminary results demonstrate that sequential access to tape segments benefits from the multi-level caching while random accesses may cause excessive overhead.

Tape Scheduling

The very high access latency associated with magnetic tape devices has prompted a number of researchers to explore alternative ways of minimizing the number of random tape I/Os. [18] and [19] extend various disk I/O scheduling algorithms to the problem of tape I/O scheduling. [18] models access behaviors for helical scan tapes (e.g. 8mm tapes) and investigates both tape scheduling and cache replacement policies. Their results demonstrate that it is very important to consider the position of the tape head when attempting to obtain an optimal schedule for a batch of tape accesses. [19] models the behavior of accesses to serpentine tapes (e.g. DLT tapes), and compares different scheduling algorithms designed to optimize random I/Os on a DLT drive. Both studies show that careful scheduling of tape accesses can have a significant impact on performance.

Data Placement on Tapes

[20] and [21] investigate the optimal placement of data on tape in order to minimize random tape I/Os. These algorithms assume a known and fixed access pattern for the tertiary tape blocks. While very effective for applications that have fixed access patterns, they may not be as effective for general-purpose database systems in which ad-hoc queries can make predetermining access patterns essentially impossible. In addition, collecting the access patterns and reorganizing data on tapes over time may be a difficult task to accomplish in an on-line system.

Tertiary Storage Query Processing

[22] and [23, 24] propose special techniques to optimize the execution of single join operations for relations stored on tape. Careful selection of the processing block size and the ordering of block accesses is demonstrated to reduce execution time by about a factor of 10. [24] exploits the use of I/O parallelism between disk and tape devices during joins. [23] also identifies a number of system factors that have a direct impact on query processing with a focus on single relational operations.

User-Managed Tertiary Storage

The first attempt to integrate tertiary storage into a database system appeared in [25]. A three-level storage hierarchy was proposed to be under the direct control of a database management system with tertiary storage at the bottom layer. Data could be "elevated" from tertiary storage to secondary storage via user-level commands. Another user-level approach is described in [26], in which the concept of a user-defined *abstract* is proposed to reduce the number of accesses that have to be made to tertiary storage. The idea is that by carefully abstracting the important contents of the data (aggregate and summary information) to form an abstract that is stored on disk, the majority of queries can be satisfied using only the abstracts.

Integrated Approach

The most comprehensive system-level approach for integrating tertiary storage into a general database management system is proposed in [9]. A novel technique of breaking relations on tertiary storage into smaller segments (which are the units of migration from tertiary to secondary storage) is used to allow the migration of these segments to be scheduled optimally. A query involving relations on tertiary storage is decomposed into multiple mini-queries that operate in terms of segments. These mini-queries are then scheduled at run-time according to the availability of the involved segments on disk and memory. A set of priority-based algorithms are used to fetch the desired segments from tertiary storage on demand and to replace segments on the cache disk.

Follow-up work in [27] details a framework for dynamically reordering query execution by modifying query plans based on the availability of data segments. The difference between this approach and ours is that our emphasis is on optimizing tape accesses at the bottom layer of the execution engine, leaving the original query plan unchanged. Not only is this strategy simpler but also it provides more opportunities for optimizing executions under multiuser environment. However, it appears fruitful to consider combining the two approaches using query pre-execution as mechanism to “resolve” [27] accesses to satellite images and using “schedule nodes” [27] in our query plans to handle data dependencies between operators in the query tree.

3. System Architecture

Paradise is an object-relational database system whose primary focus is on the efficient management and processing of large, spatial and multimedia data sets. The structure of the Paradise server process is shown in Figure 2. The SHORE storage manager [28] is used as the underlying persistent object manager. Support for tertiary storage in Paradise began by the extending SHORE. These extensions are described in the following section.

3.1 SHORE Storage Manager Extensions for Tertiary Storage

The SHORE storage manager is a persistent object manager with built-in support for multi-threading, concurrency control, recovery, indexes and transactions. It is structured as a set of logical modules (implemented as C++ classes). Access to permanent data involves four modules: a disk read/write process[4], the buffer manager, the I/O manager, and a disk volume manager. To the basic SHORE storage manager, we added the following components: a block-oriented tape I/O driver, a tertiary storage volume manager, a disk-cache buffer manager, and a cache volume manager. Together with minor modifications in other higher layer modules, the addition of these components enables the SHORE SM to directly access volumes on tertiary storage. The details of these components are described below.

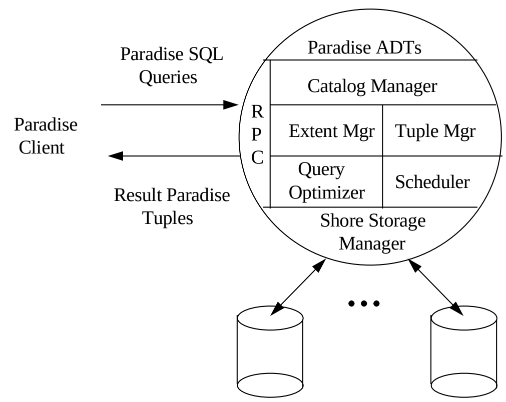

Figure 2: Paradise Process Architecture

Block-Oriented Tape I/O Driver

As the low-level physical driver for accessing data on tape volumes, this module adds a block-oriented access interface on top of the standard UNIX tape I/O routines. The driver formats a tape into a set of fixed-sized tape blocks. As a request for a particular physical tape block arrives, the driver directs the tape head to the corresponding physical address, and performs the read/write operation in a block-oriented fashion. The driver is implemented as a C++ class with tape head state information kept in its instance variables. In addition, a set of service utilities for loading and maintaining tape metadata information is provided to facilitate tape mounts and dismounts. This metadata includes information on the tape format, tape block size, current tape end block number, and tape label. The use of standard UNIX tape I/O routines allows the driver to be independent of underlying tertiary storage device and platform.

[4] The disk read/write process is used to obtain asynchronous I/O in those OS environments that lack a non-blocking I/O mechanism.

Tertiary Storage Volume Manager

The *tertiary storage volume manager* is responsible for space management on tape volumes. It has all the functionality of the normal SHORE disk volume manager for allocating and de-allocating both pages and extents of pages. In addition, it is responsible for mapping individual pages to their containing tape blocks, and keeping track of the mapping between logical and physical tape block addresses. The basic unit of access inside the SHORE storage manager is a page. To simplify implementation, the tertiary storage volume manager was designed to provide exactly the same interface as the regular disk volume manager. This has the advantage of making access to tertiary data totally transparent to the higher layers of SHORE.

While preserving the same interface was critical, it is not possible to use the same block size for both disk and tape since the two media have very different performance characteristics. In particular, seek operations on tape are almost four orders of magnitude slower than seeks on disk. Thus, a much larger block size is required [6]. Our implementation makes it possible to configure the tape block size when the tape volume is being formatted. In a separate study [29], we examine the effect of different tape block sizes for a variety of operations on raster satellite images stored on a Quantum DLT 4000 tape drive. For this set of tests, we determined that the optimal tape block size was between 64 and 256 Kbytes. Since tapes are (unfortunately) an "append-only" media, a *log-structured organization* [16] is used to handle updates to tape blocks with dirty tape blocks being appended at the current tail of the tape. A mapping table is used to maintain the correspondence between logical and physical tape blocks.

The SHORE storage manager organizes disk volumes physically in terms of *extents*, which are basic units of space allocation/de-allocation. An extent is a set of contiguous pages. Logically, the disk volume is organized in terms of *stores*, which are the logical units of storage (like a file in UNIX file system). Each store may consist several extents. Figure 3 depicts the regular SHORE disk volume organization. Each rectangle on the left denotes a page and tiles inside a page are slotted entries. As can be seen from the figure, a set of pages in the beginning of the volume are reserved for the metadata storage, which includes a *volume header*, a slotted array for the *extent map*, and another slotted array for the *store map*. The *extent map* maintains the page allocation within each extent, and extents belonging to a single store are maintained as a linked list of extents with the head of the list stored in the *store map*. Figure 4 illustrates the extensions that were made to support SHORE volumes on tertiary-storage. The only changes are the extended volume header to cover tape-related meta information and the addition of a *tape block mapping table*. This design allowed us to implement the tertiary storage volume manager as a C++ class derived from the disk volume manager with a significant amount of code reuse. In addition, storing all the needed tape volume information in its header blocks makes the tape volume completely self-descriptive. The header blocks are actually cached after mounting a tape volume.

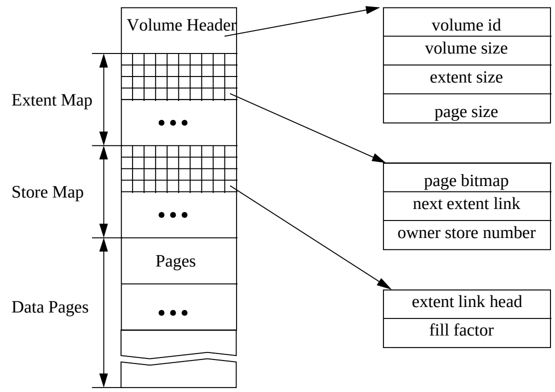

Figure 3: Disk Volume Organization

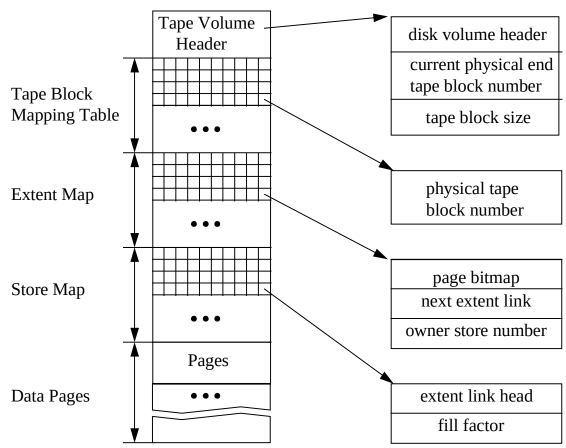

Figure 4: Tape Volume Organization

Disk Cache Manager

After being read, tape blocks are cached on secondary storage for subsequent reuse. This disk cache is managed by the *disk cache manager*. The tertiary storage volume manager consults the disk cache manager for information on cached tape blocks, acquiring cache block space as necessary. The disk cache manager uses the same resource manager utilized by the in-memory buffer manager for cache management, except that the unit of management is a tape block instead of a page. Each cached entry in the tape block mapping table contains a logical tape block address plus the physical address of its first page in the disk cache. With this information, the address for any cached page can be easily calculated. In addition, a dirty bit is used to record whether the block has been updated. While the resource manager could incorporate various kinds of cache-replacement policies, LRU is used for its simplicity.

Cache Volume Manager

The *cache volume manager* is a simplified version of the regular SHORE disk volume manager. It takes care of mounting and dismounting disk cache volumes and provides routines for reading and writing both pages and tape blocks and for transferring tape blocks between the cache volume and tape.[5]

3.2 Examples of Tertiary Storage Accesses

Figure 5 illustrates the operation of SHORE when a page miss occurs in the main memory buffer pool. There are four processes present in the figure: a SHORE SM server process, a *disk read/write (rw)* process for a regular disk volume, a second *disk rw* process for the cache volume, and a *tape rw* process for the tape volume. A shared-memory region is used for both the normal buffer pool and as a buffer for tape blocks being transferred between tape and the cache volume. The shaded components represent either new components or ones that were modified to permit access to tape data. To illustrate how each type of access is performed, we next walk through several different types of accesses and explain the actions involved using Figure 5.

Disk Volume Access

Access to pages from a normal disk volume involves steps 1, 2, 3 and 4. A page miss in the main memory buffer pool results in the following series of actions. First, the buffer manager selects a buffer pool frame for the incoming page and identifies the appropriate volume manager by examining the volumeId component of the pageId. Next, the buffer manager invokes a method on that volume manager to fetch the page (step 1). The disk volume manager translates the page number in the pageId into a physical address on the disk device and passes it along to its corresponding I/O manager (step 2). The I/O manager in turn sends[6] a read request to the associated *disk rw* process (step 3). The request contains both the physical address of the page on disk and the buffer pool frame to use. The disk driver schedules the read and moves the page directly to its place in buffer pool (step 4). Page writes follow a similar sequence of steps.

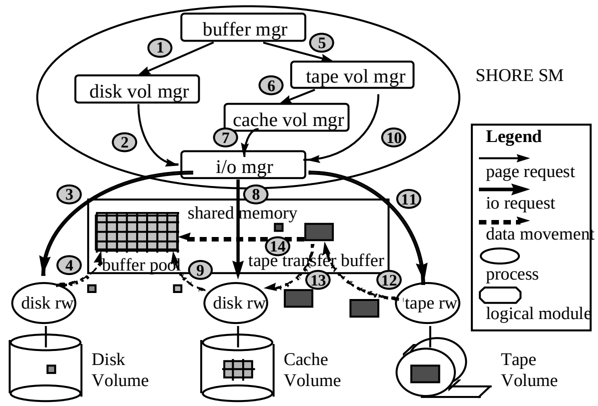

Figure 5: Tertiary Storage Access Structure

Tape Volume Access

Access to pages of tape blocks is more complicated because the desired page may reside either in the cache volume or on tape. First, the buffer manager sends a request to the tape volume manager (step 5). This is the same as step 1 except that the tape volume manager is identified from the volumeId component of the pageId. After receiving the request, the tape volume manager first asks the cache volume manager whether a copy of the desired page is in the cache volume. This is done for both performance and correctness reasons as the cache will have the most up-to-date version of the tape blocks.

If the cache volume manager finds an entry for the tape block that contains the desired page, then steps 6, 7, 8, 9 are performed to fetch the page into buffer pool. First, the tape volume manager translates the requested page address into a page address in the cache volume. The mapped address is then passed to the cache volume manager which is responsible for reading the page. The remaining steps, 7, 8, and 9, are the same as steps 2, 3, and 4.

If the containing tape block is not found by the disk cache manager, it must be read from tertiary storage into the cache volume. The tape volume manager first looks at the *tape block mapping table* to translate the logical block number into a physical block number. Then, through step 10, it calls the corresponding I/O module to schedule the migration. The I/O manager sends a mi-

[5] Via memory as one cannot move blocks of data between two SCSI devices without passing through memory.

[6] Actually, a queue is maintained in shared-memory for the volume manager to communicate I/O requests to the appropriate disk rw or tape rw process.

[7] Actually, a queue is maintained in shared-memory for the volume manager to communicate I/O requests to the appropriate disk rw or tape rw process.

gration request containing the physical tape block number and which tape transfer buffer to use (step 11). The block-oriented tape driver then processes the read request, placing the tape block directly into the specified tape transfer buffer (step 12). At this point, control is returned to the tape volume manager, which invokes the cache volume manager to transfer the tape block from shared memory to the cache volume (step 13). Finally, instead of going through the normal channels (steps 6, 7, 8, 9) to finish bringing the desired page into buffer pool, we use a short cut to copy the page directly out of the tape transfer buffer into the buffer pool (step 14).

4. Query Processing Extensions

From the previous section, it is clear that our tertiary storage implementation places a strong emphasis on minimizing the number of changes to the upper layers of the SHORE Storage Manager. By carefully placing the changes at the bottom layer of the storage structure, very few changes in the upper layers of the SHORE SM had to be modified, enabling us to preserve higher level functions like concurrency control, recovery, transaction management, and indexing for data resident on tertiary storage. Consequently, only minimal changes were needed to extend Paradise to manage data stored on tertiary storage.

However, merely storing and accessing data transparently on tape is not sufficient to insure the efficient execution of queries against tape-resident data sets. In particular, while database algorithms always strive to minimize the number random disk seeks performed, there is only a factor of 4 to 5 difference in the cost of accessing a page on disk randomly versus sequentially. Tapes are another story. With a seek on a modern DLT tape drive taking almost a minute, there are literally 4 orders of magnitude difference between accessing a tape block randomly and sequentially. In short, seeks must be avoided to the maximum extent possible. In this section we describe four new mechanisms which, when used together, help minimize tape seeks and maximize performance of queries involving spatial images stored on tertiary storage.

4.1 System-Level Object Abstraction

Given database support for tertiary storage, the first question one needs to ask is what data should be stored on tape and what data should be stored on disk. Clearly, frequently accessed data structures like indices and system metadata are better off stored on disk, but what about user data? In the context of projects like EOSDIS, it is clear tapes should be used to hold large satellite images (typically between 10 and 100 megabytes in size) while their associated metadata (typically a couple 100 bytes) should be stored on disk. Separating the metadata from the actual image will help to reduce accesses to tertiary storage for certain types of queries. For example, the metadata for a typical satellite image will contain information such as the date that the image was taken, its geo-location, and some information about the instrument and sensor that took the image. Predicates involving date or location can be processed by only accessing the metadata, without fetching unnecessary images.

Assuming that images are to be stored on tape, how should the image itself be represented in the image's metadata? A naive approach would be to store the OID of the object containing the tape-resident image as part of the disk-resident metadata. This approach is fine if images are always accessed in their entirety. However, processing of only pieces of images is fairly common [10]. As a solution, Paradise uses *tiling* [1, 2] to partition each image into multiple tiles, with each tile stored as a separate object on tape. Thus, only those tiles that are actually touched by a query need to be read from tape.

This approach requires that the OIDs for the tiles be stored as part of the image's metadata. We term the set of OIDs corresponding to the tape-resident tiles a *system-level object abstraction*. This differs from the user-level abstraction proposed by [26] in that the tiling process is handled automatically by Paradise. Figure 6 illustrates one such representation for a raster image. In this example, the body of the image is partitioned into 4 tiles stored on tape, while its metadata containing the tile OIDs are stored on disk. The collection of tile OIDs act as an object abstraction for the image data.

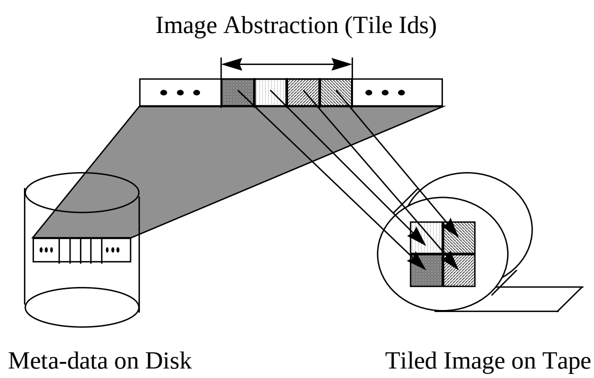

Figure 6: Raster Image Abstraction

Since Paradise uses an abstract data type (ADT) mechanism for implementing all its types, the system-level object abstraction was incorporated into the ADT that is used for satellite images. Since all methods operating on the image must pass through the abstracted object representation first, the addition of this abstraction is totally transparent to upper levels of the system. In addition, modifications and improvements are totally isolated in the corresponding ADT code. As will be described later in 4.2, this representation makes it possible to optimize tertiary storage accesses by generating reference strings to objects on tertiary storage without performing any tape I/Os.

4.2 Query Pre-execution

Accurately estimating access patterns for guiding run-time resource management and scheduling has been the goal of many projects. An accurate access pattern estimation is important for optimizing page accesses since all scheduling algorithms (disk or tape based) require a queue of requests to operate on. However, only a small number of applications have a known, fixed access pattern and, hence, can actually benefit from such disk/tape scheduling mechanisms. As part of our effort to optimize tape accesses, we developed a technique that we term *query pre-execution* which can be used to accurately generate reference strings for ad-hoc queries involving accesses to tape-resident data sets. The core idea is to execute each query twice: the first phase executes the query using the system-level object abstraction described in Section 4.1 to produce a string of tape references without performing any actual tape I/Os (access to disk-resident data proceeds as normal - except obviously for updates). After the query pre-execution phase has been completed, the string of tape block references collected during this phase are reordered and fed to the tape scheduler (Section 4.3 describes the reordering process). Finally, the query is executed a second time using the reordered reference string to minimize the number of tape seeks performed. While this idea sounds impractical, we will demonstrate in Section 5 that it works extremely well for tape-resident sets of satellite images. In the general case, a mechanism such as proposed in [27] for inserting "schedule nodes" in the query plan will be needed to resolve data dependencies between operators in the query tree.

In order to support the query pre-execution phase, special mechanisms were added to Paradise's query execution engine to monitor the processing of the system-level object abstractions. During the course of pre-execution phase, if an ADT function is invoked on a tuple for operations on the object abstraction of a large object that resides on tertiary storage, any tape-bound requests that might occur in the method are recorded in a data structure instead of actually being executed. The function returns with an indication that its result is incomplete, and the query processing engine proceeds to work on the next tuple. The end result of the pre-execution phase is a sequence of tape block references in the exact reference order that would have occurred had the query been executed in a normal manner.

- Schema

    Table rasters(time *int*, freq *int*, image *Raster*)

    Table polygons(landuse *int*, shape *Polygon*)

- Query

    Select rasters.image.clip(polygons.shape)
    from rasters, polygons
    where rasters.time = 1 and rasters.freq = 5
    and polygons.landuse = 91

Figure 7: Sample Query

Figure 7 illustrates a query involving a "join" between a set of polygons and a set of raster images. The "join" is implicitly specified via the *clip* operation on the image attribute. Each tuple in the "rasters" table contains three fields: *time* and *freq* as integers and *image* as an instance of the raster ADT. Tuples in the "polygons" table have fields *landuse* of type integer and *shape* of type polygon. By using the system-level object abstraction, the image attribute of each tuple in the rasters relation contains only abstractions (tile ids and their corresponding image partition information). The query specified is intended to select the raster images with the desired *time* and *freq* values (1 and 5) and clip them with all polygon shapes whose *landuse* value equals 91. The *clip* operation is a function defined on *raster ADT* for subsetting the image into the desired bounding rectangle region covered by the polygon shape.

The top part of Figure 8 shows the spatial layout of an example for such a query. In the figure, the selected raster image is tiled into 4 parts, and there are two polygons of interest to be processed. The middle part shows how the clip operation is accomplished for the query. The two polygons are processed in their original order of storage on disk. The result is four rectangular clipped portions of the raster image. During the pre-execution of this query, the clip function is modified to record only the tile ids for covered tiles instead of fetching the tiles from tape and producing the clipped result. At the end of the pre-execution, we have a collection of tile ids in the exact order that they must be read from tertiary storage. These tile ids are the physical OIDs of the associated tape-resident tiles and provide a very accurate prediction on which tape blocks will actually be accessed when the query is executed the second time. This is illustrated in the bottom part of Figure 8. Notice that the raster image is replaced by its abstraction and the result is a series of tile ids instead of the final, clipped portions of the image in a random order.

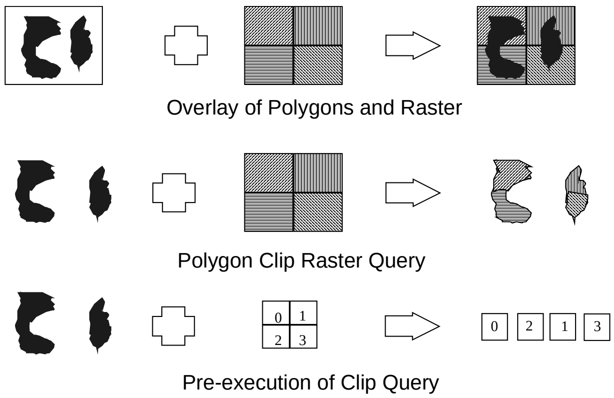

Figure 8: Pre-Execution Example

4.3 Cache-Conscious Tape Scheduling

The reference string of tape-block accesses generated during query pre-execution can be used to optimize tape accesses. Given a set of references, the problem of optimal tape scheduling seems to be straight forward. The sequential access nature of tape provides few alternatives other than to sort the requests and to make one sequential pass over the tape to process all the requests at once. However, this seemingly straightforward approach has a big drawback: it ignores the fact that the tape requests must be returned in their original order in order to execute the query. Tape-blocks in a different order must be cached long enough on primary or secondary storage to be referenced by the executing query or the access will have been wasted. This actually puts a constraint on the optimal schedule such that the distance between the original request and the reordered request cannot exceed the size of the disk cache used to buffer tape blocks as they are being read from tape. Otherwise, some of the pre-fetched tape blocks will be prematurely ejected from the cache in order to make room for more recently read blocks that have not yet been used. Ejecting such blocks not only wastes work but also adds additional random tape seeks.

To cope with this problem, one must factor the cache size (in terms of the number of tape blocks) into the process of finding an optimal schedule. The scheduling problem now becomes: given a bounded buffer and a set of requests, find the optimal scheduling of these requests such that the number of random tape accesses is minimized. The added constraint of the bounded buffer makes the problem *NP-Hard.* While exponential algorithms can be used to find the globally optimal solution, this approach is too expensive in terms of time and memory consumption for long streams of requests and for large cache sizes. A straightforward solution is *bounded sort*: break the entire stream into multiple cache-sized chunks and sort the requests in each chunk. This approach may, however, miss some opportunities for further improvement. We developed a simple heuristic-based, one-pass algorithm to find a reasonably good *cache-conscious tape schedule*. The idea of the algorithm is to reorder the original reference stream so that the new stream consists of a number of chunks having the following properties: 1) the tape block references in each chunk are sorted according to their location on tape, and 2) all the tape blocks in each chunk can be read in order without overflowing the disk cache. In addition, a sliding window is used to smooth out the boundary effect that could arise from the *bounded sort* step.

The algorithm works by moving across the original reference stream from left to right and, in a single pass, constructing a new optimized reference stream. At each step, it looks at a sliding window of references containing as many block references as would fit on the disk cache[8]. Now, if the first block reference in the sliding window happens to be the lowest request in the whole window, then this reference is added to the optimized reference stream, and the sliding window is moved forward by one position. If the first block reference is not the lowest reference in the window, then all the references in the window are sorted, and the whole chunk is added to the optimized reference string. Then the sliding window is moved past this whole chunk. This process is repeated until the whole input reference stream is processed.

Figure 9 illustrates a sample run of the algorithm. We assume that the disk cache can hold three tape blocks. Initially, the input stream contains the reference string 7,2,1,3,4,8,6,5,8. The algorithm starts by considering the first three references 7,2,1 (Step 1). Since 7 is not the lowest reference in this window, the whole chunk is reordered (Step 2). This chunk is added to the optimized schedule, and the sliding window is moved past this block to cover 3,4,8 (Step 3). At this stage, since 3 is the lowest reference in this window, it is moved out of the window immediately (Step 4). Now the window covers 4,8,6. Again, since 4 is the lowest reference in the stream, it is shifted out immediately. The sliding window now covers the string 8,6,5,8 (Step 5). We note that al-

[8] Since the same block might be referenced multiple times in a reference stream, the sliding window might actually contain more references than the number of blocks that fit in the disk cache, but the number of distinct references must be the same.

though the sliding window covers four tape block references, there are only three distinct references in the window. These references are reordered (Step 6) and the whole chunk is moved into the optimized stream (Step 7). The use of multiple chunks for each query allows chunks from concurrent queries to be easily merged together by interleaving accesses in a multi-user environment. The reordering function can be more than just an address sorting function. For example, more sophisticated functions like the one proposed in [19] can be used to deal with tapes with non-linear access characteristics (e.g. DLT tapes).

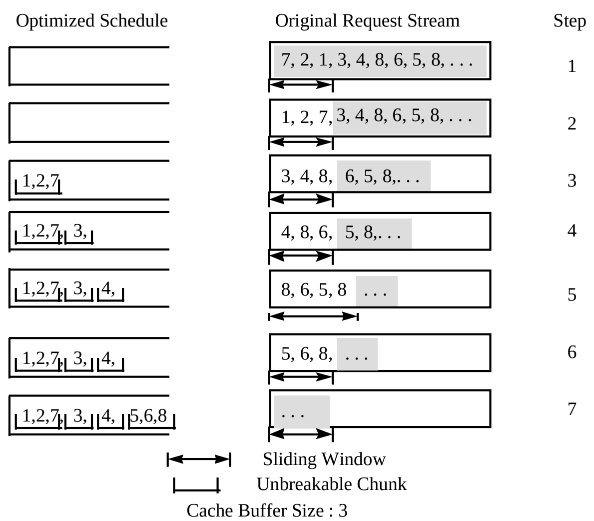

Figure 9: One-Pass Algorithm for Finding Near-Optimal Cache-Conscious Schedule

4.4 Query Batching

The final schedule produced by the cache-conscious scheduling algorithm is used to direct query execution at run time. When a query issues a request for a tape block that cannot be satisfied by the disk cache volume, the tape volume manager first examines the query's schedule to identify the block's position in the schedule. If the requested block is preceded by other blocks, reads are issued for all the blocks up to and include the requested block in the order specified by the schedule. While the execution of a query can be temporarily blocked due to the unavailability of data, when the requested tape block is finally brought into the cache, execution resumes. Since scheduling is done in terms of cache-sized units, we are assured that blocks that are read prior to their actual use will not result in the eviction of blocks that will be referenced in the near future.

This execution model also provides the opportunity for improving the performance of concurrently executing queries as intermixing queries in such an environment makes the problem of random tape seeks even worse. However, simply running optimally scheduled individual queries concurrently does not work because when their tape reference patterns are interleaved, the resulting reference pattern will, in general, contain a large number of random seeks. As a solution, we used *query batching* to batch concurrent queries together, combining individually optimized schedules to produce a globally optimized schedule, which is then used to drive the concurrent execution of the queries in the batch. The merge algorithm is also heuristic based, with the objective of eliminating random accesses among individually optimized schedules for all queries in the current batch. It combines chunks from different schedules using a "best-fit" criteria. In this scheme, query execution is indirectly controlled by the tape scheduler. With a multi-threaded server, the actual execution of the queries in a batch can be interleaved if their accesses to tertiary data in the merged global schedule are also interleaved. The merge process takes place dynamically by monitoring each new query that arrives to determine if it can be added to the currently executing batch. The criteria currently being deployed is to check whether the new query's request stream is behind the head of the current schedule. If so, it is blocked to be merged with next round of queries, otherwise it is admitted immediately to the current batch. As we will demonstrate in Section 5, this technique is extremely effective in reducing random tape accesses and improving system throughput in a multi-user environment.

Since the algorithm described in Section 4.3 assumes the exclusive control of the disk cache (by using the maximum buffer size as the constraint on the size of an unbreakable chunk), scheduling conflicts among concurrent queries can cause significant performance problems in a multiple user environment. To deal with this problem, we reduce the buffer size used to schedule the individual queries in order to reduce potential conflicts. The choice of this buffer size depends on the multiprogramming level and the access pattern to tape-resident data. Individual tape schedules are then merged at run time to produce an optimized plan.

5. Performance Evaluation

To evaluate the efficiency of the system in a realistic setting, we developed a *tertiary storage benchmark* to determine the effectiveness of our techniques. The benchmark, the experimental configuration, and the results of our tests are presented in the following sections.

5.1 A Tertiary Storage Benchmark

While designing an effective benchmark is always a challenge, doing one for a tertiary storage environment is even harder. First, the size of the database must be large enough to really test the limits of the system. This, in turn, makes the testing process very slow and tedious. Second, the benchmark must incorporate a variety of access patterns in order to be realistic. In order to study the effectiveness of the techniques described above, we developed a scaled-down benchmark that we call the Tertiary Storage Mini-Benchmark.

This benchmark uses regional data from the Sequoia 2000 benchmark [10]. In order to be able to compare the relative performance of the non-optimized strategies with our approach, we use a base benchmark consisting of 60,000 land use polygons and 130 8MB geo-located raster images corresponding to 5 raster images per week for 26 weeks (for a total database size of approximately 1 GB). The test set is then scaled in two different ways. First, we keep the number of images constant at 130 while increasing the size of each image to 32 MB. Next, we keep the image sizes constant at 8 MB and scale the number of images from 130 to 1300. This gives us a test database size of slightly over 10 GB. We limit the disk cache size to 50 MB in order to simulate a more realistic environment, which would have tape resident data sets in the 100s of GBs or more. The schema for the database is the same as the one used in Figure 7.

In Figure 10 a suite of 11 queries are defined to test a variety of query patterns. These queries are based on raster-related queries from the Sequoia 2000 benchmark that we modified to produce a wider variety of access patterns. *Q1*, *Q2* and *Q3* involve 89 small polygons clipping 26, 5, and 1 raster images respectively. Queries *Q4, Q5* and *Q6* repeat *Q1, Q2* and *Q3* at a reduced scale - using only 12 small polygons. These two sets of queries are designed to mimic the scenario that a user is interested in a number of different regions from a set of different images. *Q7*, *Q8* and *Q9* use a fixed 1% region to clip 26, 5, and 1 raster images respectively. Finally, queries *Q10* and *Q11* are designed to reflect more advanced queries that compare images taken from the same instrument over different period of times. Q10 compares fixed regions of images from two adjacent weeks on each of the five frequencies, Q11 repeats the same process on images from two weeks that are 25 weeks apart. Since the raster images are stored on tape in their chronological order, this suite of queries represents an interesting combination of many different access patterns.

5.2 System Configuration

A 200MHz Pentium Pro with 64 MB of memory running SOLARIS 2.5 was used to run the Paradise Server. Queries were submitted by processes running on a different machine. Quantum Fireball 1 GB disks were used for the log, the regular disk volume, and the cache disk volume. For tertiary storage, we used a DLT-4000 tape drive. This drive has capacity of 20 GB (uncompressed). For the benchmark, we used a 40 GB tape volume, a 50 MB cache disk volume, and a 500 MB regular disk volume.

- Q1: Polygons with *landuse 72* clip Rasters on *frequency 5* over all weeks.
- Q2: Polygons with *landuse 72* clip Rasters in *week 1*.
- Q3: Polygons with *landuse 72* clip Raster on *frequency 5* from *week 1*.
- Q4: Polygons with *landuse 91* clip Rasters on *frequency 5* over all weeks.
- Q5: Polygons with *landuse 91* clip Rasters in *week 1*.
- Q6: Polygons with *landuse 91* clip Raster on *frequency 5* from *week 1*.
- Q7: Fixed region (1%) clip Rasters on *frequency 5* over all weeks.
- Q8: Fixed region (1%) clip Rasters in *week 1*.
- Q9: Fixed region (1%) clip Raster on *frequency 5* from *week 1*.
- Q10: Compare Rasters from *week 1* with Rasters from *week 2* on each frequency.
- Q11: Compare Rasters from *week 1* with Rasters from *week 26* on each frequency.

Figure 10: Tertiary Benchmark Queries

5.3 Single-user Query Execution

For the first experiment, we tested three different configurations: "Base," "Whole," and "Pre-Exec." While the "Base" configuration provides access to tertiary data, it does not use any of the techniques described in Section 4. In effect, it corresponds to the approach in which the database storage engine treats tertiary storage as just another layer in the storage hierarchy. The "Whole" configuration is intended to simulate the approach of using an external tertiary storage manager (such as EMASS [7]) to migrate tertiary data on a per-file basis. In the "Whole" configuration each raster image is stored as a separate file on tape. When access to any part of the image is detected at run-time, the processing engine issues a request to migrate the entire file from tape to the disk cache.[9] The "Pre-Exec" configuration is our fully

[9] Because an exact implementation of the "Whole" architecture was not available, we simulated this approach by making the tape block size the same as the image size (either 8MB or 32MB, depending on the experiment). Whenever, any part of the image is accessed, the tape manager will fetch the entire image from tape.

integrated approach. For the “Base” and “Pre-Exec” configuration, the tape block size was set to 128KB. For the “Whole” configuration, a tape block size of 8 MB is used to match the size of the raster image. The main memory buffer cache and disk cache were flushed between queries to insure that all queries were executed with cold buffer pools.

	Base	Whole	Pre-Exec
Query 1	5652.1	908.8	282.5
Query 2	484.2	92.6	12.9
Query 3	141.4	25.3	11.7
Query 4	3125.4	800.4	280.7
Query 5	245.2	66.7	12.7
Query 6	89.8	23.8	11.6
Query 7	280.2	801.4	280.2
Query 8	12.4	68.2	12.4
Query 9	11.3	23.9	11.4
Query 10	165.9	246.8	23.3
Query 11	1187.1	1265.1	156.7

Table 1: Single Query Response Times (Seconds)

Table 1 contains the response times for the eleven benchmark queries for each of the three configurations. Figure 11 presents the performance of the “Pre-Exec” configuration relative to the base “Base” and “Whole” configurations. The performance of the “Pre-Exec” configuration is clearly superior for almost all queries. The “Pre-Exec” configuration provides a speedup factor of between 6 and 20 over “Base”, and a factor of between 4 and 10 over “Whole” for most queries. These results demonstrate that query pre-execution, cache-conscious tape scheduling, and run-time data driven query scheduling are very effective in reducing random tape I/Os when processing ad-hoc queries.

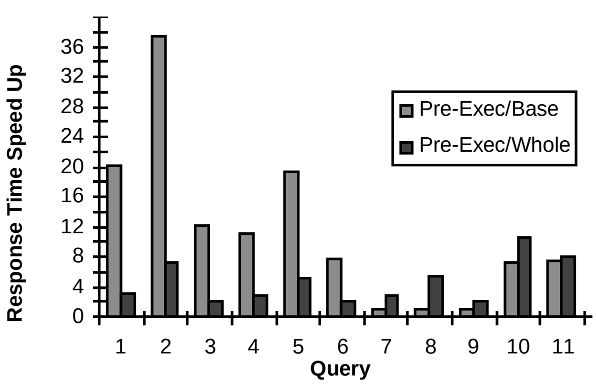

Figure 11: Speedup of Pre-Exec over Base and Whole

The significant improvement exhibited by the “Pre-Exec” configuration for the first 6 queries is due to the random spatial distribution of the polygons selected by the query. For queries Q7, Q8, and Q9, the clip region is a fixed area. This eliminates the randomness within each raster image among different clips, leaving basically no room for improvement. Consequently the “Pre-Exec” configuration is slightly slower than the “Base” configuration as a consequence of the overhead associated with query pre-execution. These three results illustrate the amount of overhead (no worse than 1%) that may incur if the tape access pattern of the original query (before reordering) is already in its optimal order. The final two queries (Q10 and Q11) benefit significantly from query pre-execution for their inherent nature of random tape access. Without query pre-execution and tape scheduling, these two queries would result in excessive amount of tape head movement between images from the two weeks.

One interesting observation from this set of experiments is that the “Whole” configuration is consistently better than the “Base” configuration for the first 6 queries. The reason is that these queries exhibit a high degree of randomness in the processing of each image. This randomness can be reduced by simply transferring the entire image from tape to disk as a single I/O. While this strategy works well for queries Q1 - Q6, for queries Q7 - Q9, which only need to access 1% of the image, the relative performance of the two configurations is reversed. In this case, unnecessary data migration costs the “Whole” configuration some extra work, and the benefit of reducing random I/O within each image cannot be capitalized due to the original serial tape access pattern in these queries. Since the random tape access patterns of queries Q10 and Q11 arise from moving the tape head back and forth between the two set of images, the “Base” and “Whole” configurations have similar performance except that the Whole configuration migrates more data. These tests demonstrate that the “Pre-Exec” configuration not only reduces random tape I/Os, but also, manages to transfer the correct amount of data for each type of query.

Query pre-execution obviously incurs some overhead during the first phase. To illustrate the extent of this overhead, the response times for each of the 11 queries are broken down in Table 2. “Phase 1” represents the percentage of the total time required to actually pre-execute the query. Given the reference string collected during phase 1, “Schedule” is the percentage of total time spent finding the optimal tape access schedule. “Phase 2” corresponds to the final, complete execution of the query. For all queries, there is less than a 5%

overhead for pre-execution and scheduling combined.

	Phase 1	Schedule	Phase 2
Query 1	0.37%	0.32%	99.31%
Query 2	2.85%	2.08%	95.07%
Query 3	0.08%	1.27%	98.65%
Query 4	0.11%	0.09%	99.80%
Query 5	0.93%	0.41%	98.66%
Query 6	0.51%	0.49%	98.99%
Query 7	0.07%	0.01%	99.92%
Query 8	0.40%	1.19%	98.41%
Query 9	0.26%	0.17%	99.57%
Query 10	2.92%	0.09%	96.99%
Query 11	0.17%	0.01%	99.82%

Table 2: Breakdown of Response Time for Pre-Exec (% of Total Time)

Next we scaled the test database in two different ways. First, we increased the size of each image from 8MB to 32MB (what we term *resolution scaleup*). Second we increased the number of images from 130 to 1300 (*cardinality scaleup)* while keeping the image size constant at 8MB. Then the same set of experiments were repeated (all queries use the same parameter). As the results in Table 3 indicate, the "Pre-Exec" configuration continues to have the same performance advantage relative to the "Base" configurations.

Query	Base (32x130)	Pre-Exec (32x130)	Base (8x1300)	Pre-Exec (8x1300)
Q1	15,413	1,122	56,154	2,963
Q2	1,070	52	494	13
Q3	304	43	141	12
Q4	7,635	1,084	29,742	2,892
Q5	522	45	247	13
Q6	145	43	90	12
Q7	1,062	1,066	2,884	2,888
Q8	42	42	12	12
Q9	41	40	11	12
Q10	501	89	168	23
Q11	1,240	185	1,148	162

Table 3: Scale Up - 32MB x 130 Images and 8 MB x 1300 Images (Seconds)

5.4 Multi-user Query Execution

We next turn our attention to the execution of concurrent queries. While the need for concurrency may be reduced in an environment with tape-resident data, executing multiple queries as a batch actually provides an excellent opportunity to amortize the cost of scanning a tape, which, as we will demonstrate below, can result in significant performance gains.

Four alternative configurations were considered: "Pre-Exec + Batch"—pre-execution with *query batching*, "Pre-Exec + Serial"—pre-execution with serialized execution (i.e. one query at a time), "Pre-Exec"—pre-execution without batching (multiple queries running at the same time with each following their optimized schedule), and "Base"—regular concurrent execution without either pre-execution or batching. Four client processes were started simultaneously. Each client picked a query randomly from the set of queries (Q1 to Q11) and submitted it to the Paradise server for execution. This process was repeated three times giving a total of 12 queries. While we would have preferred to run more queries, the base configuration for this limited set of tests consumed over 8 hours of test time. The results are presented in Figure 12. The metric used is the total elapsed time to finish all three queries from all four clients.

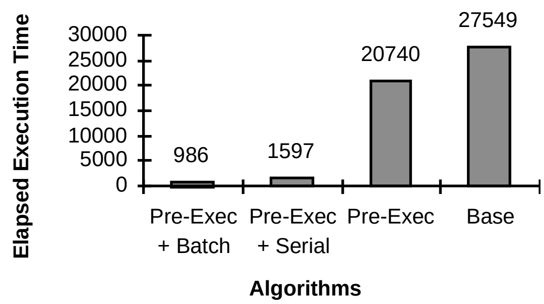

Figure 12: Multi-user Query Execution

It is obvious from these results that the combination of query pre-execution, tape scheduling, and run-time query scheduling via query batching is much better than other strategies. (There is almost a factor of 30 in improvement.) The reason is fairly obvious: the concurrent execution of multiple queries (even if they are individually executed according to their optimized schedules - the "Pre-Exec" case) results in a large number of random tape seeks and, consequently, very poor performance. On the other hand, combining the individually optimized schedules of a batch of queries to produce a global optimized schedule minimizes the number of random tape seeks and maximizes performance. In the following section we explore the sensitivity of our results to the batch size.

5.5 Batch Size Sensitivity

To further evaluate the effectiveness of query batching, we next conducted a set of tests to measure the sensitivity of the results to the size of the batch. Eight clients were used, each running five queries randomly selected from the 11 benchmark queries. The total elapsed time for all clients to complete the execution of their five queries for batch sizes 1, 2, 4 and 8 is shown in Figure 13. As expected, the performance improvement provided by query batching increases as the batch size is increased due to a reduction in the number of random tape seeks performed. However, the gains obtained are not a linear function of the batch size. The reason for this is that there is a wide range of execution times of the various queries in the benchmark. For any batch that includes a long-running query, the relative effect of query batching is reduced due to the fact that most of the time is spent executing the long query by itself. Queries that arrive subsequently cannot be admitted into the current batch due to the long query's out of band access region. Besides, the dynamic admission scheme of query batching may cause extra delays on queries need to access blocks with lower addresses. Nevertheless, noticeable improvements can be achieved by increasing the batch size. Figure 14 shows similar results for the same test repeated for 8 clients with each running 10 queries.

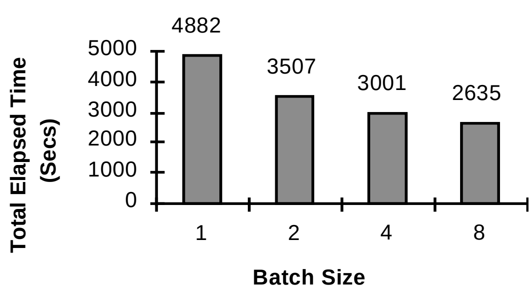

Figure 13: Query Batching – 8 Clients X 5 Queries

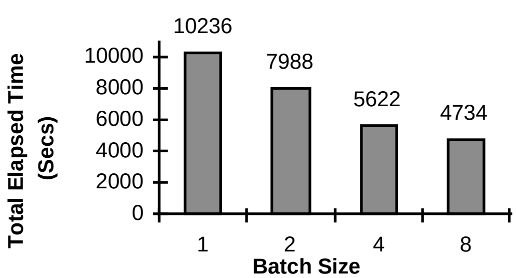

Figure 14: Query Batching -- 8 Clients X 10 Queries

6. Summary

In this paper we have described how the Paradise database system was extended to support the execution of queries against image data sets stored on magnetic tape. While our extensions to the Paradise storage engine (i.e. the SHORE storage manager) emphasized the transparent access to tape-resident storage volumes, the main contribution of this paper is a set of techniques that we developed to optimize tape accesses during query execution. The techniques of *object abstraction*, *query pre-execution*, *cache*-conscious *tape scheduling*, and *run-time, data-driven query scheduling* were demonstrated to be extremely effective in optimizing the execution of queries accessing raster images on tertiary storage in a single-user environment. When these mechanisms are augmented with *query batching*, almost a 30-fold performance improvement was obtained in a multi-user environment. One of the biggest advantages of query pre-execution and batching is its simplicity. We were able to implement the techniques without major modification of any of the higher level query optimizer and operator code. This bottom-up reordering and scheduling strategy enables us to preserve the execution order of the original query plan, yet still leaves room for new higher level efforts to further optimize query execution in a top-down fashion.

Acknowledgments

We would like to thank Navin Kabra and Jignesh Patel for patiently answering detail questions on the Paradise query optimizer and query execution engine, and Kristin Tufte to provide valuable comments on the initial draft of this paper.

References

[1] D. DeWitt, N. Kabra, J. Luo, J. Patel and J. Yu. "Client Server Paradise," Proc. of the 20th VLDB Conference, Santiago, Chile, 1994.

[2] J. Patel, J. Yu., et al. "Building a Scalable Geo-Spatial DBMS: Technology, Implementation and Evaluation," Proc. of the 1997 SIGMOD Conference, May, 1997.

[3] B. Kobler and J. Berbert, "NASA Earth Observing System Data Information System (EOSDIS)," Digest of Papers: 11th IEEE Symposium on Mass Storage Systems, Los Alamitos, 1991.

[4] Quantum Corporation, "DLT-4000 Product Manual," 1995.

[5] M. Carey, L. Haas and M. Livny. " Tapes Hold data, Too: Challenges of Tuples on Tertiary Store," Proc. of the 1993 SIGMOD Conference, May, 1993.

[6] J. Gray. “MOX, GOX and SCANS: The Way to Measure an Archive,” June 1994.

[7] R. Herring and L. Tefend. “Volume Serving and Media Management in a Networked, Distributed Client/Server Environment,” Proc. 3rd Goddard Conf. Mass Storage Systems and Technologies, NASA Con. Publication 3262, 1993.

[8] M. Olson. “Extending the POSTGRES Database System to Manage Tertiary Storage,” Master’s Thesis, EECS, University of California at Berkeley, May, 1992.

[9] S. Sarawagi. “Query Processing in Tertiary Memory databases,” Proc. of the 19th VLDB Conference, Switzerland, September, 1995.

[10] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. “The SEQUOIA 20000 Storage Benchmark,” Proc. of the 1993 SIGMOD Conference, May, 1993.

[11] J. Gray, et. al. “The Sequoia Alternative Architecture Study for EOSDIS”, NASA, October 1994.

[12] E. Shekita, M. Carey, “A Performance Evaluation of Pointer-Based Joins,” Proc. Of the 1990 SIGMOD Conference, May, 1990.

[13] T. Keller, G. Graefe, D. Maier, “Efficient Assembly of Complex Objects,” Proc. of the 1991 SIGMOD Conference, May, 1991.

[14] J. Kohl, C. Staelin and M. Stonebraker. “Highlight: Using a Log-Structured Files System for Tertiary Storage management,” Proc. Winter USENIX 1993, pages 435-447, San Diego, CA, January 1993.

[15] D. Ford and J. Myllymaki. “A Log-Structured Organization for Tertiary Storage,” Proc. of the 12th Conference on Data Engineering.

[16] M. Rosenblum and J. Ousterhout. “The Design and Implementation of a Log-Structured File System,” ACM Trans. Computer System, 10(4):26-52, February 1992.

[17] R.Grossman, D. Hanley and X. Qin,. “Caching and Migration for Multilevel Persistent Object Stores,” Proceedings of the 14th IEEE Computer Society Mass Storage System Symposium, Sept. 1995.

[18] J. Li and C. Orji. “I/O Optimization Policies in Tape-Based Tertiary Systems,” Tech Report, Florida International University.

[19] B. Hillyer and A. Silberschatz. “Random I/O Scheduling in On-line Tertiary Storage Systems,” Proc. of the 1996 SIGMOD Conference, May, 1995.

[20] L.T. Chen, D. Rotem. “Optimizing Storage of Objects on Mass Storage Systems with Robotic Devices,” Algorithms for Data Structures, Spring V, 1994.

[21] L.T. Chen, R, Drach, M. Keating, S. Louis, D. Rotem and A. Shoshan. “Efficient Origination and Access of Multi-Dimensional Datasets on Tertiary Systems,” Information Systems Journal. April, 1995.

[22] S. Sarawagi and M. Stonebraker. “Single Query Optimization for Tertiary Memory,” Proc. of the 1994 SIGMOD Conference, May, 1994.

[23] J. Myllymaki and M. Livny. “Joins on Tapes: Synchronizing Disk and Tape Join Access,” Proc. of the 1995 SIGMETRICS Conference, Ottawa, Canada.

[24] J. Myllymaki and M. Livny. “Efficient Buffering for Concurrent Disk and Tape I/O,” Proceedings of Performance '‘96 - The International Conference on Performance Theory, Measurement and Evaluation of Computer and Communication Systems, October 1996.

[25] M. Stonebraker. “Managing Persistent Objects in a Multi-Level Store,” Proc. of the 1991 SIGMOD Conference, May, 1991.

[26] J. Fine, T. Anderson, K. Dahlin, J. Frew, M. Olson, D. Patterson. “ Abstract: A Latency-Hiding Technique for High Capacity Mass-Storage Systems,"” Sequoia 2000 Project Report 92/11, University of California, Berkeley, 1992.

[27] S. Sarawagi and M. Stonebraker. “Reordering Query Execution in Tertiary Memory Databases,” Proc. of the 20th VLDB Conference, India, September, 1996.

[28] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White and M. Zwilling. “Shoring up Persistent Objects,” Proc. of the 1994 SIGMOD Conference, May, 1994.

[29] J. Yu and D. J. DeWitt. “Processing Satellite Images on Tertiary Storage: A Study of the Impact of Tile Size on Performance,” 5th NASA Goddard Conference on Mass Storage Systems and Technologies, September, 1996.