Query Processing in Tertiary Memory Databases*

Sunita Sarawagi

Computer Science Division, 396 Soda Hall
University of California, Berkeley, CA 94720, USA
sunita@cs.berkeley.edu

Abstract

With rapid increase in the number of applications that require access to large amounts of data, it is becoming increasingly important for database systems to handle tertiary storage devices. The characteristics of tertiary memory devices are very different from secondary storage devices that conventional database systems are designed for. This requires new approaches to managing data location and movement, together with query execution in a unified framework. In this paper we present methods of scheduling queries, caching and controlling the order of data retrieval for efficient operation in a tertiary memory environment. We show how careful interspersing of queries and informed cache management can achieve remarkable reductions in access time compared to conventional methods. Our algorithms use a few model parameters for each tertiary memory device and are thus designed to be portable across a wide variety of tertiary memory devices and database types. We are extending the POSTGRES database system to implement the new query processing strategies. Initial measurements on the prototype yield impressive results.

*This research was sponsored by NSF Grant IRI-9107455, ARO Grant DAAL03-91-G-0183, and DARPA Contract DABT63-92-C-0007. Additional support was provided by the University of California and Digital Equipment Corporation under Sequoia 2000 research grant #1243.

**Proceedings of the 21st VLDB Conference
Zurich, Swizerland, 1995**

1 Introduction

Applications manipulating large volumes of data are growing in number: earth observation systems, historical data base systems, statistical data collections and image and video storage systems are a few examples. There is increasing consensus amongst database researchers [Sto91] [CHL93] [Sel93] [Moh93] regarding the need of a database controlled tertiary memory for storing massive amounts of data.

A major limitation of traditional DBMSs is the assumption that all data resides on magnetic disk or main memory. Therefore all optimization decisions are oriented towards this technology. Tertiary memory, if used at all, functions only as an archival storage system to be written once and rarely read. Some database systems [Isa93] allow data to be stored on tertiary memory, but they do so by using a file system to get transparent access to data and store only metadata information in the database system. This means that the tertiary memory is not under direct control of the database system. One important exception is POSTGRES [Ols92]. POSTGRES includes a Sony optical jukebox [Son89] as an additional level of the storage hierarchy. The POSTGRES storage manager can move data transparently between a disk cache and the jukebox using a LRU replacement strategy. While this prototype implements the storage manager for tertiary memory, a lot of issues related to tertiary memory specific performance optimization still remain unexplored.

Tertiary memory devices pose a challenge to database designers because their performance characteristics are very different from those of magnetic disks. A typical device consists of a large number of storage units, a few read-write drives and even fewer robot arms to switch the storage units between the shelves and the drives. A storage unit, which we generically call a *platter*, is either a tape cartridge or an optical disk. In Table 1 we compare several tertiary memory devices with a magnetic disk. The characteristics shown are exchange time (time to unload one storage unit from the drive and then load a new unit and

Storage device	Exchange time (sec)	Full seek time (sec)	Data transfer rate (KB/sec)	Transfer time for 128 KB	Worst/best access (sec)
Optical disk	8	0.3	500	0.256	32.4
Helical scan tape	6	135	4000	0.032	4406
Optical tape	>60	90	3000	0.043	3488
Magnetic disk	-	.06	4250	0.03	3

Table 1: Comparative study of the characteristics of different storage devices.

get it ready for reading), maximum seek time, data transfer rate, transfer time for 128 KB of data and the ratio between the worst case and best case times to access and read 128 KB of data from tertiary memory. From the last column we note that magnetic disks are a relatively uniform storage medium, compared with tertiary memory. Worst case access times are only a factor of three larger than best case times whereas some tape oriented devices have three orders of magnitude more variation making it crucial to carefully optimize the order in which data blocks are accessed on these devices.

Research issues

In this paper we address the issues raised by this wide performance gap between secondary and tertiary devices. First, it becomes very important to **avoid small random I/Os**. Unclustered index scans and joins in limited buffer space can lead to disastrous performance if processed in the traditional way on tertiary memory. Consider a two-way join query between relation R stored on platters 1 and 2 and relation S divided between platters 2, 3 and 4 such that each relation is much larger than the cache. Any join processing method that is oblivious of such a layout cannot do efficient batching of accesses to one platter and may access data randomly across the four platters, leading to many platter switches.

Second, careful **query scheduling** can be employed to optimize accesses to tertiary memory. For instance, consider the case where we have two select queries on relations R and S respectively both of which are spread over three platters and there is only one read-write drive. If we intersperse the execution of these select queries so that every time we load a platter we schedule the two select queries on the fragment of the relations stored on that platter, then each platter will be loaded only once.

Third, we need **unconventional caching strategies** for managing the magnetic disk cache. A relation can be cached when its platter is just about to be unloaded even if we do not intend to execute queries on it immediately. For instance, if we have a join query between relation R on platter 1 and S on platter 2 and another join query between T on platter 1 and U on platter 2, it might help to cache both R and T when platter 1 is loaded even if we are scheduling the join between R and S first. Contrast this with the caching on demand strategies.

There are two additional challenges to solving the above problems. First, tertiary memory devices differ widely not only from typical secondary memory devices, but also among themselves. For some devices the platter switch cost is high making it important to reduce the number of I/O requests and for others the data transfer bandwidth is low making it important to reduce the amount of data transferred. Tape devices have significant seek overhead whereas disk devices allow random access. Second, we expect a lot of variation in the applications that use tertiary storage devices. Some involve a large number of relatively small objects whereas others require a small number of very large objects. It is essential for query processing methods to be aware of these differences and optimize accordingly.

Design approach

We present a two-phase query execution model. The first part is a query optimizer and the second part is a scheduler that controls the execution order of queries, the movement of data from the disk cache to tertiary memory and the combining of different queries that share data access or computation. We use the notion of a **fragment** for exposing the layout of a relation on tertiary memory to the query optimizer. A fragment is the part of a relation that lies contiguously on one platter (a fragment lying contiguously on one platter can be broken into smaller fragments as explained later). By executing queries and moving data in units of fragments we first eliminate small random I/Os. We then design policies for fetching and evicting fragments to further reduce the number of platter switches and seeks on tertiary memory. By identifying a few crucial device and workload parameters that are used to drive our optimization process, we make our system robust. For the initial version, we restrict to single relational queries and two-way joins. We plan to extend our design to handle multi-way joins in future.

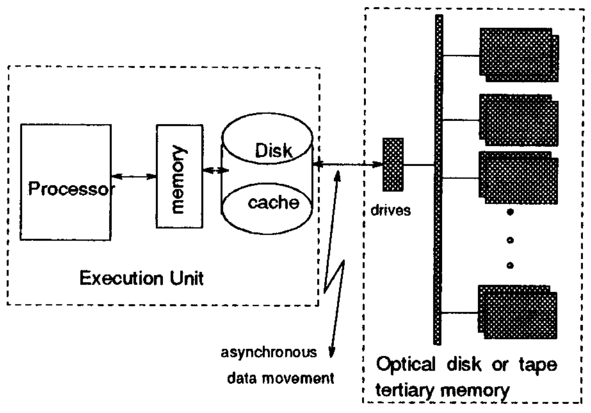
Processo
memory
Disk
cache
Execution Unit
asynchronous data movement
drives
Optical disk or tape tertiary memory

Figure 1: The Physical Configuration.

Paper organization

In Section 2 we present our query processing architecture and describe in detail the working of the query optimizer and the scheduler. The proposed framework raises issues regarding query reordering and fragment access. Since exact optimization is intractable, we develop heuristics. In the paper, we present only the final policies selected by experimental evaluation (Section 3). In Section 4 we present implementation details of the query processing architecture and present results of running the Sequoia benchmark queries on an initial version of the system. Section 5 contains related work and Section 6 gives concluding remarks.

2 The query processing architecture

We assume an extended relational architecture with a three level memory hierarchy: tertiary memory attached to a disk cache attached to main memory as shown in Figure 1. We do not impose any restrictions on the layout of a relation on tertiary memory. A relation can be larger than the disk cache, can lie over more than one platter, and can be spread arbitrarily across a platter.

To identify the part of the relation that lies contiguously on one platter we divide a relation into **fragments** of appropriate size. A fragment can be fetched as a whole without incurring platter switches and seeks during its transfer. The proper choice of fragment size is crucial to performance. The best fragment size is a function of the request size distribution, the platter switch cost, the transfer cost and the seek cost. If the fragment size is small, we make more I/O requests and the latency of first access is incurred too many times. If the fragment size is large, the transfer overhead is higher because we might be transferring extraneous data. In Section 3.3.1 we show how we can capture this tradeoff in an analytical formula that can yield reasonable values for the fragment size in a particular setup.

We will next describe our two-phase query processing engine. In the first phase (§2.1) queries are decomposed into basic executable units. In the second phase (§2.2) these are scheduled.

2.1 Optimizing queries

During the query optimization phase, each query is broken down into a number of subqueries on the fragments. E.g., a join query between relation R consisting of m fragments and relation S consisting of n fragments is broken down into mn join queries between the individual fragments. Each subquery is then optimized separately. We fix the maximum size of a fragment such that all data required by a subquery can be held totally in the cache. This means that the optimizer can view the subquery like a regular query on secondary memory and optimize accordingly. Although such fragmentation may give rise to a large number of subqueries to be optimized, it is often possible to generate one optimized template and reuse it for each subquery. Our preliminary model has some simplifications. We discuss extensions in §2.3.

2.2 Scheduling queries

The subqueries generated above are submitted to the scheduler. The scheduler fetches the fragments that are required by the subquery from the tertiary memory, puts them in the disk cache, and then schedules the subquery for execution. The scheduler knows about the state of the tertiary memory (the storage media currently loaded, the current head position etc), has knowledge of the semantic contents of the cache (not just physical page addresses) and knows about the data requirements of each subquery. It uses this global knowledge to decide on the order in which fragments are moved from the tertiary memory to the disk cache and the order in which subqueries are scheduled. The responsibilities of the scheduler can be listed as follows:

- which fragment to fetch next from the tertiary memory when the I/O unit becomes free,
- which fragment(s) to evict from the cache to make space for the fragment chosen above and
- which subquery on the cached fragments to be processed next.

We will describe how the scheduler handles its responsibilities of fetching and evicting fragments in §3.1 and §3.2. In the current version of the system, the scheduler submits a subquery for processing as soon as all its fragments are cached. We plan to optimize this part of the scheduler to do multiple query optimization between the subqueries.

2.3 Extensions to the model

A number of extensions were made to this model of query processing to handle relations with large objects, to allow more efficient use of indexing and to avoid redundant processing. The important ones are listed below:

- Databases often have images and video clips which are stored as large objects. In our model, we assume that each large object is stored as a separate fragment and a select query on a relation with one of the attributes a large object is executed in two phases. In the first stage, we do a select on the base relation, get a list of large objects to be fetched and fetch them in any order in the second phase.

- We assume that each fragment has its own index. Depending on the size of the index, the DBA can choose to store it either on magnetic disk or tertiary memory. When doing an index scan on a relation, it might help to scan the index trees first, find out which fragments contain qualifying tuples and fetch only those fragments later. This will help remove random I/Os which can be wasteful on tertiary memory.

- Although breaking queries into independent subqueries is favorable for reducing I/O costs to tertiary memory, we may pay higher processing cost for some queries. For instance, in a hash join if the probe relation is broken into n fragments, then the hash table for each fragment of the build relation has to be constructed n times. To reduce this overhead, we will modify our scheduler to order the execution of subqueries so that whenever possible the hash table can be shared across multiple subqueries. This will be treated as a part of the general multiple query optimization to be handled by the scheduler.

- For some queries the order of the result tuples is important and executing subqueries independently is not possible. In our initial model, we are ignoring queries that require sorted results.

3 Scheduling policies

3.1 Fragment fetch policies

The scheduler has a pool of tasks which are either two-way joins or select queries on a single fragment or fetch request for a list of large objects. For an index scan on a fragment with the index residing on tertiary memory, we view the index tree as another fragment and the index scan query as a join between the index and the base fragment. Implicitly, this collection of plans forms a *query graph* where the nodes denote the fragments and the edges denote the joins between two fragments. In this graph, an edge between two nodes implies that both the fragments represented by these nodes must reside in the cache together for the query to be processed. Fragments which do not join with any other fragment will be represented as isolated nodes. We are given a limited amount of disk cache, typically, less than the sum of the sizes of the fragments queried. The query graph keeps on changing as queries get completed and new queries arrive.

At any time, there is a pool of subqueries waiting to be executed, each of these subqueries requires one or more fragments to be present in the cache for processing. Of the fragments required, some fragments are already in the disk cache and others need to be fetched from tertiary storage. Of these fragments, some reside on platters that are currently loaded and others reside on unloaded platters. Our objective is to migrate these fragments to and from tertiary memory and the disk cache so as to minimize the total time spent doing I/O on tertiary memory.

The above on-line problem is NP-complete since an off-line restriction of the formulation has been shown to be NP-complete in [MKY81]. Hence, an algorithm that finds the optimal solution is likely to be too expensive to be useful. Consequently, we use a number of heuristics for reducing the search space.

Design Methodology

The design of a good heuristic for fetching fragments is made challenging by the large number of parameters, e.g., cache size, number of users, size of fragments, platter switch cost, data transfer cost and seek cost. In order to control the complexity, we designed the algorithm in multiple stages. We first started with an algorithm that minimizes transfer cost, then we added the platter switch cost to the cost model and refined the algorithm to minimize the sum of the platter switch and transfer cost. Finally, we incorporated seek cost into the cost model by refining the algorithm. For brevity we present the final resulting set of heuristics. We used extensive simulation to aid us in the search for good heuristics.

Optimizing for transfer cost

We first started with the case where the platter switch and seek overhead is zero and minimizing I/O time is equivalent to minimizing the total bytes transferred. Even this problem is NP-complete. Hence, we tried out different heuristics for deciding on the order in which fragments should be fetched from tertiary memory. Some of the important heuristics were: fetch fragment with the largest number of queries next; fetch the smallest fragment next; and fetch fragment that joins with the maximum number of cached fragments next. Amongst these and others that we tried, we found that the policy which performed the best overall was:

> POLICY-1: Fetch fragment that joins with the largest sum of sizes of cached fragments. Resolve ties by choosing fragment that has the greater number of queries.

Incorporating platter switch cost

To enable POLICY-1 to optimize for both platter switches and transfers we refined it as follows: As long as there are fragments on the loaded platters that join with the cached fragments we fetch fragments from the loaded platters. When there are no more fragments of that type, we could either fetch fragments from the loaded platters or load a new platter that contains fragments which join with the cached fragments using the order given by POLICY-1. This decision depends on the amount of cache space available. If the cache space is large so that we do not have to evict active fragments from the cache, we can fetch fragments from the loaded platters, or else, we need to switch platters. The modified policy is given below:

```
POLICY-2
Fetch next fragment that joins with the cached
  fragments and resides on a loaded platter.
If no such fragment,
  If ("no room in cache")
     Switch to an unloaded platter choosing
       platter with fragments that join with
       maximum cached fragments
     Fetch fragment from the chosen platter
  Else
     Fetch fragment from the loaded platters
     If no fragment on the loaded platters,
       switch an unloaded platter choosing
       platter with maximum queries
```

We need a method to estimate if there is "room in cache" for fragments on the loaded platters that do not join with the cached fragments. Clearly, just using the total size of the cache for estimating this predicate is not sufficient because the current set of active fragments in the cache and the fragments that they join with play an important part. Let a be the total size of active fragments in the cache and b be the total size of fragments that join with cached fragments. Hence $a+b$ is an estimate of the amount of cache space that will be needed in the future.

This leads to the notion of *pressure* on the cache:

$$\text{Cache pressure} = \frac{a+b}{C},$$

where C is the cache size. Thus the pressure expresses potential demand for the cache as a fraction of the cache size. We can use cache pressure to determine if there is any room for unrelated fragments. The predicate "no room in cache" then translates to "cache pressure > threshold". Next we need to choose a value of the "threshold". Using the same value of the threshold for tertiary memory of widely varying characteristics is not suitable. A low value of the threshold implies more frequent platter switches, which is unsuitable for tertiary memory devices with high switch cost. Similarly, high value of the threshold implies more active eviction of cached fragments, which is unsuitable when the data bandwidth is low. To understand these tradeoffs, we tried the above algorithm for different values of the threshold, over different tertiary memory devices and workload parameters. From our experiments we observed that one important parameter that affects the threshold is *the ratio of the platter switch time to the average transfer time incurred in fetching a fragment*. When the value of the threshold was set to be this ratio we obtained the best overall performance. Hence, in our heuristics we set the value of the threshold to this ratio.

Incorporating seek cost

The seek cost on tape devices consists of a fixed startup cost and a variable search/rewind cost. The only way we can reduce the startup cost is by making fewer I/O requests. The variable search/rewind cost can be reduced by fetching fragments in the order in which they are placed on tape. In our policies so far we have used a ranking function based on join size for determining the order in which fragments are fetched from a loaded platter. While this order is good for reducing transfer time, it is preferable to fetch fragments in their storage order when the goal is to reduce seek cost. Thus, we need to identify which cost is more important to optimize at any time.

Suppose we have a tape of capacity T bytes, transfer rate d bytes/second and seek rate s bytes/second. Assuming that on an average the seek distance is a fraction, f, of the tape, the average seek cost is Tf/s seconds. This means that seek time dominates transfer time only for fragments smaller than Tdf/s bytes. In most tapes, the seek rate is 10–100 times higher than the transfer rate (refer Table 2), so the object size has to be smaller than 150*th* the tape capacity for the seek cost to dominate the transfer cost (for $f = 1/3$). Hence, when choosing fragments from a loaded platter, if Tdf/s exceeds the average fragment size, we use proximity to the tape head as the criteria for choosing the next fragment. This formulation assumes that the seek cost is linearly proportional to the distance seek-ed. This assumption does not hold for DLT tapes where the seek rate is higher for larger seek distances. For such tapes we need to put the average seek cost in the formula instead of deriving the average seek cost from the seek rate.

3.2 Fragment eviction policies

Once a fragment is selected for fetching, we choose fragments to be evicted from the cache to make space for this fragment. Like the fetch policy, our eviction policy is also based on the careful combination of a number of simple heuristic policies.

The classical cache replacement policy is LRU when all objects are of the same size and WEIGHTED-LRU when the objects are of varying size. In our case, we might also have to evict fragments which have pending queries on them. This makes policies like LRU and WEIGHTED-LRU meaningless since we already know that the fragment will be used in the future. Hence, to choose among fragments with pending queries we use a policy we call LEAST-WORK, which evicts the fragment with the fewest remaining queries.

Ties are resolved using a policy we call LEAST-OVERLAP. Intuitively, while resolving ties, we want to avoid evicting fragments that join with many overlapping fragments so that when the overlapping fragment is fetched it can complete joins with many fragments together. Thus, policy LEAST-OVERLAP chooses the fragment with the least overlap between fragments that join both with the given fragment and other cached fragments. Our final eviction policy is given below.

```
Choose fragment using LEAST-WORK
Resolve ties by using LEAST-OVERLAP
Resolve further ties using WEIGHTED-LRU.
```

3.3 Performance results

Evaluating the benefit from various policies is a difficult task, in part because it is unclear what the baseline performance ought to be. In particular, it is not feasible to pick the optimal schedule as the baseline because the search space is absurdly large, even for problems of reasonable size. Our approach was to estimate bounds on the optimal performance and compare the performance of our policy against these bounds. The baseline policy merely provides a scale for comparison; absolute performance numbers are less significant.

Another practical issue that arises is the choice between real vs. simulated tertiary devices. Loading data (of sizes up to a terabyte) and running queries is an inconveniently slow process. Besides, a small set of tertiary devices gives us but a few data points regarding performance parameters, whereas much of our intuition in heuristic design originated from a deeper understanding of the parameter space. Therefore, we used an event driven simulator where workload, device, and heuristics were all flexible. Details of the simulation setup are presented next.

3.3.1 Simulation details

Our simulator consists of a centralized database system serving requests from different query streams. We model a closed queuing system consisting of multiple users who submit a query, wait for the result, and then think for an exponentially distributed time before submitting the next query. Table 2 lists the performance specifications of the four tertiary memory types we used in our study: (1) the Sony WORM optical jukebox, (2) the Exabyte 8500 tape library, (3) the Metrum RS6000 tape jukebox and (4) Sony's DMS tape library. These devices were chosen so as to cover adequate representatives from the diverse tertiary memory hardware in existence today. Table 3 lists the three datasets that we used as the underlying database. Each dataset is characterized by the range of sizes of the relations and the number of relations. The size of a relation is assumed to be uniformly distributed within the range specified by the dataset. The default size of the cache and the number of users is given in Table 4. Further details about the simulator are given below:

Relation layout

For laying out the relations on tertiary memory we use the following approach: We divide a relation into partitions of size no more than p. The value of p is always $\leq$ to the platter capacity. These partitions are laid out contiguously on the platters. A partition is stored with equal probability in one of the partially filled platters that has space for it or a new platter if one is available. We will denote the total number of platters over which data is spread as P. For disk-based platters, the layout of data partitions within a platter is not modeled. For tapes, the space between two adjacent partitions is uniformly distributed between 0 and the total free space left on tape over the number of relations that are assigned to the tape.

Workload

Table 4 summarizes the relevant workload parameters and their default values. We simulate a stream of single relation queries and two-way joins. Base relations for queries are chosen using the 80-20 rule i.e, 80% of the accesses refer to 20% of the relations. The scan

	DAT autochanger 1200C	Sony	Exabyte	Metrum	DMS
classification	tape stacker	small optical jukebox	small tape library	large tape library	large tape library
switch time (sec)	101	8	171	58.1	39
transfer rate (MB/sec)	0.17	0.8	0.47	1.2	32
seek rate (MB/sec)	23.1	-	36.2	115	530
seek start (sec)	11	0.5	16	20	5.0
number of drives	1	2	4	5	2
platter size (GB)	2.0	3.27	5	14.5	41
number of platters	12	100	116	600	320
total capacity (GB)	24	327	580	8700	13120

Table 2: Tertiary Memory Parameters: The switch time is a summation of the average time needed to rewind any existing platter, eject it from the drive, move it from the drive to the shelf, move a new platter from shelf to drive, load the drive and make it ready for reading. The seek startup cost is the average of the search and rewind startup cost and the seek rate is the average of the search and rewind rate.

Dataset	# relations	range of sizes	total size
SMALL-DATASET	2000	5 MB to 50 MB	50 GB
MEDIUM-DATASET	400	250 MB to 2.5 GB	500 GB
LARGE-DATASET	80	12.5 GB to 125 GB	5 TB

Table 3: Datasets: sizes of the relations are uniformly distributed across the given range

on the base relation can be either a sequential scan, a clustered index scan or an unclustered index scan. In our setup 20% of the scans are assumed to be sequential and the rest are clustered or unclustered with equal probability. The selectivity of an index scan can be anywhere between 0.1 and 0.2. We assume in these experiments that all indices reside on magnetic disks and the index tree is pre-scanned to get a list of fragments that contain qualifying tuples.

Execution model

The processing time of a query after the component fragments are fetched from tertiary memory is computed as the sum of the time needed to read/write data between disk and the main memory and the CPU processing time. In Table 4, we list the number of instructions required for various query types. The time to process a join is derived assuming a hash join method. The time to read a page from disk is modeled as a sum of the average seek time and the time to transfer a page of data. Our model of the execution unit is not very detailed, but this hardly impacts the accuracy of our results because we are taking measurements of only tertiary memory I/O in this paper.

Fragment Size

For a given database and tertiary memory, we determine a maximum fragment size, F. Any partition larger than size F is divided into fragments of size

Description	Default
Workload	
Mean think time	100 sec
Number of queries per run	800
Number of users	80
Join fraction	0.8
Sequential scan fraction	0.2
Selectivity	0.1-0.2
Execution Model	
MIPS	50
Instructions for seq scan	100 per tuple
Instructions for index scan	200 per tuple
Instructions for hash join	300 per tuple
Instructions for starting a scan	20,000
Tuple size	400 bytes
Disk Characteristics	
Average seek time	20ms
Data transfer rate	5 MB/sec
Cache size	3% of database size

Table 4: Simulation Parameters and their default values.

Tertiary Memory	SMALL DATASET	MEDIUM DATASET	LARGE DATASET
Sony	4	-	-
Exabyte	20	300	-
Metrum	20	250	5000
DMS	50	1500	20000

Table 5: Maximum fragment size (in MB) for each tertiary memory and dataset pair

at most F. The optimal fragment size depends on the transfer time, access latency, the request size distribution and the kind and degree of sharing between queries. In general, it is hard to find the optimal fragment size since it is difficult to get exact specification of the request size distribution and the degree of sharing between queries. However, one can use approximate ideas about the expected pattern of reference for determining reasonable values of fragment size analytically. We present below one such method.

Let f be the average access latency, d be the data transfer rate and R be the maximum size of a relation. For a fragment of size F, the average time required to read n bytes of data is:

$$T(n, F) = \left\lceil \frac{n}{F} \right\rceil (f + Fd)$$

If $p(n)$ denotes the probability that a request is of size n, then the average access cost for a request is:

$$A(F) = \sum_{n=1}^{n=R} T(n, F)p(n)$$

For a given value of d, f and R we can calculate the value of F for which $A(F)$ is minimum by plotting a graph of $A(F)$ versus F.

Using the workload parameters in Table 4 to get estimates of the request size distribution, we plotted $A(F)$ versus F for different relation sizes and tertiary memory devices. It was observed that there was not much variation in the optimal fragment size for relations in the same dataset. Hence, for each tertiary memory and dataset pair we chose a fragment size. In Table 5 we show the fragment size we obtained for each tertiary memory and dataset pair using this method. Estimates like these could be used by the database designer to choose the fragment size for a particular setup.

3.3.2 Estimating performance bounds

We define our baseline policy to be first come first serve (FCFS) for fetching fragments with LRU for evicting fragments. We first used this policy to estimate the fraction of time that is spent in doing tertiary memory data transfers, seeks and platter switches. We then estimated bounds on maximum achievable reduction in I/O time based on the number of queries per relation and the number of queries per platter. If a relation has q queries on it, then the maximum possible reduction in transfer time is $(q - 1)/q$. Similarly, if a platter has r queries on it, the maximum possible reduction in number of platter switches is $(r - 1)/r$. In reality, these improvements might be unachievable because of further limitations imposed by the amount of cache.

In Table 6 we list the percentage of time spent in transfers (column 2), platter switches (column 3) and seeks (column 4) using the baseline policy. We give our calculation of the values of q and r in columns 5 and 6 respectively. We show the maximum possible reduction in total I/O time by reducing the transfer time and the switch time in columns 7 and 8 respectively. The maximum improvement achievable by reducing seeks is more difficult to analyze. By summing columns 4, 7 and 8 we can get an upper bound to the maximum improvement that can be achieved by any policy (column 9). For instance, for SMALL-DATASET on the Sony although 75% of the tertiary memory I/O time is spent in data transfers, we cannot reduce the transfer time any further because there is only one query per fragment on an average. The only saving we can get is by reducing the number of platter switches. There are about $q = 20$ queries per platter, hence there is a possibility of getting a 95% (= (20-1)/20%) reduction in switch time and, a 24% (= 25% × 95%) reduction in total I/O time by reducing the switch time.

The last column in Table 6 lists the improvement we achieved over the baseline policy using our fetch and eviction policy. These numbers are very close to the maximum estimated improvement in column 9. Some deviations can be explained by the seek overhead which we could not quantify exactly and others by the inherent limits placed by the limited amount of cache.

4 Implementation

We are extending the POSTGRES relational database system to support the proposed query processing architecture. The original storage manager used an LRU managed disk cache. It was modified to take hints from the scheduler in deciding which data blocks to evict from this cache and which to fetch next. The scheduler was thus able to control the movement of fragments from the disk cache to the tertiary memory. In the old architecture all user-sessions run independently as separate POSTGRES process — as a result there is no synchronization between I/O requests to the tertiary device. In the new architecture, each user

1	2	3	4	5	6	7	8	9	10
Tertiary	% total TM I/O spent in			# of queries per		max. % reduction by		improvement	
memory	transfer	switch	seek	relation (q)	platter (r)	transfers	switches	projected	achieved
Sony	71.6	25.0	3.4	1	20.2	0	24	27.4	27
Exabyte	23.2	42.1	34.7	1	6.8	0	36	70.7	66
Metrum	16.7	31.9	51.4	1	14.4	0	30	81.4	63
DMS	1.9	58.6	39.4	1	14.4	0	55	94.4	83
				SMALL-DATASET					
Exabyte	83.8	8.6	7.6	1.35	1.7	22	3.7	33.3	31
Metrum	75.5	6.9	17.6	1.35	5.18	20	5.6	43.2	37
DMS	30.0	22.3	47.7	1.35	7.2	8	19	74.7	42
				MEDIUM-DATASET					
Metrum	97.0	1.1	1.9	5.96	5.96	81	1	83.9	46
DMS	81.5	5.8	12.7	5.96	5.96	67	5	84.7	59
				LARGE-DATASET					

Table 6: Percentage distribution of the time spent at various stages: Columns 7 and 8 represent the maximum reduction in transfer and platter switch cost possible. Column 9 is an upper bound on the maximum total improvement in total I/O time possible. Column 10 is the best improvement we could obtain by our policies. (Some dataset-tertiary memory pairs are missing because the dataset was too large to fit on that tertiary memory)

process first compiles and fragments the query and then submits the fragmented queries to a centralized scheduler process. The scheduler maintains a number of slave backend processes. These processes are used for transferring data from the tertiary memory to the disk cache. The number of processes of such type is equal to the number of drives in the tertiary memory device. This way it is possible to employ all the drives of the tertiary memory in parallel for transferring data. Submitting multiple requests to multiple drives also helps hide some of the latency of platter load/unload operation — when one drive is transferring data, the robot arm is free and can be employed for switching platters on some other drive. When all the data required by a particular subquery have been put in the disk cache the corresponding user process is notified. The user process can then execute the subquery whenever it is free. After finishing execution of the subquery it notifies the scheduler which can evict the fragments used by this subquery when desired.

Two tertiary memory storage devices – a Sony optical jukebox and an HP magneto-optical jukebox have already been interfaced with the POSTGRES's storage manager switch as described in [Ols92]. In addition, to facilitate measurements on robots for which the actual device was unavailable to us, we interfaced a tertiary memory device simulator to POSTGRES. The simulated storage manager used a magnetic disk for data storage but serviced I/O requests with the same delay as would an actual tertiary device which received the same request sequence.

To compare the performance of the new architecture with the old one and also to evaluate the payoffs of the policies in a real system, we measured the performance of the Sequoia benchmark [SFGM93] queries. We chose the national version of the benchmark which is of total size 18 GB. Since the data for the national benchmark was not available, we constructed the national benchmark by replicating the regional benchmark. The data was stored on a (simulated) tape stacker whose performance characteristics are summarized in Table 2 (Autochanger 1200C with DAT tapes). The database consists of four different kinds of relations: RASTER, POINT, POLYGON and GRAPH as summarized in Table 7. For the RASTER data, each tuple contains a 2-dimensional array of size 129 MB which is stored as a separate large object. The base table for the raster data is stored on magnetic disk whereas the two-dimensional raster images are stored on tertiary memory over 12 different DAT tapes. The POINT, POLYGON and GRAPH data are stored on one tape each. All indices reside on magnetic disk. The benchmark consists of 10 data retrieval queries which consist of two-way joins and select queries on various relations. The last query involves a "*" operator on the GRAPH table which we could not run on POSTGRES, instead we run a select query on the table. Since, the Sequoia benchmark does not have any information about the frequencies of posing individual queries – we let each user choose one of the 10 queries uniformly randomly. The default number of users was 5 and the total number of queries ran per user was 10. The size of the

Table name	# of tuples	tuple size	total size
RASTER	130	129 MB	16,744 MB
POINT	1,148,760	24 bytes	27.5 MB
POLYGON	1400, 000	204 bytes	286 MB
GRAPH	6500,000	175 bytes	1110 MB

Table 7: Sequoia Benchmark relations (national).

magnetic disk cache was varied as shown in Figure 2. The total time required to run the benchmark for the new architecture as compared against the old architecture is shown in Figure 2 for cache size of 32 MB and 64 MB. On moving from the old to the new architecture the total time reduces by a factor of 4 with a 32 MB cache and by a factor of 6.4 with a 64 MB cache. The main reason for this change is the reduction in the number of platter switches. The time to switch a tape is almost 1.6 minutes. Hence, when we reduced the number of switches from 2333 to 533 (for 32 MB cache size) the I/O time reduced by 50 hours.

5 Related work

Although tertiary memory devices are not common in databases they have long been used in mass storage systems like the NCAR's MSS [N+87], Lawrence Livermore Laboratory's LSS [Hog90] and the Los Alamos National Laboratory's CFS [C+82]. These are typically centralized supercomputing systems with multiple clients and use huge tape libraries for storing data. Disk caches are used for staging data in and out of tapes in units of a file. Files are brought from the tape library on user request and when space needs to be freed from disk, techniques like WEIGHTED-LRU [Smi81] are used to select files to be evicted next. Our environment is different from the conventional mass storage systems because we are working in a relational framework where it is possible to get more information about the nature of data accesses from the query semantics. Other areas where use of tertiary memory is gaining popularity recently is image archiving systems [SB+93] and multimedia databases [RFJ+93]. However, there is little reported work on the efficient use of the tertiary storage devices in this context.

Many device scheduling algorithms developed in a disk to main memory environment are relevant in our context. [SLM93] discusses the problem of reading a set of pages from disk to main memory so as to minimize the sum of the seek and transfer time. [BK79] and [Wie87] discuss scheduling policies for magnetic disk arms to minimize seeks. [MKY81] and [MR93] address the problem of minimizing the number of pages fetched from disk to a limited amount of main memory while processing a two-way join represented as a graph on the pages of the relation. This problem is a special case of our formulation for fetching and evicting fragments from the tertiary memory to the disk cache. [MSD93] discusses the problem of scheduling parallel hash joins in a batched environment. Query scheduling with the aim of reducing seek cost or platter switch cost in tertiary memory has been addressed in a few places: [KMP90] addresses the question of finding the optimum execution order of queries on a file stored on a tape and [Won80] addresses the problem of placing records with known access probability on tape to minimize expected head movement. [ML95] studies the benefit of doing hybrid hash join and nested loop join with the data still resident on tape instead of caching all of it on to disks before executing the query.

6 Conclusion

We presented the design of a query processing and cache management strategy that is optimized for accesses to a tertiary memory database. Our main contributions can be summarized as follows:

- We take a more unified and aggressive approach to reducing I/O on tertiary memory. Our system consists of a centralized scheduler that knows about the state of the tertiary memory, the disk cache and the queries present in the system. Instead of processing queries from separate users independently, the scheduler uses global consideration to decide on the order in which data required by the query will be fetched from tertiary memory and batches the I/O and computations of this query with other queries in the system.

- We employed the notion of a fragment to reveal the layout of the relation on tertiary memory to the query optimization and the cache management modules. Data is moved to and from the disk cache and the tertiary memory in units of fragments. This avoids small random I/Os, common in many conventional query execution methods thereby dramatically improving the performance of tertiary memory.

- We showed how we can further optimize tertiary memory I/O costs by carefully scheduling the order in which these fragments are fetched from tertiary memory and evicted from the disk cache. We developed a fragment fetch policy that performs well under a wide range of tertiary memory characteristics, workload types, cache sizes and system load and adapts dynamically to changes in these parameters.

- We are extending POSTGRES to implement this architecture. Initial measurements of the Sequoia

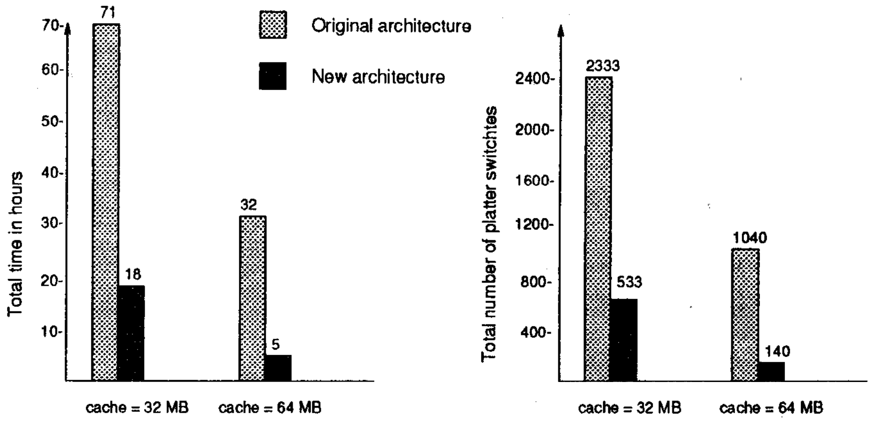

Figure 2: Result of running Sequoia benchmark on the original and new architecture in POSTGRES.

benchmark on the new architecture yield significant improvement over the old architecture used in POSTGRES.

Our next project is to extend the model so as to handle multi-way joins and sort-merge joins. We want to design the multiple query optimizer to reduce the time spent in processing queries. Finally, we would like to measure the payoffs we can get on more real-life workloads.

Acknowledgements

I would like to thank my advisor Mike Stonebraker for suggesting this topic and reviewing initial design of the system. I would like to thank my group-mate Andrew Yu for helping with the implementation of the system. Soumen Chakrabarti and Jolly Chen deserve special thanks for editing drafts of the paper. Finally, I would like to thank the reviewers for their useful feedbacks.

References

[BK79] F.W. Burton and J. Kollias. Optimizing disk head movements in secondary key retrievals. *Computer Journal*, 22(3):206–8, Aug 1979.

[C+82] B. Collins et al. A network file storage system. In *Digest of Papers, Fifth IEEE Symposium on Mass Storage Systems*, pages 99–102, Oct 1982.

[CHL93] M.J. Carey, L.M. Haas, and M. Livny. Tapes hold data, too: challenges of tuples on tertiary store. *SIGMOD Record*, 22(2):413–417, 1993.

[Hog90] C. Hogan. The Livermore distributed storage system: requirements and overview. In *Digest of Papers, Tenth IEEE Symposium on Mass Storage Systems*, pages 6–17, May 1990.

[Isa93] D. Isaac. Hierarchical storage management for relational databases. In *Proceedings Twelfth IEEE Symposium on Mass Storage Systems.*, pages 139–44, Apr 1993.

[KMP90] J.G. Kollias, Y. Manolopoulos, and C.H. Papadimitriou. The optimum execution order of queries in linear storage. *Information Processing Letters*, 36(3):141–5, Nov 1990.

[MKY81] T. Merrett, Y. Kambayashi, and H. Yasuura. Scheduling page-fetches in join operations. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 488–98, Sep 1981.

[ML95] J. Myllymaki and M. Livny. Disk tape joins: Synchronizing disk and tape access. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling on Computer Systems*, May 1995.

[Moh93] C. Mohan. A survey of DBMS research issues in supporting very large tables. In *4th International Conference on Foundations of Data Organization and Algorithms*, pages 279–300. Springer-Verlag, October 1993.

[MR93] M.C. Murphy and D. Rotem. Multiprocessor join scheduling. *IEEE Transactions on Knowledge and Data Engineering*, 5(2):322–38, Apr 1993.

[MSD93] M. Mehta, V. Soloviev, and D.J. Dewitt. Batch scheduling in parallel database systems. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 400–410, 1993.

[N+87] M. Nelson et al. The National Center for Atmospheric Research Mass Storage System. In *Digest of Papers, Eighth IEEE Symposium on Mass Storage Systems*, pages 12–20, May 1987.

[Ols92] Michael Allen Olson. Extending the POSTGRES database system to manage tertiary storage. Master's thesis, University of California, Berkeley, 1992.

[RFJ+93] M.F. Riley, J.J. Feenan Jr., et al. The design of multimedia object support in DEC Rdb. *Digital Technical Journal*, 5(2):50–64, 1993.

[SB+93] T. Stephenson, R. Braudes, et al. Mass storage systems for image management and distribution. In *Digest of Papers, Twelfth IEEE Symposium on Mass Storage Systems*, pages 233–240, Apr 1993.

[Sel93] P. Selinger. Predictions and challenges for database systems in the year 2000. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 667–675, 1993.

[SFGM93] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The sequoia 2000 storage benchmark. *SIGMOD Record*, 22(2):2–11, 1993.

[SLM93] B. Seeger, P. Larson, and R. McFadyen. Reading a set of disk pages. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 592–603, 1993.

[Smi81] A.J. Smith. Long term file migration: development and evaluation of algorithms. *Communications of the ACM*, 24(8):521–32, Aug 1981.

[Son89] Sony Corporation, Japan. *Writable Disk Drive WDD-600 and Writable Disk WDM-6DL0 Operating Instructions*, 1989. 3-751-047-21(1).

[Sto91] M. Stonebraker. Managing persistent objects in a multi-level store. *SIGMOD Record*, 20(2):2–11, 1991.

[Wie87] G. Wiederhold. *File organization for database design*. McGraw-Hill, New York, 1987.

[Won80] C.K. Wong. Minimizing expected head movement in two dimensional and one dimensional mass storage systems. *ACM Computing Surveys*, 12(2):167–78, Jun 1980.