Database Systems for Efficient Access to Tertiary Memory*

Sunita Sarawagi

Computer Science Division
University of California, Berkeley

Abstract

Tertiary storage devices have long been in use for storing massive amounts of data in file-oriented mass storage systems. However, their use in database systems is relatively new. Database systems associate more structure to the data than just raw sequence of bytes. Hence if they are allowed control of the tertiary memory devices, they can greatly reduce access cost by doing more informed caching, query optimization and query scheduling. However, most conventional database systems are designed for data stored on magnetic disks. Accesses to tertiary storage devices are slow and non-uniform compared to secondary storage devices. Therefore, inclusion of tertiary memory as an active part of the storage hierarchy requires a rethinking of conventional query processing techniques. In this project, our aim is to design a database system that can use its knowledge of the layout of data on storage devices to increase the speed of running queries.

1 Introduction

Applications manipulating large volumes of data are growing in number: earth observation systems [12], [13], historical data base systems, statistical data collections and image and video storage systems [10] are a few examples. For these applications, we need effective hardware and software solutions for storing, retrieving, and managing massive amounts of data. In spite of the falling price-line of magnetic disks, tertiary memory devices are still an order of magnitude cheaper than magnetic disks. This often makes them a viable alternative for storing data over a terabyte in size. To get cost-effective performance, the slow yet cheap tertiary storage devices are often used in conjunction with faster magnetic disks that act as caches for hot data.

As regards software alternatives, the first option is to directly use a mass storage file system interface like unitree (Figure 1a). Most file systems provide only a byte oriented interface to the underlying data. This means that the functionalities offered by a data management system are not available to the applications. The second easy option is to build a database management system on top of an existing file system (Figure 1b). Examples of systems of this are given in [4] and [2]. These systems use the underlying file system to get transparent access to data stored on tertiary memory and store only the metadata information in the database. For both these above options, the file system controls the movement of data to and from the disk cache and the tertiary memory. Typically, replacements policies like LRU and weighted LRU [9] are used for managing the disk cache. These policies have been known to be inefficient for conventional database systems and are more so for tertiary memory databases because of the higher I/O cost on tertiary storage devices. Database systems have more semantic information about the data and the kind of queries posed and hence allowing the database system to exercise control of the caching between the disk and tertiary memory can yield better I/O performance. This brings us to the third alternative where the disk cache is directly under the control of a database system (Figure 1c). POSTGRES [6] is one of the pioneer projects in this regard. POSTGRES includes a Sony optical jukebox as an additional level of the storage hierarchy. The POSTGRES storage manager can move data transparently between a disk cache and the jukebox using the 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. Inclusion of tertiary memory devices in the storage hierarchy requires a rethinking of many design decisions made for a conventional database system. Many database researchers [1, 11, 5, 8] have reached consensus regarding the need of a database system specially optimized for manipulating data stored on a tertiary memory device. In this paper, we will see how we modified the design of an existing

*This research was sponsored by the National Science Foundation under grant IRI-9107455, the Defense Advanced Research Projects Agency under grant T63-92-C-0007, and the Army Research Office under grant 91-G-0183.

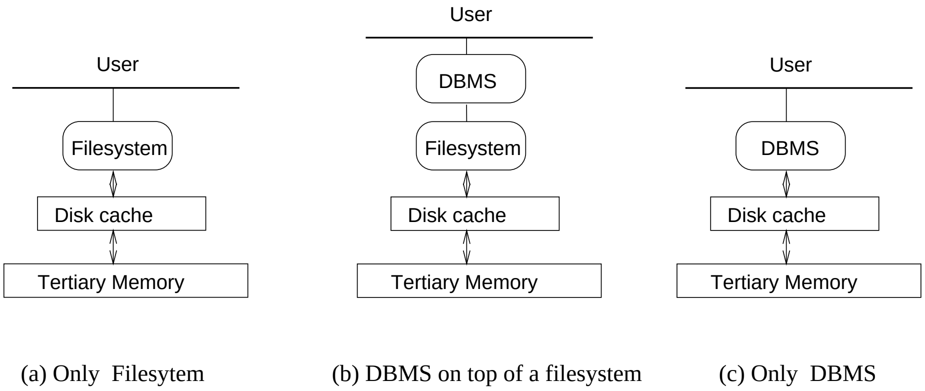

Figure 1: Available software alternatives.

database system to optimize for accessing data stored on tertiary memory.

Outline of the paper We will start by bringing out the difference in the characteristics of a tertiary memory devices and magnetic disks that makes it necessary to change conventional database systems. We will then argue how a conventional database system can lead to miserable performance on data stored on tertiary memory (Section 2). We will then present the design of our tertiary memory database and contrast it with a conventional database system (Section 3). In Section 4, we will present the extensions made to an existing relational database system namely POSTGRES to support the new architecture. In Section 5 we present future work and discuss alternative approaches. Finally, we make concluding remarks in Section 6.

2 Tertiary memory devices on conventional system

Tertiary memory devices have very different characteristics than 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 drives. The storage units are either tape cartridges or optical disk platters. We will use the term *media* to refer to a storage unit. 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 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 time (exchange time + full seek time + transfer time) and best case time (only transfer time) 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 when 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. This makes it crucial to carefully optimize the order in which data blocks are accessed on these devices. Most disk-based database management systems use the number of pages transferred from the disk to main memory as a measure of the I/O cost and the optimizer tries to reduce this cost. For tertiary memory systems it is very important to reduce the number of media loads/unloads and the seek overhead by suitably reordering the I/O. Reordering I/O is often more effective in improving overall performance than trying to eliminate I/O. As we will see next, this makes database systems that were originally designed for data stored on magnetic disks very inefficient for tertiary memory data.

Figure 2 shows a block diagram of POSTGRES, a typical conventional database system. POSTGRES uses a process-per-user architecture, in which a different server process is started for each user. These servers have a shared pool of pages in the magnetic disk cache. Except for locks controlling shared resources, each user process runs independently of others, requesting data from the disk cache as and when they need data from tertiary memory. If the data is not already cached, a request is sent to the tertiary memory device. Each server process can query data on arbitrary storage media, and there is no communication between the server processes regarding the order in which data pages are requested. Thus, the tertiary storage device might have to support a fairly random sequence of data requests. The device

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.

scheduler can do local reordering of I/O requests that are pending and avoid unnecessary media switches by batching requests on the same media together. However, the amount of reorganization that it can do is very limited because it has no idea about other data that are required by the currently running queries. Also, the server process has no idea about the storage media that are currently loaded or the pages that are in cache and does not modify its data requests according to the cache state or the tertiary memory state. Lack of global planning and coordination can thus lead to bad I/O performance on the tertiary memory device.

3 Our Approach

We follow an integrated approach to disk cache management, tertiary memory I/O scheduling and query processing (Figure 3). This is in contrast to conventional database systems described earlier where these tasks are handled by independent functional units. The system consists of a centralized scheduler that 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 cache (not just physical page addresses) and knows about all the queries in the system and what data they access. Instead of queries requesting data to be fetched from tertiary memory to the disk, the scheduler infers the data required by a query in advance, caches the data in the disk cache and then schedules the query for execution. The decisions made by the scheduler are aimed towards making the query execution process globally efficient. The scheduler uses its knowledge of the state of I/O system and the executing queries to judiciously batch I/Os from several queries and combine the execution of similar queries.

3.1 Query Processing in the new architecture

In general, a query accesses data that is spread arbitrarily across many storage media. We divide relations that spread over multiple storage media or are not placed contiguously on tape into *fragments*.

A fragment is the part of a relation that can be transferred without incurring additional media switches and seeks — thus fragmentation is a mechanism to expose the layout of a relation on tertiary memory. Data is fetched from disk to tertiary memory in units of fragments instead of whole relations. Hence the size of the fragment is important to performance. The best fragment size for a particular setup is a function of the average request size, the latency of first access and the transfer cost. When the fragment size is small, we make greater number of I/O requests and the latency of first access is incurred too many times. When the fragment size is large, the transfer overhead is higher because we might be transferring more data than we need. This is a standard problem that arises in all caching systems and it is easy to find rule of thumb values of the best fragment size based on estimates of the latency of first access (measured in terms of media switch cost and seek cost), the data transfer rate and the average size of data requests.

An arriving query is processed in multiple stages. In the first stage, the query on the base relation is broken down into independent subqueries on the base fragments. For instance, a join query between relation R and S where relation R consists of fragments $R1$, $R2$ and $R3$ is broken down into three join queries between fragment $R1$ and S, $R2$ and S, and $R3$ and S respectively. We fix the maximum size of the 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. However, optimizing each subquery separately this way can cause a lot of time to be spent in the optimization process, especially when

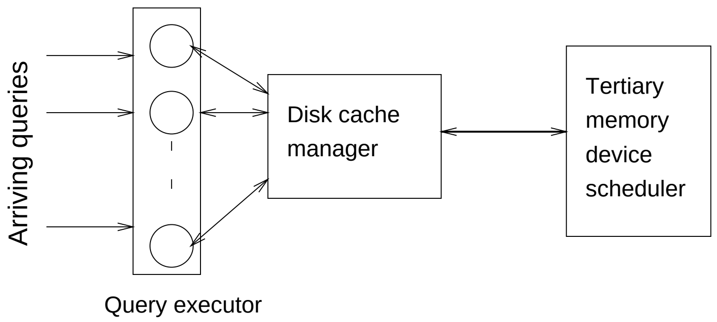

Figure 2: Architecture of a conventional database system.

the number of fragments is large. This problem can be easily removed by noting that most of the subqueries of a query are expected to be identical, so the optimizer can reuse the optimized plan for one subquery repeatedly for other subqueries with identical structure. There are some limitations to this model of breaking query into independent subqueries as described above. This will be discussed in Section 5.

At the end of this phase, we get a set of independent subqueries and the scheduler has to optimize the order in which fragments required by the query are fetched from tertiary memory.

3.2 Responsibilities of the Scheduler

The scheduler has a collection of such subqueries which are pending execution. The scheduler then decides the following:

1. What fragments are to be fetched next from tertiary memory to disk.
2. If the disk cache is full, what fragment should be evicted from the disk cache to make space for the newly fetched fragment
3. What subqueries on the cached fragments should be scheduled next for execution. When scheduling subqueries for execution it tries to combine subqueries that can be computed together. For instance, if there are two subqueries which do sequential scan on the same base relation, the scheduler combines the execution of such queries.

These decisions are guided by global consideration and are based on the state of the tertiary memory and the magnetic disk cache. We will next describe how the scheduler makes the decisions to fetch and evict fragments between the disk cache and the tertiary memory.

3.3 Fragment fetch and eviction policies

When deciding what fragment to fetch next the scheduler is faced with the following situation. There are a collection of pending subqueries each of which requires one or more fragments for processing next. Of the fragments required, some may already be in the disk cache and others may need to be fetched from tertiary memory. Of the fragments to be fetched from tertiary memory some may reside on storage media which are already loaded on the drives and others may be residing on the shelves. First, the scheduler has to decide whether it should choose a fragment from the currently loaded media or load a new media for fetching a fragment. Then, it has to decide what fragment it should evict from the disk cache to make space for the newly fetched fragment. These decision depends on a number of factors: the tertiary memory characteristics, the nature and number of pending queries and the size of the disk cache. Since finding the optimal solution is intractable, we studied a number of different heuristics which made intuitive sense. A simulation modeling various tertiary memory devices, cache sizes and query and database characteristics was implemented. The simulation helped us study the performance bottlenecks that arise for different tertiary memory devices and workloads, get an estimate of the best performance achievable and design good policies for fetching and evicting fragments. A qualitative summary of the effect of the three main factors (the tertiary memory characteristics, cache size and the workload) is presented next. More details on the simulator and quantitative results appear in [7].

Tertiary memory characteristics The important characteristics of the tertiary memory that influence the policies used for fragment fetch and eviction are

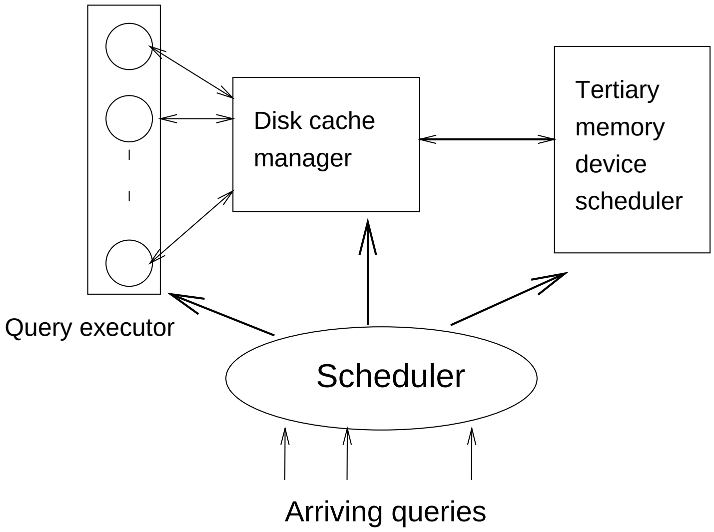

Figure 3: Proposed query processing architecture.

the number of drives, the date transfer rate, the media load/unload time, and the seek costs (the fixed seek overhead and the seek rate). Modeling the effect of the tertiary memory is complicated by the presence of a wide variety of tertiary storage products with widely varying performance characteristics. For some tertiary memory devices the data transfer bandwidth is low, so that reducing number of bytes transferred is more important. For others, the latency of first access is high making it more important to reduce the number of I/O requests. In Table 2 we present the performance characteristics of a few tertiary memory products to illustrate the wide variation in the cost metrics. This diversity in performance makes it important for the fragment movement policies to be a function of the tertiary memory characteristics and thus be applicable to a wide range of storage devices.

Workload Characteristics The sizes and number of objects in a database system can vary considerably. When a query requires fetching many small objects it is more important to use a policy that reduces latency of first access, i.e., the media load/unloads and the seek cost. On the other hand, if a query requires fetching of large objects, the transfer cost is the main bottleneck and it is important to employ policies that reduce the transfer cost. The number of queries currently pending defines the load level on the system and also affects the relative performance of various policies.

Size of the disk cache When the cache size is large compared to the total size of queried fragments, it is easy to optimize for I/O by fetching all the queried fragments on the currently loaded media before unloading it. When the cache is a limited resource, it becomes necessary to favor fetching of those fragments that join with cached fragments and thus complete all queries on the cached fragment even if the fragment will require an additional load/unload operation. This means that in order to avoid evicting fragments from cache which have queries on them we might have to unload media which contain fragments which have pending queries on them.

In designing a policy for caching fragments on the disk cache, our goal was to choose a policy that generalizes to diverse tertiary devices and application domains and is sensitive to changes in the load level and reference pattern of queries. To enforce this, we incorporate provisions in the database system to store information about the performance characteristics of the particular tertiary storage device used, the average size of database relations and the size of the disk cache available to the database system. The system monitors the current load and uses this information along with the stored information to decide on the policy to use for fragment fetch and eviction.

	Sony WORM	Exabyte EXB120	Metrum RSS-600	Sony DMS
classification	small optical jukebox	small tape library	large tape library	large tape library
switch time (sec)	8	171	58.1	39
transfer rate (MB/sec)	0.8	0.47	1.2	32
seek rate (MB/sec)	-	36.2	115	530
seek start (sec)	0.5	16	20	5.0
number of drives	2	4	5	2
platter size (GB)	3.27	5	14.5	41
number of platters	100	116	600	320
total capacity (GB)	327	580	8700	13120

Table 2: Tertiary Memory Parameters.

4 Implementation of the new architecture

We are extending the POSTGRES relational database system to implement a tertiary memory database system. In the old architecture all users first submit queries to a master process called the POSTMASTER which spawns off one POSTGRES process for each user session. In the new set up, queries are submitted first to a newly-implemented scheduler process. The scheduler maintains a number of slave POSTGRES processes. These processes are of two types:

- One set is 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 drive controllers present 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 this way also help hide some of the latency of media load/unload operation — when one drive is transferring data, the robot arm is free and can be employed for switching media on some other drive.
- A second set of slave backends is used for executing the subqueries on data that is cached on magnetic disks. Since these processes work concurrently with the data-transfer processes I/O on tertiary memory can be effectively overlapped with query execution. The number of backends employed for running queries is fixed based on the number of users submitting queries to the scheduler.

Implementation Status Two tertiary memory devices, a Sony optical jukebox and an HP magneto-optical jukebox have already been interfaced with POSTGRES. Initial estimates with the prototype yield dramatically faster performance compared to processing queries using conventional techniques.

5 Future Work

5.1 Extending the query processing architecture

The architecture described so far has a number of limitations when we try to use it for general query processing. These problems along with the proposed solutions are listed below.

- Redundant processing: 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. E.g., for a hash join, if the probe relation is broken into n fragments, then for each probe fragment the hash table has to be built n times. To reduce this overhead, when the scheduler tries to select a subquery from the disk cache for execution, it tries to combine subqueries that share computation. This will be treated as a part of the general multiple query optimization to be handled by the scheduler.
- Result ordering: For some queries the order of the result tuples is important and executing subqueries independently is not possible. One solution to this problem is to partition the query into subqueries based on ranges of values chosen to fit in the disk cache and execute these queries one after another. The other approach is to use the earlier method of executing as independent subqueries but store the result of each subquery in a temporary file to be sorted at the end. These two approaches, along with others, will

be evaluated for their efficacy in running ordered queries.

- Unknown data requirement: For some query processing methods it is not possible to know all the fragments required by a query in advance. Such queries need to be executed in multiple phases. For instance, when doing a select on a relation with one of the attributes a large object (stored as another fragment), we first 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. Similarly, for unclustered index scans with low selectivity, it might be very wasteful to fetch all the fragments of the relation before scanning the index tree(s). This problem can again be solved by two-phase execution. In the first phase, the scheduler is asked to fetch only the index tree(s). In the second phase, the index tree(s) are scanned and only fragments found to contain qualifying tuples are scheduled for fetching. Another solution to the unclustered index scan problem is to maintain two-tier indexing as suggested in [3].

5.2 Evaluating alternative approaches

So far we have assumed that the database system has full control of the tertiary memory and is the only application using it. This may not be true in many existing systems. Managing massive storage systems is expensive; hence these systems are usually shared by many applications. We intend to evaluate the applicability of the optimization techniques when a database system is just one of the clients of the file system managing the tertiary memory instead of the sole user of it. Getting good performance in such a configuration may still be possible, but the file system has to be extended to provide some additional facilities and hooks used by the database system. Some of the features that are desirable for enabling efficient operation of the database system are listed below.

- The database system should be able to know the layout of its relation on tertiary memory — for instance the file system should be able to tell the database what part of a relation resides on what storage media and if the storage media is tape, what is the seek overhead involved in accessing the first byte of the fragment.
- The storage system should be able to provide estimates of a few cost parameters of the tertiary storage device e.g., the time needed to load/unload a storage unit, the data transfer rate and the seek cost.
- The database system should be able to reserve a few drives for its own use and should be able to dictate what storage unit should be loaded next and the duration for which it should be loaded.
- The database system should be able to control what fragments should be evicted from the disk cache and should be able to infer the total space available to it for caching on the disk cache.

6 Conclusions

In this paper we first presented the limitations of the current options available to a user for accessing data stored on a tertiary storage system. We then presented the design of a database system that is optimized for accesses to a tertiary storage system. The main features of the system 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. The scheduler then submits queries for execution and controls the caching from the disk to the tertiary memory in a globally optimal fashion.
- We used the notion of a *fragment* to reveal the layout of the relation on tertiary memory to the query optimization and 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 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. These policies import a few model parameters for each tertiary memory device and are thus designed to be portable across a wide variety of tertiary memory devices.

We are in the process of extending the POSTGRES database system to handle the new cache management and query optimization strategies. Our next project is to extend the model so as to handle multi-way joins and sort-merge joins and design the multiple query optimizer. Finally, although the simulation has given us useful insights into performance optimization of tertiary memory, we would like to measure the performance of this new architecture on real-life workloads.

Acknowledgement

I would like to thank my advisor Mike Stonebraker for helping me with the initial design of the system and suggesting this topic. I would like to thank Soumen Chakrabarti for greatly improving the presentation of this paper.

References

[1] 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.

[2] Jr. Carino, F. and P. Kostamaa. Exegesis of DBC/1012 and P-90: industrial supercomputer database machines. In *Proceedings of the fourth International PARLE Conference*, pages 877–92, Jun 1992.

[3] D. Choy and C. Mohan. Locking protocols for two-tier indexing for partitioned data. Technical Report IBM Research Report, IBM Almaden Research Center, Jun 1993.

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

[5] 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.

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

[7] S. Sarawagi. Query processing and caching in tertiary memory databases. Technical report, University of California at Berkeley, 1995. in preparation.

[8] 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.

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

[10] 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.

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

[12] Michael Stonebraker. An overview of the Sequoia 2000 project. Technical Report 91/5, University of California at Berkeley, 1991.

[13] Michael Stonebraker and Jeff Dozier. Large capacity object servers to support global change research. Technical Report 91/1, University of California at Berkeley, 1991.