Tapes Hold Data, Too: Challenges of Tuples on Tertiary Store

Michael J. Carey* Laura M. Haas† Miron Livny*

1 Introduction

Enormous quantities of data are being accumulated by both the commercial and scientific communities. In the hope that analyzing past activities will help them improve their business (e.g., profile sales by customer groups and products for the last 24 months), many commercial enterprises keep a record of every customer transaction they have ever performed, typically in some DBMS. Yet, the total volume of data generated by such an enterprise is in most cases larger than disks can (affordably) accommodate. Thus, the commercial community is increasingly driven to store data offline on less expensive media. Tertiary stores offer virtually unlimited capacity, at a lower price and with a smaller footprint than disk. Since today most DBMSs only operate on disk resident data, data on tertiary store can no longer be queried without loading it back onto disk.

Tertiary devices are routinely used by the scientific community to store vast amounts of measured and derived data. Typically, the data is stored as files and is not under the control of a DBMS. Many scientific applications, such as the global change studies that the Earth Observing System (EOS) and Project Sequoia [15] will enable, require advanced data analysis capabilities, and would like to use a DBMS for relating and tracking the data. Unfortunately, aside from metadata management, today's DBMSs have little to offer these applications. They can neither handle the volume of data required by the applications, nor can they access the media on which the data is (and due to price and volume, will continue to be) stored.

Tertiary devices have long been important to the commercial and scientific communities, but from a DBMS prospective, tapes and optical disks are second class citizens compared to magnetic disks and main memory. In light of the increasing need for DBMS controlled tertiary storage systems [12], our challenge to the database community is to accept tertiary devices as first class storage devices. Only then will we be able to extend the benefits of DBMSs to cover the commercial and scientific data that is found on tertiary stores. The goal is to provide both the DBMS applications that are overflowing their disks and the scientific applications that are longing for DBMS control of their data with location transparent access to data, with system-controlled placement and migration of data, and with declarative associative access to data. End users and application writers should not need to know where or how their data is stored for correct function of their queries, but should have a mechanism to communicate with the system in order to understand or influence the system's performance. The DBMS should spare the DBA most of the pain involved today in administering a large hierarchy of devices. It should automatically cache and migrate data, and should provide tools for the DBA to understand and influence its actions.

This goal will not be easy to achieve. Tertiary devices cannot be treated as just another disk system. They do not behave like disks. Their performance characteristics vary widely. The data they store is not constantly on-line in the drive that can read it. Not all tertiary devices are even random access! To meet our goal we must create a framework that will let us analyze and compare different tertiary storage systems. We must understand the roles that different types of devices can play in the system, and how to use their resources effectively for different kinds of data processing, under different sorts of workloads. In addition to the framework, we need new mechanisms to deal with these devices. For example, we must adjust our query processing techniques, developed based on properties of disks, to the vagaries of tertiary store. Finally, we must develop policies for controlling the system that will promote good performance. As an example, we must decide when and where to allow caching of data.

This paper is organized as follows. In the next section, we present a framework for discussing database controlled tertiary storage systems. Most work on handling tertiary stores to date has been in the context of archival Mass Storage Systems [1, 2]. The only work of which we are aware in which the DBMS controls a tertiary device is that of Stonebraker [14] for the Sequoia project. We discuss these systems in more detail in section 3. In section 4 we take a more detailed look at the challenges we see, while in section 5 we discuss our plans for research in this area.

*Computer Science Dept., University of Wisconsin, Madison, WI 53706

†IBM Almaden Research Center, K55/801, San Jose, CA 95120

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0413...$1.50

2 A Framework

In this section we propose a framework for discussing database controlled tertiary storage systems. The framework consists of a hardware component, a software component and a cost component. The range of I/O devices controlled by the system is captured by the hardware component. The policies and mechanisms employed by the DBMS to manage the data and to control these devices is captured by the software component. The cost component captures the parameters needed to characterize system performance. The framework should be rich enough to compare existing and proposed solutions; we consider the following a "rough stab", and we invite the database community to help us refine and elaborate it further.

We adopt a very general view of the hardware for a database controlled tertiary store. The DBMS runs on one or more processors, to which are attached several storage devices, including, but not limited to, disks and main memory. These storage devices are typically viewed as a pyramid, with small amounts of expensive store (e.g., main memory) at the top, and increasingly larger amounts of cheaper store at successive levels underneath. We will refer to those devices falling below disk in the pyramid as tertiary devices. Note that data stored on a tertiary device need *not* travel through all higher levels of the pyramid before eventually reaching main memory. This is different from the classical pyramid in which, for example, several layers of caching store (extended memory, solid state disk) separate disk from memory, and a data item not found in memory is looked for in the next layer down, and then the layer below that, and so on. In our framework, how data travels and where it is looked for are policy decisions.

Different classes of tertiary devices may be attached to the same DBMS. An optical disk library and a magnetic tape library represent two such classes and might both serve the same system. Within a class the framework distinguishes between different device types. For example, tape drives may differ in their technology and in the characteristics of the tapes that they serve. There are several different tape technologies, including helical scan and conventional linear recording, and within a technology, tapes may come in several different widths with different density characteristics (e.g., 4mm DAT vs. 8mm video vs. 19mm D-2). They may also vary widely in capacity, data rate, and search speed. Conventional 1/2" tapes hold 200MB of data, and the drives are capable of a 3 MB/sec transfer rate, with a slightly higher search speed. By contrast, an 8mm tape can hold 5 gigabytes, searches at up to 37.5 MB/sec, but transfers data at only 500 KB/sec. As in the case of device classes, a single DBMS may control a heterogeneous collection of device types.

The software component of the framework consists of a storage architecture and a set of data handling capabilities. We define a system's *storage architecture* as the answers to a series of questions defining the basic mechanisms and policies of the system. There are two basic mechanisms: *migrating* data (placing data in a new, "permanent" location) and *caching* data (making a temporary copy of data). The policies that determine how these mechanisms are used can be classified as data storage, data movement, and device management policies.

Data storage policies specify where data can be stored, that is, which devices can be "home" for data items, and where data can be cached. For example, some systems might allow data to be stored at any level in the hierarchy, while others might store data only on tertiary store, allowing selected items to be cached upward. The *placement* policy is another data storage policy that decides where a given data item will be stored, if multiple devices can store data. Data placement could be by size, by age, by last reference, by predicate (as in [14]), and so on.

There are several types of *data movement*. In addition to migration and caching, some systems may also want to distinguish *archiving*, which makes a copy for backup purposes. For each type of movement, there are policies that describe when that type of movement is triggered, what path (if any) the data follows, and how the data to be moved is selected. Migration might be triggered by the system, for example, when a device gets too crowded, or it might be done periodically, to maintain a certain occupancy, or it may require an explicit user command, or occur automatically when a cached item is updated, and so on. Data may always migrate along a particular path (e.g., from tape to disk to memory), or it may be allowed to skip levels (e.g., go straight from tape to memory), or travel unconstrained (from tape to optical disk, perhaps). If the system decides to migrate data because, for example, a device is too full, it could select the data to migrate based on its reference history, its size, or some other factor.

Finally, there must be *device management policies* to specify how space is managed on each device. For example, how much of the device is used for caching versus data storage? Is it a fixed percentage, or does it vary? If a device is used to cache data from multiple devices, how is space in the cache allocated among these different devices?

We must also be able to describe a system's *data handling capabilities*, or how the system fulfills traditional DBMS responsibilities. Again, we can point to the available mechanisms, and a closely related set of policies. Chief among the mechanisms of any system are the execution strategies available for the various operations. Other choices of mechanism would be the set of costs that are modeled, and whether optimization occurs at compile or run time. Important policy decisions include: can the system operate directly on data on tertiary store, or must it first be brought to disk? and: what decisions can the optimizer make?

Due to the range of devices and their diverse characteristics, the cost component of the framework should be more detailed than those typically used in modeling database algorithms today. For example, it has been argued [7] that disk-based join algorithms can be compared simply by the number of I/O requests that they make. In this more diverse world, though, the number of I/O requests may be most important for one device, while minimizing the number of bytes transferred may be paramount for another. Thus the various different costs involved in retrieving data from a device should all be modeled separately. These costs include start time (latency), data transfer rate, search rate, exchange time (the time to return one media unit to storage, and fetch the next to the drive), load time (the time to move the media

unit into the drive), capacity/unit, and data block size for each device.

This was just a brief summary of the framework that we propose, and that underlies the remainder of this paper. Existing frameworks are not adequate for the task of designing a database controlled tertiary store. The IEEE Reference Model for Mass Storage Systems [3, 10] deals with some storage architecture concerns, but its focus is more on the software structure of mass storage systems, and less on how they accomplish the various functions or what policies they enforce. It does not cover any data handling capabilities. The database literature provides lists of expected data handling abilities, but rarely deals explicitly with storage architecture policies. Thus we feel that this new framework is necessary, and should be elaborated and refined as a basis for discussion of these new systems.

3 Existing Systems

Most work on handling tertiary storage devices has been done in the context of file-oriented Mass Storage Systems [1, 2]. These systems have typically been developed for use in scientific supercomputing contexts. They are for the most part distributed systems consisting of a large mainframe server, a supercomputer (compute engine), and multiple clients. All of these generally have their own disks. The server is additionally attached to the tertiary storage device, which is most frequently a tape library. Mass Storage Systems generally have a well-defined storage architecture. Data files are typically stored on tape, and cached on disk at the server and/or client(s). Caching may be triggered by explicit user request, or, in more sophisticated systems, automatically on reference (or on the nth reference in some time frame). When space is needed on disk, the copies to be freed are typically chosen by a function based on size and frequency of reference. Some systems initially store files on disk, then later migrate them to tertiary (or make an archival copy that later becomes the permanent store). Interesting systems include LSS [6, 8] from Lawrence Livermore Labs, NASA's MSS-II [17], Los Alamos National Laboratory's CFS [5], the National Center for Atmospheric Research's MSS [16], and Epoch's InfiniteStorage Architecture [9]. These systems generally have only primitive data handling capabilities. The unit of movement is the file, and the only operations are store and retrieve file. Data must be brought to disk before it can be used.

Stonebraker [14] was the first to propose extending database technology to a hierarchy of storage devices. POSTGRES [13] has been extended to allow data to be stored in an optical disk library [11]. POSTGRES' storage architecture allows data to be stored on any device. The storage location is set during data definition, based on a user-supplied predicate or by user-identified field. The user must explicitly partition the tables, then reference them through views. POSTGRES provides two unrelated types of caching. The *ELEVATE* command allows the user to make an explicit copy of some portion of a relation. In addition, the components that manage individual devices (called device managers) may have their own internal cache. For example, the POSTGRES device manager for the optical disk library automatically caches referenced data on disk. Either type of cache may be freed by the system when space is needed; explicit copies may also be freed by the *RETURN* command. Migration is always user-driven, though once the command is issued, movement may be done asynchronously over time. POSTGRES is a full-fledged DBMS, providing typical DBMS data handling. Again, it seems that data must be brought to disk to be used; the optimizer is unaware of caching, however, and cannot control the movement of data to disk. The optimizer does model device-specific costs (i.e., the cost of getting the data off of the tertiary device). There is no mention of new query execution strategies or access methods. In summary, POSTGRES has done groundbreaking work in a new and important direction. However, it falls short of the vision outlined in section 1, as it requires manual placement and migration of data by the user or DBA. Better performance might also be achievable with optimizer control of data movement and knowledge of data caching.

4 Challenges

The overall challenge is to determine a set of policies and mechanisms (e.g., those defined by the framework) that will result in a DBMS controlled tertiary store with the properties outlined in section 1. Individual DBMS challenges can be grouped into three rough categories that we elaborate on further in this section: *understanding the basics*, namely, tertiary technology and the applications that would use the DBMS controlled tertiary store, *determining how best to process queries*, and *determining policies for storage management*. Another challenge that is clearly important is *system configuration*: how does the DBA determine what set of devices is needed, what capacity they should have, how many drives are needed, and so on? However, since this is not a DBMS challenge *per se*, we will not elaborate on it further.

4.1 Understanding the Basics

To make any design decisions, we must understand the technology that we are using. What are the right set of parameters to characterize such a broad range of devices? What are the parameter values for various real devices that exemplify the different technologies? It is important to get these values by measurement, and not just rely on manufacturers' specifications, as devices may behave very differently on real tasks than they do in the laboratory. It is also important to observe these devices under the sort of operations likely to be generated by a DBMS, which may be quite different than those produced by a file system. For example, whereas a file system's typical operation sequence might be load tape, read a file, replace tape, a DBMS joining two large relations might need to load two tapes, read some blocks from one, then some from another, then read from the first, and so on. How will start/stop times affect performance for this sequence? What will be the effect of exchange times? What search speeds will actually be observed on DBMS-like searches? This type of work is fundamental to progress towards our goal. In addition, our observations may allow us

to influence the next generation of tertiary storage devices.

Equally important, we must understand the applications that will exploit our new system. What types of operations will they do? What types of analyses? Understanding these will help us interpret the results of the technology work, by teaching us which characteristics of the devices are important. Are lots of cartridge replacements likely to be needed? Platter flips? Searches? We also need to understand why a given application has such a huge data volume. This could be due to a huge number of moderately sized (i.e., individually disk-managable) data sets, for example, the results from many scientific experiments. On the other hand, the large volume might come from a moderate number of huge data sets, each with moderately sized records. This is typically the case when commercial enterprises store several years of customer data for data mining purposes. Finally, it might result from a moderate number of moderate data sets, each of which has huge data items, for example, medical image data or earth sciences images gathered from EOS-like devices. The reason for the application's large data size will likely influence, if not outright dictate, the techniques that are needed for data handling for tertiary-sized DBMSs.

4.2 Query Processing

There are three key challenges in determining how best to process queries: Index Structures, Join Algorithms and Query Optimization.

Disk-based indices are designed based on the random-access characteristics and cost functions of disks. These are quite different from those of many tertiary storage devices. Various storage and index structures have been designed for WORM optical disks, but these disks are still very different than tape, for example. Thus, existing structures and algorithms may prove helpful as starting points, but are probably not solutions by themselves. Among the issues we need to address are questions of location (where should the index reside?), scale (when indexing a veritable sea of data, how big will the index be?), and structures for "barely random" access devices such as tape robots. Should (can) indexes reside on disk, or will they be on tertiary store, or split somehow between the two? What index structures will interact well with tape robots and their ilk? What granularity will be good for index nodes, and how should the search (and update, if any) algorithms work?

Given that tertiary devices have very different performance tradeoffs and characteristics, the existing "best of class" algorithms for join processing (e.g., hash join, sort-merge join) may perform extremely poorly. For example, hash joins involve random I/Os to flush bucket tails as they fill while partitioning the relations (or else when buckets are read back during the join phase). These random I/Os will kill performance if the source relations' size and location necessitate the use of tape for the intermediate results. A similar argument will eliminate sort-merge joins because of the random I/O needed to merge the sorted subruns. As a result, new algorithms are needed. A variety of cases should be considered, including joins where indices but not relations fit on disk, joins where no indices exist, but key/pointer pairs would fit on disk, and joins where one, but not both, relations fit on disk, requiring the system to do clever movement of data. Minimization of seeks, or maximization of sequential access, will be very important, clearly; sorting (vs. hashing) may prove beneficial for key/pointer pair algorithms.

The query optimizer will need to incorporate the new access methods and join algorithms we develop. To do this, it will need new cost formulas that incorporate device specific costs. In this environment it will also be important for the optimizer to be aware of, and perhaps to some extent control, caching. It must at least be aware of what is cached and where. One major opportunity is to do multiple query optimization, to get as much out of one scan over a given tape or optical disk as possible [12]. A more advanced challenge is how to allocate resources to competing tertiary queries if these must be run simultaneously. These resources include disk space, memory, space and drives on tertiary store, bandwidth, and so on. Unfortunately, similar problems of balancing multiple queries and transactions even in today's simpler two-level hierarchies are still poorly understood [4].

4.3 Storage Architecture Policies

The major challenge here is to determine an appropriate set of storage architecture policies, including those for data storage and those for data movement. Data storage challenges include *data placement* (where to put the data in the hierarchy) and *data tracking* (how to avoid losing data in the hierarchy). Data movement challenges involve data migration, caching and prefetching.

In an ideal world, data placement would be trivial for the user/DBA. We would simply hand the data to the DBMS, ignoring the existence of the hierarchy, and the system would decide how to partition the relations, and where to store each partition. In practice, this is likely to prove extremely hard. It is not clear what criteria the system should use, or how it should gather and use the necessary workload and access statistics. We could instead limit ourselves to a conceptually simple storage architecture in which all data is stored on a single tertiary device, with other devices viewed simply as caches. This would simplify the placement problem, but may lead to inefficient query processing.

Alternatively, we might require the data definer (user, application programmer or DBA) to provide input on where data should reside. The challenge then becomes deciding what view the data definer should have of the data placement problem that will make it manageable. What will the granularity of placement be? How specific must the data definer be about where the data should be stored? (e.g., must (s)he specify the particular tape, or only which library?)

Once we have the data stored, we must keep track of it. This is also likely to be a challenging task. We must decide what metadata needs to be kept; for example, what statistics are needed in order to make migration and caching decisions? We must decide on the appropriate granularity for tracking data location (this is intimately connected to the granularity used for data placement). Finally, we must decide where the metadata should be stored. Will it fit on disk, or will it have to be at least partially stored on tertiary store? How should the metadata be organized so that both

the system and its users and administrators can locate data?

If the DBMS is responsible for data placement to some degree, then it may decide to migrate data from place to place (e.g., from tertiary to secondary store or vice versa). Clearly, there must be some policies on how, why and when data should be migrated. What policies will be most effective? What statistics must be kept to enforce them? What is the granularity of data that will be migrated?

No matter who is responsible for data placement, data caching will be crucial for effective tertiary storage management. Key questions include: How should the memory and disk caches be sized? For example, how much of the disk should be set aside for caching versus storage of disk-resident data? How should the hierarchy be structured: as a tree, or as a graph? For example, should tertiary data always go through disk on the way to memory, or should memory hold tertiary as well as disk pages? What should the unit(s) of caching be? What policies should be used in deciding what to cache and what to replace when the cache fills?

Prefetching is possible when we have some advance knowledge of what data will be needed when. It is most useful when the data is not likely to have been recently referenced. For example, consider a medical records application, where a patient has a doctor's appointment scheduled in advance. Prefetching could be extremely useful in such cases. If prefetching is used, it could be based on explicit hint calls from the application. For example, the application might run a program each evening to issue hints about tomorrow's appointments. Or, it could be based on rules defined by the application, for example, a rule that tells the system how to automatically use the appointments relation to prefetch data. Finally, a fully automatic solution would be to have the system somehow detect access patterns and correlations, and then do appropriate prefetching. This would be extremely hard, however, and is unlikely to be as effective as simpler mechanisms. It should be noted that prefetching also provides opportunities for doing batch processing and multiple query optimization.

5 Our Research Plans

We have sketched many challenges we see on the way to reaching the goal of a database controlled tertiary store. We feel there are three basic steps to a complete solution for this problem. First, we must understand the tertiary devices we are using, and the applications that will use a DBMS controlled tertiary store. Second, we must determine how to maximize the performance of individual operations. To do this we must discover the best access methods, execution strategies, etc. for each device type. We will also need to determine how best to exploit the different devices we have in the execution of each operation. For example, assuming the data is all stored on one tertiary device, how should we use other devices such as disk? How can we handle multiple levels of cache? Third, we need to look at global system level issues, such as data storage and data movement through the system, and balancing resources across multiple operations.

We are currently working on various aspects of the join problem in order to gain an understanding of the key parameters, and to get a better feel for the technology. We hope that we will learn which techniques to use under which circumstances to successfully cope with large volumes of data. Once we have an understanding of the basics, we plan to tackle some of the system level issues, for example, the open problems related to caching, migration, multiple queries, and so on.

We expect this effort to be challenging and rewarding, and we invite the community at large to join us in this research. We also invite potential customers to let the database research community know their needs, so that we can make reasonable assumptions about data types, data sizes, workloads, and so on. This would help to eliminate the (undesirable) challenge of simultaneously trying to predict the needs of the future while attempting to meet those future needs.

References

[1] *Digest of Papers, Ninth IEEE Symp. on Mass Storage Systems*, Oct. 1988, Monterey.

[2] *Digest of Papers, Tenth IEEE Symp. on Mass Storage Systems*, May 1990, Monterey.

[3] Special section on the "Mass Storage Reference Model – Special Topics". In [1].

[4] K. Brown et al., "Resource Allocation and Scheduling for Mixed Database Workloads". Tech. Rep. 1095, Univ. of Wisconsin, Madison, July 1992.

[5] B. Collins, M. Devaney, and D. Kitts, "Profiles in Mass Storage: A Tale of Two Systems". In [1].

[6] J. Foglesong et al.,"The Livermore Distributed Storage System: Implementation and Experiences". In [2].

[7] R. Hagmann, "An Observation on Database Buffering Performance Metrics". *Proc. 12th VLDB*, Aug, 1986, Kyoto, pp. 289-293.

[8] C. Hogan et al., "The Livermore Distributed Storage System: Requirements and Overview". In [2].

[9] G.G. Kenley, "An Architecture for a Transparent Networked Mass Storage System". In [2].

[10] S. Miller, "IEEE Reference Model for Mass Storage Systems". *Advances in Computers*, Vol 27, 1988.

[11] M.A. Olson, "Extending the POSTGRES Database System to Manage Tertiary Storage". Master's thesis, Univ. of California, Berkeley, July 1992.

[12] A. Silberschatz et al., eds, "Database Systems: Achievements and Opportunities". *Comm. of the ACM*, 34:10, Oct. 1991.

[13] M. Stonebraker and L. Rowe, "The Design of POSTGRES". *Proc. SIGMOD*, May 1986, Washington, D.C.

[14] M. Stonebraker, "Managing Persistent Objects in a Multi-Level Store". *Proc. SIGMOD*, May 1991, Denver, pp 2-11.

[15] M. Stonebraker and J. Dozier. "Sequoia 2000: Large capacity object servers to support global change research". Technical Report Seqouia 2000 91/1, Univ. of California, Berkeley, March 1992.

[16] E. Thanhardt and G. Harano, "File Migration in the NCAR Mass Storage System". In [1].

[17] D.Tweten, "Hiding Mass Storage Under UNIX: NASA's MSS-II Architecture". In [2].