# MANAGING PERSISTENT OBJECTS IN A MULTI-LEVEL STORE

*Michael Stonebraker*

*Computer Science Division, EECS Department*
*University of California*
*Berkeley, Ca. 94720*

## Abstract

This paper presents an architecture for a persistent object store in which multi-level storage is explicitly included. Traditionally, DBMSs have assumed that all accessible data resides on magnetic disk, and recently several researchers have begun to consider the possibility that significant amounts of data will occupy space in a main memory cache. We feel that future object managers will be called on to manage very large object bases in which time critical objects reside in main memory, other objects are disk resident, and the remainder occupy tertiary memory. Moreover, it is possible that more than three levels will be present, and that some of these levels will be on remote hardware. This paper contains an architectural proposal addressing these needs along with a sketch of the required query optimizer.

## 1. INTRODUCTION

Traditionally DBMSs have assumed that all data resided on magnetic disk. Therefore, all optimization decisions were oriented toward disk technology. For example, access methods have been proposed that are efficient on disk devices, e.g. B-trees and R-trees. Query processing strategies have been designed that work well for disks, e.g. merge-sort [SELI79], and query optimizers have been architected with a disk environment in mind [SELI79].

Recently, there has been some work on including in this traditional environment the possibility that significant portions of a data base may reside in main memory. Query processing strategies have been designed which require large amounts of main memory to be effective, e.g hash joins [DEWI84, DEWI86]. In addition, access methods appropriate to main memory have been constructed, e.g. T-trees [LEHM86] and the possibility of using AVL trees was evaluated in [DEWI84]. Lastly, query optimizers have begun to take more careful note of available main memory when constructing query plans [HONG90].

However, current general purpose DBMSs are still too slow to manage the real-time data bases associated, for example, with factory floor applications or telephone switching. Such applications are addressed by special purpose system software such as VAX-Elan and Rdb-Elan. It would clearly be desirable to expand the set of applications amenable to a general purpose DBMS solution by increasing its support for main memory data.

We also believe that most data bases will expand dramatically in size, requiring the inclusion of tertiary storage. In addition, it is possible that there will be multiple tertiary stores, perhaps at remote locations. Moreover, it is not unreasonable that there may be more than three levels in future systems. Therefore, in Section 2 we present a proposal for a multi-level storage architecture. The special needs of long fields are covered in Section 3, and then we move in Section 4 to an outline of the query optimizer needed in this environment. We conclude in Sections 5 with our prototyping plans.

In the remainder of the paper we make several assumptions. First, we assume that an abstract data type facility is available [STON86B, STON90]. Hence, a user can define new data types, functions and operators. Moreover, such types, functions, and operators are automatically available in the query language supported by the DBMS. For the purposes of this discussion, we assume the ADT facility available in POSTGRES and the query language, POSTQUEL [STON86]. However, any comparable capabilities could be readily substituted.

We also assume that the storage manager uses a no-overwrite philosophy as in [STON87]. Therefore, certain techniques that we propose take advantage of this property. Anyone interested in a Write Ahead Log (WAL) storage manager must make minor adjustments to our proposal.

## 2. A MULTI-LEVEL STORAGE MANAGER

We assume that the storage system consists of a collection of L **logical** devices that form a rooted tree. Hence, there is a unique root, called main memory, with zero or more direct descendant devices, each of which can have zero or more descendants. Moreover, these L devices can be on various computer systems in a network. For the moment we will assume that L = 3 and denote the devices by main memory, disk and archive, which are assumed to be on a single computer system. The extension of our proposal to L > 3 devices and to a distributed environment is discussed in [STON91].

A multi-level storage manager must be able to address the needs of the following clients:

1) real time applications which need sub-millisecond response times for requests to a main memory data base along with conventional response times to disk based data. Persistent programming languages are an example of this class of applications.

2) applications with mamouth data bases which need conventional response times to disk based data and reasonable response times to archival data.

To address these needs, one could either use a **physical** hierarchy or a **logical** hierarchy. We first discuss this issue and then turn to our specific proposal.

### 2.1. Physical or Logical Hierarchy

The traditional way of viewing a multi-level store would be to generalize the **physical block** model used by current disk-based systems. Current DBMSs and file systems assume that data blocks reside on disk and that **worthy** blocks are placed in main memory. Movement of blocks between main memory and disk is controlled by the buffer manager, which utilizes a replacement policy based on how recently each block has been touched and perhaps other semantic information [CHOU85]. The generalization to a three level store is presented in column 3 of Figure 1, where data blocks reside on the archive, worthy blocks are placed on the disk and very worthy blocks are placed in main memory. Storage blocks would be moved in the hierarchy as access patterns change by a generalized buffer manager. Some work on migration of whole files in such a three level store has been done in [SMIT81].

Although a physical block model is appealing because of its simplicity, we feel that it is inadequate for the following reasons:

1) Persistent programming languages need an object cache in main memory with the property that pointers to other objects are "swizzled". Specifically, a pointer to an object should be represented as a unique identifier (UID) on disk but as a physical pointer in main memory. Another simple example of an object changing representation when it moves between levels is a variable length character string. It might be stored in main memory as a pointer to a location in a heap containing a null terminated string. However, on disk the string would be represented by a length designator followed by the data. The physical block model is inadequate because it does not support an object changing representation when it move between levels.

2) A conventional relational DBMS maintains a main memory cache of system catalog objects (e.g. open tables, scan positions, etc.). In all systems we are familiar with, this cache is managed as a separate main memory data base. Moreover, system catalog objects have different representations in main memory and disk. Again, objects must change representation when a table is "opened" or "closed".

3) An application specific compression algorithm should be applied to images when they are placed on disk or archive. Moreover, when an image is fetched, it will sometimes be appropriate to decode it (for example to display it) and sometimes to leave it encoded (for example if the function to be applied to the image can use the encoded format). Moreover, it is possible that multiple levels of compression will be appropriate. For example, one should spend a small number of seconds compressing an image when it is sent to the disk and a larger amount of time when it is dispatched to the archive. Again, multiple representation changes may be required.

4) Objects are usually accessed through secondary (or primary) indexes. Such indexes can use direct physical pointers for main memory objects, but must employ UIDs for disk based data. Moreover, an AVL tree is an effective indexing scheme for main memory data, but fails disasterously on disk-based data, and a B-tree should be used instead. Hence, both the representation of an index and its basic algorithm should change when objects move in the storage hierarchy.

5) The degree of security controlling access to an object may be different for the different storage levels. One reason to place data in main memory is to ensure the highest possible performance. However, a traditional DBMS runs in a different address space from the application program. Hence, objects are not directly accessible to an application; rather commands must be sent over an interprocess message system and the result returned in the same way. This overhead is expensive for main memory data, and one might choose to allow the application program direct access to the main memory data. Consequently, the physical mechanisms used for access control to objects may vary from level to level in the storage hierarchy.

Because the physical block model does not support data objects or indexes changing representations or degree of security when they move between levels, we propose a more general **logical** model to rectify these deficiencies, as indicated in the rest of Figure 1. Here, the right-most column indicates the archive data base is the physical block model discussed earlier and supports caching of worthy blocks at higher levels in the storage hierarchy. However, the physical block model is generalized by supporting additional columns in the table. Each column corresponds to a different representation for an object which is appropriate for a specific storage device. Hence,

| location | main memory data base | disk data base | archive data base |
|---|---|---|---|
| main memory | main memory objects in main memory format (M) | cache of disk blocks | cache of archive blocks |
| disk | | disk objects in disk format (D) | cache of archive blocks |
| archive | | | archive objects in archive format (A) |

Logical Model of a Three Level Store
Figure 1

there are three representations possible in Figure 3, one appropriate for the archive (A), one appropriate for the disk (D), and one appropriate for main memory (M). Moreover, **three** different logical data bases coexist, a main memory data base, a disk data base and an archive data base, each of the latter two with caching at higher levels of the hierarchy. Furthermore, an object may exist in any of these three data bases. Lastly, it is possible that one or more of these columns will not be used in a given application. In this case, the model simply contains less columns.

This architecture is a generalization of traditional DBMSs, which assume that all records are in disk format, D even if they are cached in main memory. Main memory caching of system catalogs in a different format then occurs outside the storage manager using specialized code. It is also a generalization of the POSTGRES storage system [STON87], which utilizes two different representations for disk and archive objects. Hence, objects are converted when they move between these levels. However, POSTGRES has no support for main memory objects. Lastly, it is also a generalization of commercial persistent object stores, (e.g. the products from Ontologic, Object Design, Objectivity, and Versant) which usually "swizzle" pointers when an object is moved from disk to main memory. Hence, they support two formats for objects, namely M and D, and convert between these representations when objects are moved. However, none of these products support an archive format.

We now turn to our specific proposal for a storage manager supporting the logical model.

## 2.2. The Storage Manager

The DBMS stores a set of instances (objects, tuples) of collections (classes, tables). The instances of each collection may be in the format appropriate to any logical device, and we can think of the instances in a particular format as forming a **logical** data base. When a user query is submitted to this system, e.g:

retrieve (EMP.name) where EMP.age = 30

it must be executed against some subset of the these logical data bases. To optimize this process, we require a **distribution criteria,** for each collection which indicates the location of instances among the three logical data bases.

Current persistent object stores, e.g. Orion [KIM90], maintain a list of UIDs of instances in main memory, with the remainder assumed to be in disk representation. Specifying which objects are in main memory using a list of UIDs is unduly restrictive. For example, employees over 65 are retired and might be placed on the archive, while those under 30 might exist in the main memory representation and those over 30 in the disk representation. It is very inefficient to require specifying such partitioning using lists of UIDs. Not only is it space inefficient, but also it does not permit optimization of queries. For example, finding employees under 20 is a query which need be directed only to the objects in main memory representation; however unless the partitioning is described semantically there will be no way to figure this out.

On the other hand, POSTGRES uses a semantic criteria to specify the location of objects. Unfortunately, the criteria is hard-coded to be:

disk representation: all objects valid at time = now
archive representation: other objects

A much better alternative would be to allow a user or application program or maybe even the DBMS itself to dynamically specify the semantic composition of objects at each level through a general distribution criteria, and two possible approaches seem reasonable. First, we could require that the criteria for each device be mutually exclusive and correct at all times. One such set of criteria for EMP might be:

main memory representation: EMP where age >= 30 and age < 60
disk representation: age < 30
archive representation: age >= 60

If the criteria form a partition, then any update or insert must be installed in the correct data base before the installing transaction commits. Moreover, certain queries need only be processed for one data base. For example, to find the names of all 25 year old employees, one need only query the disk data base. We will call this form of operation **synchronous,** because the distribution criteria is dynamically kept correct for each device.

The other possibility would be to require the criteria to form a partition as above. However, instead of guaranteeing that each criteria is correct, the execution engine only guarantees that each instance will be on its correct **home** device or on a device that is an **ancestor** of its home device. In this case, each insert or update can be installed on the main memory device, and then moved **asynchronously** at some later time to a lower level. This asynchronous mode of operation supports faster commit than synchronous mode because modifications can be installed in main memory. However, it has the disadvantage, that queries that should be logically processed by device-i must be processed for all ancestor devices in addition.

In a transaction processing environment, we can see the obvious utility of asynchronous operation, while in a decision support application, the synchronous mode might be better. Therefore, one might expect to support both modes of operation; however, we choose to support only asynchronous operation. Because we expect that each device is faster than its descendants by perhaps one order of magnitude, a query to a specific device will generally be much faster than the same query to any of its descendants. Hence, requiring each query to be processed by all ancestors of a device may not be a significant burden.

Our storage architecture assumes a general purpose DBMS whose query optimizer has been modified to produce the correct number of queries to the various actual collections. Moreover, we require a collection of background demons, one per logical storage device, which perform sophisticated storage management functions. Hence, we need a storage manager for main memory, the disk and the archive, and each storage manager controls space allocation on its device. Therefore, each one controls the caching of blocks of lower level objects on its device and also controls the asynchronous migration of objects from its data base to lower level home data bases. To signify that this is much more than a buffer manager, we term this software the **vacuum cleaner.** Hence, the main memory vacuum cleaner is responsible for the main memory buffer pool of disk blocks and archive blocks as well as for reclaiming space in the main memory data base by migrating instances to their home data bases. Of course the vacuum cleaner must be able to identify instances to be moved, so it must contain a complete execution engine. Also, when an instance is moved, the vacuum cleaner must pass the affected instance to a **receiver** for the target data base, who will install the instance.

The last task of the vacuum cleaner associated with each device is to support dynamically changing the distribution criteria, and there are two models of redistribution which we propose. The first model is used by a specific application prior to executing a query and supports **temporary** redistribution of instances, while the second model is used by a data base administrator for **permanent** redistribution. Denote the permanent distribution criteria as {PERM(device)}, and consider a second temporary distribution criteria, {TEMP(device)}. For each device, initially TEMP = PERM.

A user can execute a query in one of two ways. First, he can run the query directly as:

retrieve (target-list) where qualification

For example, he might want the names of employees with low salaries as follows:

retrieve (EMP.name) where EMP.salary < 1000

The query optimizer will construct queries for those logical data bases which have a distribution criteria which intersects the above qualification.

Alternately, he can temporarily move the data to a higher level and then run the query. The syntax he requires is:

elevate objects where qualification to destination

For example, he might specify:

elevate EMP where EMP.salary < 1000 to main-memory

In this case, the distribution criteria changes as follows:

TEMP(destination) = TEMP (destination) union qualification
TEMP(other-levels) = TEMP (other-levels) minus qualification

Elevation is performed synchronously with appropriate locking. An elevate command to a lower level device serves little purpose and will be ignored.

Therefore, if the application accessing the low salary employees expects to access this set several times, it can perform the following:

elevate EMP where EMP.salary < 1000 to main-memory
retrieve (EMP.name) where EMP.salary < 1000

This latter command sequence will cause qualifying instances to be moved to main memory, which may require instance conversion, as well as insertions into main-memory access methods. This overhead will be worth while only if the retrieved objects will be subsequently accessed several times. In this case, the user is requesting data with temporary locality of reference to be elevated to a faster mode of access. When the user is finished with this data, he can cause the data to be returned with a second command:

return objects where qualification

This command will cause each vacuum cleaner to perform the following steps:

1) Identify the instances to be moved from its device, j, to device-i with the following query:

    retrieve (collection-instances)
    where qualification AND PERM(device-i) and TEMP (device-j)

2) Asynchronously return the instances to their correct home.

3) Appropriately update TEMP for device-j.

Because we are dealing with a no-overwrite storage environment, the following optimization can be employed for an elevate-return sequence. When an elevate command is executed the data can be **copied** to the destination, leaving the instances also in the lower level data base. If the higher level objects are not modified, then execution of a return command can cause them to be discarded rather than returned. On the other hand, any updates will result in new instances which require return. As a result, only the new instances need be actually moved back to the destination. The bookkeeping to support this is straightforward. One need only record the time, TIME, of the elevation command. When the return command is executed, the instances to be move back to device-i can be identified by:

    retrieve (collection-instances) where qualification AND PERM(device-i)
        and collection.valid-time > TIME

It is clear that users ofter move data to higher levels and then forget to return them to a lower level when they are done. In this case, the higher storage levels will become cluttered, causing two significant disadvantages. First, space will be taken up that could be better used for caching blocks from lower levels of storage. For example, space in main memory can be used either to store main memory data or to cache disk blocks or archive blocks. The more space that is used for main memory data, the less space that is available for the disk cache. This may mean that there is no space for **worthy** disk blocks such as root nodes of B-trees, etc. The second disadvantage is even more serious. If there is too much main memory data, then data instances will be paged out to a swapping device. In this case, a data base optimized for main memory storage will actually reside partly on disk storage, and the performance implications may be disasterous. For example, main memory data may use AVL trees for an access method. However, in [DEWI84] it is demonstrated that AVL trees perform much worse than B-trees if any significant fraction of their structure goes out to disk.

To alleviate this problem, we propose that each vacuum cleaner be able to issue **return** commands on the user's behalf to free up space if required. Traditional storage managers make such placement decisions solely based on the time since the last access to the object. In our environment, the sets of instances that have been temporarily moved are of various sizes. Therefore, the vacuum cleaner should make placement decisions based on both the time since an instance satisfying the qualification has been touched and by the total size of the set of instances. When qualifications overlap, this will lead to errors, but it is an easy to administer policy.

The above capability supports temporary movement of data under user control from lower levels to higher levels. The data is then returned to its permanent home under user control or vacuum cleaner control. We now turn to a mechanism to support changing the permanent distribution criteria, PERM.

Many users will not take advantage of the **elevate** command because their individual accesses will not justify the cost of the conversion of objects. However, the data base administrator might notice that overall better performance would occur if storage was rearranged. This requires a change in the permanent distribution criteria, PERM, and therefore we require the following **move** command:

move objects where qualification to destination

**Move** changes PERM in the obvious way, i.e:

PERM(destination) = PERM (destination) union qualification
PERM(other-levels) = PERM (other-levels) minus qualification

Like the **elevate** command, movement of instances up the tree occurs synchronously and down the tree asynchronously. However, **move** differs from **elevate** in that movements of instances are performed by actually deleting the instances from the source data base and inserting them in the target data base, rather than by copying them and subsequently invalidating the copies. Because of this fact, a **move** command will cause the vacuum cleaner for device-i to identify instances to be sent to device-j by:

retrieve (collection-instances) where qualification AND NOT PERM (device-i) AND PERM(device-j)

i.e. any temporary redistributions currently in effect can be ignored.

We observe that some collections of objects to be moved may take substantial rearrangement time. For example, to move a one gigabyte collection of objects from archive to disk requires about 83 minutes. It makes no sense to perform this rearrangement synchronously, since the data in question would be unavailable for a long period of time. To deal with this issue, we propose the notion of a **goal** distribution criteria, {GOAL (device)}. The data base administrator can set a new goal at any time for any level using the following command:

target objects where qualification for destination

For example,

target EMP where EMP.salary < 500 for main-memory

The vacuum cleaner for device-i must find instances where:

PERM (device-i) AND NOT GOAL (device-i)

and asynchronously move them to the correct level with minimal locking. Consider the case that there is a term in GOAL of the form:

collection.attribute-1 operator-1 value-1

and that an ordering index such as a B-tree or AVL tree exists on the field, attribute-1. If PERM does not include a clause that uses attribute-1, then the vacuum cleaner records the above clause as its movement goal. On the other hand, if PERM includes a clause using attribute-1, e.g:

collection.attribute-1 operator-1 value-2

then, the storage manager has a movement goal of the form:

collection.attribute-1 operator-1 value-1 AND NOT collection.attribute operator-2 value-2

In either case, the vacuum cleaner can enter the indicated index and identify small collections of records to incrementally move. The required algorithms are sketched in [STON89] for a similar problem, that of incrementally building secondary indexes.

We also propose that the same goal mechanism be used to construct new indexes for instances, which require sufficient time that synchronous index construction is not advantageous. Again [STON89] contains detailed algorithms.

The last requirement is extensions to the type system. We assume an abstract data type (ADT) system of the form in [STON86B]. Therefore, a user can construct a collection using the syntax:

create collection-name (attribute-1 = type-1, ... , attribute-n = type-n)

For example, the EMP collection could be constructed as follows:

create EMP (name = char16, address = point, manager = EMP, age = int4, salary = int4)

This specification indicates the types that the user wishes to give to the system for storage and receive back as answers to queries. However, the types actually stored in the three collections, may be different. To support construction of the various representations, we propose the addition of the following commands:

use name-1 for name-2 on device-1
convert name-1 to name-2 using function-name

For example, the following specification supports a main memory representation of the char16 type:

use m-char16 for char16 on main memory
convert char16 to m-char16 using make-string

Here, make-string is a previously registered function with an argument of type char16 and a result type of m-char16.

When a create statement is processed, the DBMS must construct three actual create statements. To do so it identifies any relevant collection of **use** statements for the types specified by the user and utilizes them in the obvious way. If no **use** statement is encountered, then the DBMS simply utilizes the type specified in the user's create statement. Whenever, a vacuum cleaner is required to convert from one representation to another, it makes use of information in a relevant **convert** statement to identify which function to call.

An ADT system must support indexing for all three data bases. Since each index may be specific to a level, then a user must indicate a specific device in an index creation command as follows:

create sal-index (EMP) on salary as B-tree for disk

If he leaves out the location clause, then the DBMS should assume that the index is to be built for all devices.

The last extension we propose is to allow the data base administrator to specify for each collection which of two security modes he wishes, **trusted** or **secure** mode. In either case, the application program runs in a separate address space from the DBMS. With trusted mode, objects in the collection are placed in a segment which is shared between the DBMS and the application. Thereby, instances can be directly manipulated by the application. Alternately, instances occur in a segment private to the DBMS and records must be exchanged over an interprocess message system.

Trusted mode allows very fast access by the application, especially for main memory data. Once the application had identified a collection of records by running a query to obtain their storage addresses, subsequent accesses could be performed by direct memory access.

## 2.3. How Many DBMSs

It might be argued that the above proposal is equivalent to **three** DBMSs, one for each storage device, and that the software complexity will be prohibitive. In this section, we claim that the above proposal is only modest extra work on top of an extendible DBMS such as POSTGRES. The main pieces of a DBMS are the parser, planner, executor, access methods, utilities, and transaction management system, and we discuss each piece in turn.

The parser is the same one used for a traditional system. Concerning the planner, it must be extended for a tertiary memory environment, and the modest extensions required are discussed in Section 5. The executor must evaluate a query plan for each instance fetched by the access methods. Given that our proposal specifies representation issues within the ADT system, there is essentially no extra work in this module, given that an ADT system has been constructed. Access methods may well be specific to devices, and we envison ones optimized for each type of device. Clearly, there is extra effort required to add access methods; however, if the DBMS has been designed along the lines of [STON87] to allow indexes to be added, then there is no additional complexity outside of the access methods. In the utilities, there is low level code to support each device. However, if any device comes with a reasonable file system, then such code should be modest. Lastly, a single concurrency control system must be used for all devices; hence no extra difficulties occur here, and a no-overwrite storage manager obviates the need for crash recovery code. The only requirement is a vacuum cleaner for each device. Hence, we estimate that a three level DBMS is much closer in complexity to a single extensible DBMS than to three times that complexity.

It might be argued that a two level hierarchy is enough because disk will be formatted the same way as the archive. In some applications this may be true; however, we believe that an architecture allowing N representations will be generally superior to one allowing only two representations.

# 3. STORAGE AND INDEXING OF LARGE OBJECTS

Because very large data bases will usually contain long fields, we discuss their storage and indexing in this section. First, the characteristics of archival memory that constrain the problem of storing long fields are discussed. Then, we discuss record storage in this environment. Lastly, functions may be very expensive to compute, and the section closes with a proposal that addresses this fact.

## 3.1. Memory Characteristics

In this section the abstract model for a three level store is considered. The archive device consists of a collection of 2 or more read/write stations onto which **media platters** can be loaded from storage racks by mechanical robots. The load time for a platter is dominated by the speed of the robotics, which is typically 5-10 seconds in current devices. Once loaded, the time to read or write any specific storage block depends on the characteristics of the device. For optical disks, access times vary from 10-100 msecs depending on how far the disk arm must move to the desired block. For tape media the access time is on the order of 1-100 seconds depending on what tape technology tape is used (9track, 8mm, DAT) and how far into the tape the desired block resides. Once located, a block can be read quickly as sequential read speeds are usually 0.2 mbyte per second or higher. Hence, tertiary store is characterized by the following parameters:

AP -- platter switch time in seconds
AR -- time to move to a random block on a loaded platter in seconds.

AT -- transfer rate in bytes/sec once the desired information is under the read head

For secondary memory, it will be useful to include the following parameters:

DR -- time to move to a random block on a magnetic disk drive in seconds
DT -- transfer rate in mbytes/sec once the desired information is under the read head

Although it is possible to build an I/O controller which would move archive blocks directly to the disk and back, we assume a more conventional organization in which archive blocks are read into main memory. In this case, we assume the existence of two physical block sizes, B1 and B2, which are the units of transfer respectively between the disk and main memory and the archive and main memory. Presumably B1 is 4K bytes, while B2 will be a much larger value, say 64K or 128K.

## 3.2. Storage of Records with Long Fields

Clustering has been studied extensively with a disk as the assumed storage device [CHAN89]. Such studies try to arrange a collection of records so that the number of physical disk reads which must be performed to access a set of related objects is minimized. Similar work assuming an optical disk as a storage device is reported in [CHRI87].

The two popular forms of archives are ones using optical disks and tapes. In tape systems, AR is the dominent time and must be carefully optimized. On the other hand, in an optical disk tertiary memory system, platter switches (AP) are a factor of 100 larger than seeks (AR), and will therefore dominate performance. Consequently, we believe that the important clustering problem for this device is to arrange data records so as to minimize the number of platter switches when accessing a set of records. Therefore, we assume that all the instances of each collection are allocated to a single **home** platter. As a result, queries to a specific collection would be confined to a single media. Since, the platter capacity of most archives exceeds 3 Gigabytes, this will support moderate size collections. Larger ones must be horizontally partitioned [CERI84] to multiple platters by an **archive** distribution criteria. We assume that the archive distribution criteria is a set of clauses of the form:

collection.fieldname operator value to platter-number

Should a user wish to cluster together instances of different collections, he can ensure that they are allocated to the same platters by carefully structuring the collection of archive distribution criteria.

Lastly, we assume that instances of each collection are stored with all short fields together in a record and all long fields stored separately. The reasoning behind this decision is presented in [STON91].

## 3.3. Functions on Long Fields

Functions in a large object environment may be very expensive to compute. For example, given the collection:

EMP (name, address, manager, age, salary, picture)

one might ask the query:

retrieve (EMP.name) where EMP.age > 50 and beard (EMP.picture) = "red"

In this case, **beard,** is a classification function operating on the image of the employee and will take many thousands of instructions to compute. Hence, the second term is vastly more expensive than the first one in CPU time. Moreover, the function **beard** may not require all the bits in a picture to perform the classification; therefore it should be possible for the function to selectively retrieve just the information it requires. Lastly, an index on EMP.picture will not accelerate the above query; rather we require indexes on a function of EMP.picture.

The following proposal addresses these requirements. When a new type is registered with the DBMS, a single extra flag must be supported in the type definition, namely LONG. This flag denotes to the DBMS that the type in question is a candidate for separate storage. For types which are not LONG, functions can reference their arguments either by value of by reference. On the other hand, functions on LONG types must use a special interface. Specifically, when a function on a LONG field is called, it receives a **magic cookie** which is similar to a file descriptor. The function can then use a library of routines to **seek** to a specific location in the long object and then **read** or **write** a specific number of bytes to or from a buffer. Hence, it receives an abstraction for long fields that is the same as that of a file and can buffer as much of the long field as desired. Unnecessary portions of the long field never need be read from secondary or tertiary memory.

We assume that qualifications have clauses of the following forms:

1) collection.fieldname operator constant

2) function (collection.fieldname) operator constant

An example of the first form is

EMP.salary > 5000

Clauses of this sort have long been addressed by query optimizers [SELI79], and the generalization to an abstract data type context performed in [STON86B]. An example of the second form is:

beard (EMP.picture) = "red"

Such classification functions are very common, and example functions for photographs include beard, glasses, hair color, large nose, and scowl.

We assume that indexes are available for the abstract data type fields present in any collection as in [STON86B]. However, we also assume that indexes are generalized to be available on a function of a data type. This idea was proposed in [LYNC88] in the context of textual data, and there is little difficulty in implementing this generalization, as noted in [AOKI89]. Moreover, [MAIE86] proposed essentially the same construct by suggesting that the member sub-objects of a complex object be indexable.

We further assume that archive indexes are stored on the same platter as the data they index. Moreover, they may be cached at higher levels of the storage hierarchy and should be allowed to be built incrementally.

The optimizer must deal intelligently with functions which are very expensive to compute and/or fetch substantial portions of long objects. Earlier, [STON86B] identified a collection of information which must be specified by the definer of each function. This information is used by the parser and query optimizer to process queries on ADT fields. In the proposed environment four additional parameters must be added to this collection when a function, f, is defined:

1) Fraction of archive blocks read -- AB(f)

When a function is applied to a long field, this parameter specifies the fraction of the blocks it expects to read from the archive through the **magic cookie** interface.

2) Fraction of disk blocks read -- DB(f)

In case the object is buffered on the disk or converted to disk representation, the query optimizer can estimate the fraction of the disk blocks that will be read using this parameter.

3) Fraction of bytes examined -- FB(f)

This number represents the fraction of the bytes in a long field that the function must examine. This quantity will be used in the CPU computation to follow.

3) CPU time per byte -- CPU_b(f)

Since classification functions are often extremely expensive to compute, this parameter allows the optimizer to make a better estimate of the CPU time to execute the function. This will also be used in the CPU computation to follow.

4) CPU time per call -- CPU_c(f)

There are occasional functions on short fields that are CPU intensive. For example, suppose passwords are added to the EMP collection and stored in encoded form. Then, a system administrator might want to execute the following command

retrieve (EMP.name) where break(EMP.password) = "easy"

to look for users with passwords that are too easy to break. In this situation, the break function is extremely CPU intensive, even though the password field is short.

With the above three parameters, the CPU time for applying a function to an attribute can be estimated as:

CPU_c(f) + CPU_b(f) * FB(f) * (expected length of the attribute)

If the definer of a function does not specify these parameters, they would be defaulted to the values appropriate for short fields. Reasonable values might be: AB = 1, DB = 1, FB = 1, CPU_b = 10 and CPU_c = 100.

## 4. QUERY OPTIMIZATION

The optimization framework in [SELI79] carries over into the environment of this paper with a few extensions and modifications. This section discusses how the cost function must change and how the space of plans to be considered must be extended.

A traditional optimizer [SELI79] uses a cost function of the form:

query cost = expected (I/Os) + W1 * expected (records examined) (2)

These terms are respectively the expected number of I/O's and the expected number of records examined, multiplied by a conversion factor, W1. When tertiary memory is considered, the above formula must change to:

query cost = expected CPU time +
W1 * (DR + B1 / DT) * expected disk I/Os +

W2 * (AR + B2 / AT) * expected archive I/Os + (3)
W2 * P * expected platter changes

Here, expected CPU time is estimated by multiplying the expected number of records examined by the expected CPU time per record. The second clause is the expected disk time in seconds, and W1 is therefore the system-specific conversion rate between CPU seconds and disk seconds. The third and fourth terms together form the archive time, and therefore W2 is the conversion rate between CPU seconds and archive seconds.

In our environment, each query will be decomposed into as many as three actual queries, one to the main memory data base, one to the disk data base and one to the archive data base. The first term is the only one considered for the main memory query, while the first two terms are considered for the disk query. Only for the archive query are all terms considered.

An optimizer should therefore compute (3) for each possible plan and then choose the expected cheapest one. However, in the environment of this paper, the order of evaluation of clauses that restrict the same collection must be carefully considered. For example, consider the query

retrieve (EMP.name) where beard (EMP.picture) = "red" and EMP.salary = 500

In the case that there is no applicable index, a conventional optimizer will evaluate the two clauses on picture and salary in random order. Since, the cost of evaulating

beard (EMP.picture) = "red"

is very large compared to

EMP.salary = 500

the optimizer should construct a plan whereby the latter clause is evaluated first. Only for those instances with the correct salary must the expensive computation be performed. Therefore, the optimizer must consider the two different orderings of the clauses as separate plans.

In addition, when multiple clauses exist for a collection, a traditional optimizer will process all of them at the same time in some order. However, in this environment, the optimizer must also consider processing the clauses separated by other intervening operations. An example of the utility of this approach is discussed in [STON91]. Also discussed in [STON91] is the necessity of using more than one index when processing a restriction query for a collection.

Therefore, in our environment, if an optimizer is given a query with N clauses spanning K collections, there are as many as N! different ordering of the clauses that merit consideration. For each of these orderings there may be several different actual plans to evaluate. Consequently, the search space grows dramatically relative to the space evaluated by a conventional optimizer.

For each possible plan, the optimizer must construct an estimate for (3) and then choose the expected cheapest one. We need to extend [SELI79] with cost calculations for clauses involving long fields, clauses involving expensive functions on short fields, and costs for archival store access. We now treat these topics in order.

Consider a clause C of the form:

f (long field) operator constant

If there is an index on

f (long field),

then this clause becomes a "normal" one and can be evaluated using [SELI79]. Otherwise, the optimizer must estimate the following constant:

CONST(C) = (expected number of instances examined) * (expected field length)

Hence, the query optimizer must guess the number of instances examined using classical mechanisms, and the average field length is assumed available from the system catalogs. Moreover, it must make two additional estimates for the long fields in each collection:

disk fraction: the fraction of the bytes in the long objects present in the disk cache
m-m fraction: the fraction of the bytes in the long objects present in the main memory cache

The archive fraction is one minus these two numbers. Of course, the obvious restrictions on these numbers for the disk and main memory data bases should be assumed.

As a result, the query optimizer can evaluate the cost of a clause containing a long field as:

CPU time = CPU_c + CPU_b * FB(f) * CONST(C)
expected disk I/Os = (disk fraction) * DB(f) * CONST(C)
expected archive I/Os = (archive fraction) * AB(f) * CONST(C)

CPU intensive functions on short fields are the second kind of access to be individually accounted for. Here, the I/O will be accounted for in the traditional metrics for the rest of the query from [SELI79]. Only the CPU resources need be considered separately, and CPU time for functions of short fields is the same as that computed for long fields above.

We now turn to the cost calculations for operations to archival storage other than long field access. A query plan consists of a collection of query processing steps, each of which is a scan of a collection, an indexed scan of a collection, a join of two collections by iterative substitution, a join of two collections by merge sort, or a join of two collections by hash join. The CPU time for each plan operation can be estimated using conventional means. The disk and archive I/O for sequential scans and indexed scans can be computed by the optimizer using the disk fraction and main-memory fraction for the short fields in a given collection. The number of platter swaps required is one more than the number of platters on which the collection resides.

For merge-sort or hash joins, it is clear that any required temporary collections should be allocated in main memory or on disk. Hence, the archive need be read only during the initial scan of both collections. After that standard disk-based formulas apply. Furthermore, the total number of platter switches for each collection is 1 plus the number of platters that each collection occupies.

The last tactic is iterative substitution. For disk based collections, the standard formulas apply. For archive collections, one should only consider this strategy if there is an index on the inner collection. Furthermore, if the inner collection is a multi-platter collection, then there is a danger of one platter switch per outer record. To avoid this disasterous overhead, iterative substitution should only be considered if the inner collection is clustered on platters in the same way as the outer collection. This requires that both collections use the same field in the distribution criteria used to partition the table. In this case, the number of platter switches is again 1 plus the number of platters for each collection.

Based on these considerations, an optimizer can estimate the cost for any given query plan according to (3) above and then choose the expected cheapest one. However, there are at least two optimization tactics that may be useful to reduce the search space. First, it may be reasonable to automatically delay all expensive clauses to the end of a query. Then, at the end of a plan, they should be executed in ascending order of

function cost * clause selectivity

Second, it may be very desirable to perform multiple query optimization. This tactic has been studied in a disk-based environment when queries have overlapping terms in their qualifications

[SELL86]. In our environment, it may be desirable to group a large collection of queries into a **bundle** and then make a sequential pass through the instances of a collection processing all queries in parallel. Consider the following two queries:

retrieve (EMP.name) where EMP.age > 40

retrieve (EMP.salary) where EMP.dept = "shoe"

If there are indexes on neither age nor dept, then a sequential scan of EMP is required. In this case, both qualifications can be economically checked in a single scan of the collection.

## 5. CONCLUSIONS

We have proposed an architecture for a multi-level storage manager which integrates both main memory and archive data bases into a common framework and substantially generalizes many previous proposals. Specifically, it supports real-time applications, the caching requirements of persistent programming languages, and the needs of applications with very large data bases in a common framework.

Moreover, we have proposed the query optimization support required for the resulting environment. Basically, the optimizer must be extended to cope with:

1) the characteristics of the archive media
2) the desirability of storing long fields in a separate location from short fields
3) the prospect of CPU intensive functions

and we have shown a methodology to accomplish these tasks.

In order to turn a prototype like POSTGRES into the system outlined in Section 3-5, the main steps required are:

1) support for a main memory data base
2) the possibility of indexes on functions of an attribute
3) the replacement of a hard coded distribution criteria with a general one
4) extending the optimizer as noted in Section 5
5) implementation of separate storage for long fields
6) rearchitecting the POSTGRES disk-to-archive vacuum cleaner
7) implementation of a main memory vacuum cleaner

We are currently designing such a system, currently denoted POSTGRES II, with these characteristics. The scope of distribution support in POSTGRES II is also under study.

## 6. REFERENCES


[AOKI89] Aoki, P., ‘‘Implementation of Extended Indexes in POSTGRES,’’ Electronics Research Laboratory, University of California, Technical Report 89-62, July 1989.

[CATT90] Cattell, R. and Skeen, J., "Engineering Database Benchmark," Sun Microsystems, Technical Report, April 1990.

[CERI84] Ceri, S. and Pelagatti, G., "Distributed Databases, Principles and Systems," McGraw-Hill, New York, N.Y., 1984.

[CHAN89] Chang, E. and Katz, R., "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-oriented DBMS," Proc. 1989 ACM- SIGMOD Conference on Management of Data, Portland, Ore., June 1989.

[CHRI87] Christodoulakis, S., "Analysis of Retrieval Performance for Records and Objects Using Optical Disk Technology," ACM TODS, June 1987.

[CHOU85] Chou, H. and Dewitt, D., ‘‘An Evaluation of Buffer Management Stockholm, Sweden, August 1985.

[DEWI84] Dewitt, D. et. al., ‘‘Implementation Techniques for Main Memory Data Base Systems,’’ Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Ma., June 1984.

[DEWI86] Dewitt, D. et. al., “GAMMA: A High Performance Dataflow Database Machine,” Proc. 1986 VLDB Conference, Kyoto, Japan, Sept. 1986.

[DEWI90] Dewitt, D. et. al., " A Study of Three Alternative Workstation-Server Architectures for Object-oriented Database Systems," Proc. 1990 VLDB Conference, Brisbane, Australia, August, 1990.

[HONG90] Hong, W. and Stonebraker, M., "Parallel Query Processing in XPRS," Memo UCB/ERL M90/47, University of California, Berkeley, Ca., May 1990.

[KIM90] Kim, W. et. al., "Architecture of the Orion Next-Generation Database System," IEEE Transactions on Knowledge and Data Engineering, March 1990.

[LEHM86] Lehman, T. and Carey, M., "Query Processing in Main Memory Database Management Systems," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., June 1986.

[LYNC88] Lynch, C. and Stonebraker, M., ‘‘Extended User-Defined Indexing with Application to Textual Databases,’’ Proc. 1988 VLDB Conference, Los Angeles, Ca., Sept. 1988.

[MAIE86] Maier, D. and Stein, J., "Indexing in an Object-Oriented DBMS," OGC Technical Report CS/E-86-006, Oregon graduate Center, Beaverton, Ore., May 1986.

[MOSH90] Mosher, C. (ed), "The POSTGRES Reference Manual, Version 2," Electronics Research Laboratory, University of California, Berkeley, Ca., Report M90/53, July 1990.

[NAVA82] Navathe, S. et. al., "Vertical Partitioning for Physical and Distribution Design of databases," Stanford University, Technical Report STAN-CS-82-957, 1982.

[SELI79] Selinger, P. et. al., ‘‘Access Path Selection in a Relational Data Base System,’’ Proc 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1979.

[SELL86] Sellis, T., ‘‘Global Query Optimization,’’ Proc 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., June 1986.

[SMIT81] Smith, A., "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms", IEEE Transactions on Software

Engineering, July, 1981.

[STON86] Stonebraker, M. and Rowe, L., “The Design of POSTGRES,” Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[STON86B] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," Proc. 1986 IEEE Data Engineering Conference, Los Angeles, Ca., Feb. 1986.

[STON87] Stonebraker, M., "The POSTGRES Storage System," Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[STON89] Stonebraker, M., "The Case for Partial Indexes," SIGMOD RECORD, Dec. 1989.

[STON90] Stonebraker, M. et. al., "The Implementation of POSTGRES," IEEE Transactions on Knowledge and Data Engineering, March 1990.

[STON91] Stonebraker, M. et. al., "Managing Persistent Objects in a Multi-Level Store," Electronics Research Laboratory, University of California, Technical Report M91/16, March 1991.