Audit History and Time-Slice Archiving in an Object DBMS for Laboratory Databases

Development of an object database management system allows rapid, convenient access to large historical data archives generated from complex databases.

by Timothy P. Loomis

The requirements for laboratory databases include many of the same features specified for other types of databases, including enforcement of a rigorous transaction model, support for concurrent users, distributed recovery capabilities, performance, and security. However, the requirements differ from most databases by the emphasis on saving a complete and recoverable record of historical data for some types of data. This requirement comes from the regulatory overseeing authority of the pharmaceutical industry by organizations such as the U.S. Government's Food and Drug Administration or Environmental Protection Agency, and often, the legal importance of the data (patent law). Some examples of historical data in a chemical laboratory include previous values of test results, designated reviewers and approvers of data, methods of analysis, and ingredients used to produce a product. It is necessary to be able to determine when this data changed, who changed it, and why a change was necessary.

Most laboratory database systems have tried to deal with historical data by adding complex logic to the application code to record and retrieve historical data in special tables that are added to traditional relational database schemas. While this technique works for simple schemas with a few objects that need to be monitored for change, its complexity overwhelms development, testing, and support efforts for more realistic databases. In short, it does not scale to the complex databases needed for the future.

Keeping track of historical data became a critical design factor when the HP ChemStudy product was being developed in the laboratory information management system program in HP's Chemical Analysis Solutions Division. HP ChemStudy controls all the information used in multiyear projects that determine the expiration dates on drugs. The database is complex with 128 types of application objects interconnected through numerous relationships. It is necessary to be able to reproduce the contents of objects and the state of their relationships at any time in the past to satisfy regulatory requirements.

Our solution to the historical data challenges of laboratory databases has been to develop a database management system (DBMS) that provides built-in support for historical data for any object and for groups of objects that are connected through relationships. The simplicity and extensibility of this system are possible because we have developed a pure object DBMS (ODBMS) in which relationships are themselves objects. Although the ODBMS provides many advantages for applications development, this article will concentrate on the issue of historical data.

The ODBMS is implemented in C++ on the HP-UX* operating system and Windows® NT.

System Overview

Before considering the details of how historical data is managed in the database, we need an overview of the distributed ODBMS to understand how an object is created and stored. While this modular system can be configured in many ways, Fig. 1 presents an example configuration that is used in the HP ChemStudy product.

In Fig. 1, a client is a process that incorporates C++ class code that defines application objects. While the object created by the application can contain any data needed in the application, the object is managed (locked, updated, saved) through the services of the generic object manager module. The object manager also controls logical transactions (commit and rollback) and provides save points and other DBMS functions. At the object manager level, all objects are treated alike and no changes are required to support any new application object types. The client may have a user interface (shown as a graphical user interface (GUI) in Fig. 1) or it could be an application server with code to support its own clients.

The object manager can connect to one or more object servers that control a database. The ability to connect to multiple object servers makes the system a distributed DBMS and necessitates a two-phase commit protocol to ensure that a transaction affecting multiple databases works correctly. The distributed capabilities of the ODBMS are employed for archiving operations (described below) and for integrating data from multiple active databases.

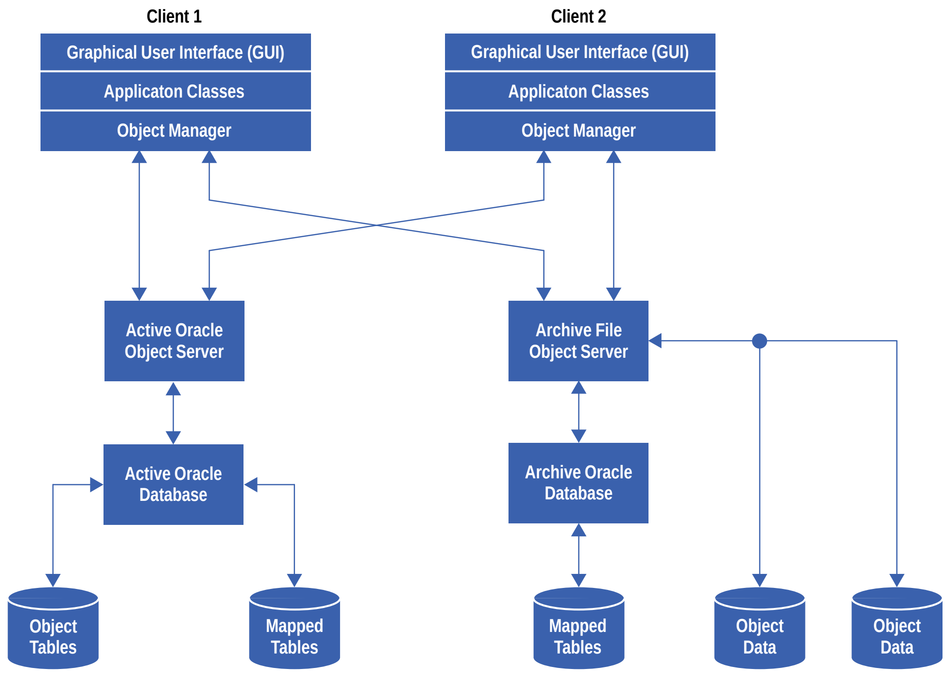

Fig. 1. *Example ODBMS configuration.*

Currently, we provide two types of object servers which differ only in the driver code module that stores object data. From the point of view of a client process, there is no difference in the way an object is treated. The Oracle object server stores an object in Oracle tables while the file object server stores the object in one or more redundant file structures as object data. While the file version is faster than Oracle for read and write by a factor of 30 to100, some customers prefer the Oracle version because it conforms to their corporate information systems requirements. The file version also stores data more compactly and is ideal for embedded databases that are not visible to users and for databases in which the speed of storing and retrieving data is critical. Because the object data stored by either type of server is binary, multimedia data or a binary file can be stored by breaking the data into objects. Objects are also useful for processing a large binary data file in clients that do not have enough memory to hold all the data at once.

Laboratory databases become so large that it is necessary to remove old data periodically from the *active* database and place it in some type of *archive* for long-term storage. Most systems have used a special storage medium for archived data and require that the data be *dearchived* back to the active database for review. Instead, we use the distributed capabilities of the ODBMS to transfer data from the active database to an archive database as a simple distributed database transaction. The archive database can then be taken offline without limiting current operations. Fig. 1 shows an Oracle server being used for the active database and a file version being used for an archive database.

The object database provides access for C++ object applications but lacks facilities for ad hoc queries and reports that can be customized by a customer. To accommodate ad hoc queries and report writers, a collection of mapped tables can be created that provide a more traditional relational database schema of the application data. Each type of C++ object can be mapped to its own table in the map schema when it is inserted or updated, but it is always read by the application from the object database. In practice, only some data in selected objects is mapped. This object-relational DBMS combination has proven to be very successful at providing the customer with reporting flexibility, while preserving the speed and simplicity of a pure object system for the application code.

An example of mapping is shown in Fig. 2. The example considers three objects of three different types: Dept, Emp and EmpList (relationship). A client connected to the object server transports binary objects to and from the server cache. Except for objects newly created by a client, all objects in the cache have persistent counterparts in the object database and are read into the cache from this database. All objects are inserted or updated in the object database during the commit operation. At the option of the application designer, selected data from an object can also be mapped to the map database as shown for the Dept and Emp objects. The EmpList relationship object is not mapped in this example. Relationships are usually defined using *foreign keys* in relational schemas.

We can see from this overview that an object is a bundle of data that can exist simultaneously as a C++ object in multiple clients, as an object in the cache in the object server, as object data in a database, and as mapped data in a relational table. Managing the relationships among these multiple representations of an object requires adherence to a rigorous transaction model. Many of the features necessary to deal with historical versions of an object are extensions to controls that already exist for object data.

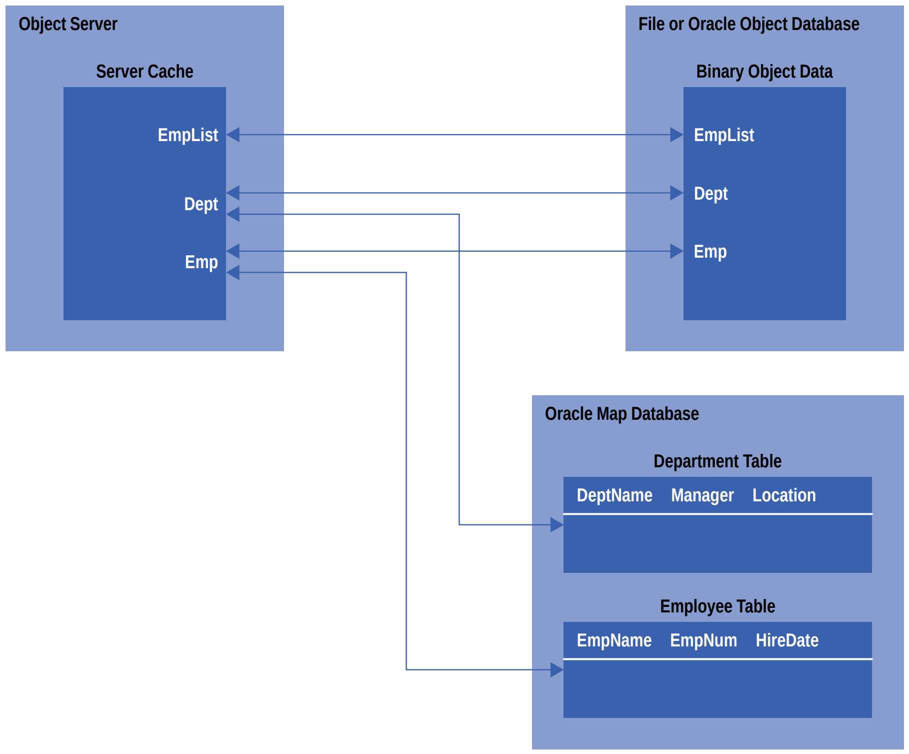

Fig. 2. *Object mapping example.*

Auditing Laboratory Data

There is more to a database data item than a value that can be retrieved. For example, that value was created by someone or some calculation, it may have been converted from a string representation with a specific precision, it was created at some date and time, it may have some application-specified limits that cannot be exceeded, and so on. Moreover, the current value may have replaced a previous value, requiring a justifying comment, and it may be necessary to retrieve all earlier values of this data item. It has long been a requirement for laboratory databases to maintain this type of information associated with a laboratory measurement and to record a history of changes to the measurement. We generally refer to the process of maintaining a record of a value and its associated information through time as *auditing* or maintaining an *audit trail*. In the context of an object database, auditing means keeping a record of the history of an object and objects associated with it through relationships.

Auditing database data has generally meant keeping a separate record or audit log of selected changes made to the database. For example, Oracle provides the capability to audit user, action, and date for access to selected object types but requires a user to write triggers to record changes to data values. While this straightforward mechanism does accomplish the task, its use for large and complex databases rapidly generates huge volumes of data that require sophisticated searching to identify particular changes of interest. A simple audit log of database changes is practical only if one hopes that it will never be needed! Audit logs are routinely needed in the pharmaceutical industry and will soon be a common requirement for other industries subject to regulatory oversight, such as software development processes subject to ISO validation. Searching through a huge audit log is not a reasonable way to answer an auditor's questions about the history of an object that may contain, or be associated with, hundreds of component objects.

The alternative to an external audit log is a DBMS that has an intrinsic method for auditing an object and its relationships. In the next section we discuss general methods developed to audit selected classes of composite objects stored in an ODBMS so that the audit data can be retrieved easily.

The subject of temporal databases has received considerable research attention directed mainly toward extending the relational model and providing time-based query methods.[1,2] The implementation presented here differs from these models principally by:

- Using an object model
- Using relationship objects together with lock-and-update propagation to synchronize the time history of related objects, rather than attempting to deal with the more general problem of "joining" any set of objects
- Being a working implementation for audit-trail applications that deals with load errors and numerous practical programming problems.

Commercial extended relational databases such as Illustra[3] are beginning to provide some time-based capabilities for specialized data.

Example Schema

Auditing an object is complicated by references to other objects. Consider Fig. 3, which shows an abbreviated class schema for a division of a company containing departments, department offices, and employees within departments. Relationship classes (objects) derived from the class list are shown explicitly in this diagram because they are important in auditing. (For clarity all lists are shown as separate classes rather than as inherited base classes). A reference to another object is shown explicitly as an arrow in this diagram because we will be concerned with the details of propagation of information between objects. A line terminated with a dot (as in IDEF1X or OMT modeling)[4] indicates that references to multiple objects can be stored. An A in the lower-right corner of a class indicates that objects in that class are audited.

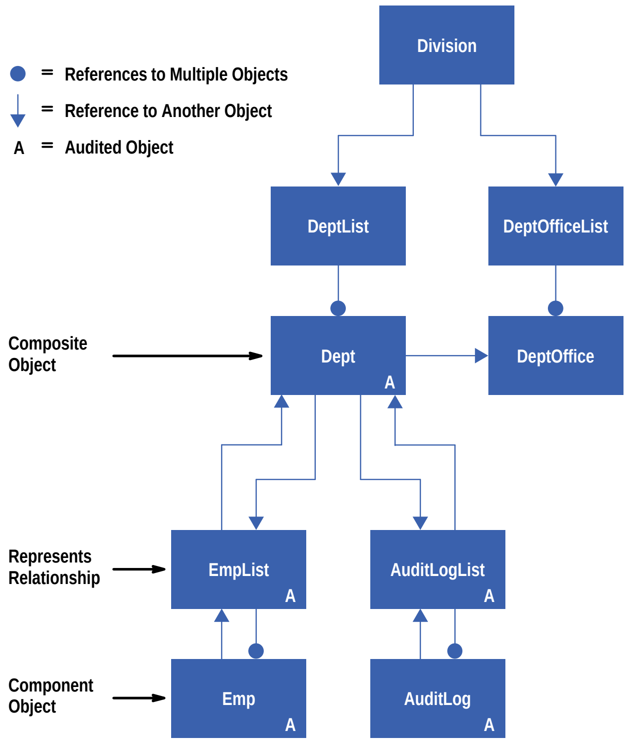

Fig 3. *Example class schema.*

Composite Objects. Audited relationships should be used to contain the components of a composite object. A composite object is one that can be considered to logically contain other component objects in the application. More precisely for our purposes, a composite object can be defined as one that should be marked as changed (updated) if component objects are added, deleted, or changed even if the data within the composite object itself remains unmodified.

In the example of Fig. 3, we will consider a Dept to be a composite object because it logically contains Emp component objects. An EmpList object is the relationship or container connecting the composite and its components. We consider Dept to be a composite object in this example because we implicitly include all the employees in the department as part of the department and want to consider the department to be modified if there are any changes to any of the employees. Alternatively, we could have considered Dept to exist independent of its employees. Clearly we can sink into the dark waters of a long philosophical discussion here (If you change the engine in the car is it the same car?), so the design is best approached physically. The basic question is whether examination of the history of a composite object should reflect changes to its component objects. For many complex objects in our products the answer is yes.

Two references are necessary for an audited relationship. References traversed from Dept to Emp are called *component references* and the reverse references are called *back references*.

Audited and Nonaudited Objects. As exemplified by the use of classes derived from the list class in audited and nonaudited relationships, auditing can be specified on a subclass or an individual object. Moreover, it is permissible to turn auditing on only after some event in the life of an object. For the moment, we consider only the case where an object in an audited class is audited from inception.

We see in Fig. 3 that objects of the composite Dept class should be audited from creation but that DeptOffice and Division are never audited. Semantically, this design means that the history of a Dept object, including the composition of all of its component Emps, can be retrieved at any stage of its history. In contrast, the DeptOffice for the Dept and the list of Depts in the Division can be retrieved only for their current values.

Audit Mechanism

Auditing Objects. Auditing an object means that all images of the object must be maintained in the database, starting with the image that existed when auditing was turned on. In contrast, only the latest (current) image of a nonaudited object is retained. Note that when an audited object is to be written to the database, the decision to replace the old image depends on whether the old one was audited. Successive object images generated through update will be referred to as *revisions* of the object, whether the object is audited or not. The revision number is used by the ODBMS to ensure that a client is working with the correct image of an object. There can be only one current revision of an object and only the current revision can be updated.

The term *version* is used for the concept of distinguishing variations of an object that can all be current. For example, different versions of a glossary can exist for different languages but each version may undergo revision to add terms or correct errors. An object is also marked with a *commit timestamp*, which is exactly the same for all objects in a (possibly distributed) transaction. These attributes of an object, along with its identifier and other data, are contained in a header that is prepended to the object in the database and maintained separately by the C++ object in the client object manager.

Auditing Relationships. Auditing relationships requires some mechanism for recording the history of the relationship. Rather than implement a database relationship mechanism and audit it separately from auditing objects, it makes sense to implement relationships as objects themselves. Auditing a relationship is then no different than auditing an object.

Deleting Audited Objects. Deleting an object becomes complicated when the object is audited because the object still exists in the database until the delete is committed. The delete action must be represented in the database somehow, so that the timestamp and revision number marking the end of its life are available. We use a pseudo-object for this purpose. Archiving audited objects, or portions of their history, may involve actually referencing and loading these pseudo-objects representing the delete operation.

Update Propagation. An important objective of the audit mechanism should be to update the minimum amount of information to document a change fully. For this reason we reject the simple “archive copy” approach to auditing whereby the entire composite object is copied each time a component changes. Thus, we should not simply make a copy of the entire Dept composite hierarchy just because an Emp changed because this produces a huge amount of redundant data.

Auditing a composite hierarchy is implemented in our system by propagating the update of a component through the relationship and composite parent objects using back references. For example, updating member data in an Emp object will trigger an update in the EmpList and Dept but will not necessitate an update or copy of other Emps or of other components of Dept. It is necessary to mark composite objects as updated even though their member data has not changed because the composite they represent has changed. Note that there is nothing to be gained by updating a nonaudited object that references an audited one because it does not have a history corresponding to the past history of the referenced object. Therefore, for example, Division is not updated when Dept changes.

It is impractical to expect programmers to follow these back references each time they update an object. It is also asking for bugs to expect them to qualify the propagation correctly according to audit state and update type. We have solved this problem by incorporating back references implicitly within relationship objects and component objects. The object manager code propagates updates automatically as appropriate.

The audit contents of a database can be illustrated using Fig. 4, an example history of a part of the example schema in Fig. 3. The number shown for each object at a particular time is its *revision number*, a simple count of the number of database transactions that have changed the object. We see that Division has not been changed since it was created. DeptList was created at the same time as revision 1 but has been modified twice since then (when Dept1 and then Dept2 were added). Since DeptList is not audited, only the last revision (revision 3) exists in the database.

Object	Time →						
Division	1						
DeptList							3
Dept1		1	2	3	4	5	
EmpList1		1	2	3	4	5	
Emp1			1				
Emp2				1	2	3D	
Dept2							1
EmpList2							1

Fig. 4. *Example object history.*

The behavior of audited objects is different. Dept1 and its EmpList1 were added to the DeptList as revisions 1. When Emp1 was added to EmpList1, the update was propagated to Dept1 as well as EmpList1 so that the revision of the composite object Dept1 reflects a change to one of its components. The same thing happens when Emp2 is added. Note that Emp1 is not updated in this operation, nor does the update propagate to the nonaudited DeptList. A subsequent update of Emp2 (revision 2) similarly causes propagated updates to EmpList1 and Dept1. To make the example interesting, Emp2 has been deleted, represented by the creation of the pseudo-object with revision number 3D. This object really exists in the database as a marker of the end of the life of Emp2 (figuratively, we hope). Just as for an update, this delete operation causes an update of EmpList1 and Dept1.

Lock Propagation. For pessimistic concurrency models it is necessary to acquire an explicit lock on all objects to be updated at commit. Consequently, the object manager should propagate exclusive locks in the same way that it propagates updates and be able to deal with restoring locks to their original type if the propagation should fail partway through the propagation.

Audit Log. Another objective is to summarize changes to the composite Dept object in one place. In this example, suppose there are several changes to each of three Emps and to some other components (not shown) in a single transaction. The update mechanism records the fact in the Dept object that something changed in at least one component object in this transaction, but we need the AuditLog text object to itemize the specific changes bundled in that transaction. Fig. 3 shows a list of AuditLog objects hanging from Dept. Each AuditLog object summarizes the changes for the composite Dept object during a transaction. From the user's point of view, a convenient implementation is to generate one-line entries in the log automatically for each change the application makes to a component object or the composite object, and then require the user to add only a summary comment before commit.

Object Access

Revision and Time Retrieval. An audited object can be retrieved from the database by specifying either a specific revision of the object or by specifying an absolute time and finding the object that was current at that time. A special time token represents current time (also known as NOW in the literature), corresponding to the most recent object revision. Accessing objects by absolute time requires that the commit timestamp of an object be determined so that it corresponds correctly to the actions of multiple clients in a distributed database environment. A consistent source of time must be available to all clients and time must be specified precisely enough to distinguish two transactions on a fast network.

An example is the best way to explain why both access methods are needed. A common way to query the database history in Fig. 4 would be to locate the current Dept1 and then ask to see each of its previous revisions. Retrieving revision 5 of Dept1, the system would use its commit timestamp to retrieve revision 1 of Emp1 and not find Emp2 because it was deleted at this time in EmpList1. Moving back in time to revision 4 of Dept1, its EmpList1 would recover revision 1 of Emp1 again and also find revision 2 of Emp2. Instead of starting with the current revision of Dept1, the initial query could have specified any absolute time, say one somewhere between revisions 2 and 3 of Dept1 to find revision 2 of Dept1, then the commit timestamp of revision 2 would be used to find component data.

Multiple Revision Management. A consequence of auditing objects is that multiple revisions of the same object can exist in the client cache at the same time. This presents a number of practical problems for application programmers who need a simple mechanism for specifying the correct object revision to access. We have found that extending the meaning of locking an object to include cache management of old and current revisions of an object as well as the traditional meaning of granting an explicit lock on the object is a practical solution to this problem.

Accessing Objects through References. Mixing audited and nonaudited objects in the same application exposes the implementer to numerous opportunities to generate run-time database load errors. Despite the problems of a schema with both audited and nonaudited objects, it is often necessary to mix the two to avoid creating impractical quantities of data in the database. A few referencing rules, if they can be enforced, solve the problems.

- Rule 1: Current access to nonaudited objects. A nonaudited object must always be accessed as a *current-time* object, meaning the latest one available from the database. For example, all revisions of Dept use current time when accessing DeptOffice because old revisions of DeptOffice do not exist. If an old time were specified in the access request and DeptOffice had not been changed, the access would succeed, but a few minutes later, after DeptOffice had been updated by another client and its timestamp had changed, the same request would fail!

  This rule is simple enough but does introduce some opportunities for apparently inconsistent behavior. For example, if a report generated for a Dept uses the reference to DeptOffice to include its room number, the same report repeated later on the same revision of the Dept could have another room number if DeptOffice had been changed. Worse, the DeptOffice could have been deleted from the Division causing a load error. These apparent problems are not the fault of the database system but rather intrinsic in the heterogeneous schema. They are solved either by auditing DeptOffice or by indicating that DeptOffice is deleted by status data within the object rather than deleting the object.

- Rule 2: Qualified access from nonaudited to audited objects. As explained above, an access time or specific revision number must be specified when accessing an audited object. For example, the Division can reference a Dept in three ways: by specific revision, by current time (meaning the latest revision), or by absolute time. In practice, a user does not generally know a specific revision of the Dept object or a specific commit timestamp. Therefore the most useful access times are current time or an absolute time the user specifies for some reason.

A continuing complication when accessing audited objects is that the object exists at some times but not others. For example, if we delete the Dept when it is transferred out of the Division, we can't simply delete it from the DeptList because we may need to access the old Dept information in the future. Thus, the reference to a Dept should be tested for accessibility before we try to load it for a specific time to avoid a load error. These problems are solved if we simply audit the DeptList and Division.

- Rule 3: Self-timestamp access between audited objects. The easy and foolproof way for an audited object to access another audited object is for it to use its own commit timestamp. Furthermore, it is permissible for an audited object to drop a reference when the object is deleted (or for any other reason) because its previous revisions will still have the reference. However, there are some complications.

  It may be necessary for an object to access the same object in different ways. Suppose the DeptOffice in Fig. 3 were audited. If we create a report on a revision of Dept and include DeptOffice information, the method in Dept creating the report should use its timestamp access to DeptOffice to get contemporaneous information. However, if a Dept method is programmed to update the DeptOffice, say with its identification information, it is important that the current DeptOffice be accessed, because only a current object can be updated. As long as the Dept is updated first, timestamp access can be used for both but it will not work if the update in Dept is marked after accessing DeptOffice. In general, it is safer to code current access explicitly when updating a referenced object.

Midlife Changes of an Object. It is permissible to change an object from nonaudited to audited at some time in its life. Probably the most common reason to do this is to avoid generating large amounts of data while an object is in some draft stage and being updated frequently. Keep in mind that the object can be a composite object hierarchy encompassing hundreds of large objects. Only after some approval stage does the application really want to track the life of this composite construct.

Making an object audited may change the rules it uses to access component objects and propagate updates. By implementing these mechanisms in object manager utilities, the change can be made transparent to most application developers.

Schema Constraints

The previous discussion leads to a simple rule for auditing classes in a schema: audit the components and relationships if the composite is audited. For a composite object to truly represent the state of a component hierarchy, all the components and component-composite relationships beneath the composite must be audited when the composite is audited. Only then will locks and updates be propagated correctly and can the composite use its timestamp to access its components reliably.

For example, Fig. 3 shows the AuditLog as audited even though we expect to create only a single AuditLog revision for each transaction. Marking it audited follows the rule to acquire the programming simplifications enumerated above. There is really no penalty in this case, because storing one revision of an audited object takes no more room than storing one revision of a nonaudited one.

There are reasons for breaking this rule. In large realistic systems (in contrast to small demonstration ones) we face realistic constraints on space and often somewhat ambiguous application requirements. As an example, consider DeptOffice which is marked as nonaudited in Fig. 3. If we assume that there are good application reasons for not auditing DeptOffice, we have to carefully access the references between Dept and DeptOffice according to the complications discussed above and accept the apparent inconsistencies that these relationships may produce.

Database Storage

Object storage implementations are beyond the scope of this article, but it is worthwhile to mention a couple of considerations. First, it is not necessary to have a specialized database to store audited objects. We have implemented an auditing database that can use either Oracle tables or our own file storage manager. The main complications are:

- Providing an efficient access method that will find an object current at a time that does not necessarily correspond to a timestamp
- Handling pseudo-objects representing delete.

Second, it is advisable to provide efficient access to current objects. Because audited objects are never deleted it is not unreasonable to expect hundreds of copies of an object in an old database. Most applications will primarily access the current revision of an object and have to stumble over all the old revisions unless the storage manager distinguishes current and old audited data. It may be worth introducing some overhead to move the old revision of an object when a new revision appears to maintain reasonable access efficiency.

Some object database systems map object data to relational tables. The relational system can represent the primary object depository or, alternatively, only selected data can be mapped to enable customers to use the ad hoc query and report-writing capabilities of the relational database system. Extending these systems to handle audited data simply requires adding a revision number, timestamp, and object status code to the mapped data. The ad hoc user should be able to formulate the same type of revision and time dependent queries of the relational database as a programming language does of the object database. The status is necessary to distinguish old audit data, current objects, and deleted pseudo-objects.

Archiving

A lot of database data is created very rapidly in auditing databases. At some point some of it must be moved to secondary storage as archived data. As usual, auditing database systems pose special challenges for thinning data without corrupting the remaining objects.

What Is an Archive? Several types of archives are possible. One common repository is a file containing object data in a special format and probably compressed. Data is moved to the archive using special archive utilities and must be *dearchived* back into the active database for access using the same special utilities. This method maximizes storage compactness but pays for it by a cumbersome process to retrieve the archived data when needed. Another possibility is to move data to a separate data partition (table space) that can be taken offline. Access to the archived data might require dearchiving or, if the complexity is tractable, unioning the archived data with the active data in queries.

At the other extreme is the use of a distributed database system to connect the active and (possibly multiple) archive databases. The archive medium, then, is just another database that should have read-only access (except during an archive operation by system utilities). A distributed database system connects the active and archive databases during the archive and dearchive processes, allowing the data to be moved between databases as a distributed transaction. This is the method we have chosen to use in our products. A distributed archive system allows continued growth of archived data while retaining reasonable access times when necessary. Another advantage is the reliability of the archive and dearchive processes because they are a distributed transaction subject to two-phase commit protocols and recovery mechanisms. Finally, it is possible to access archived data automatically without dearchiving if the archive database is on line. This indirect access feature is explained more fully below.

Archiving Entire Objects. The first mechanism for thinning data is to remove objects that will no longer be modified. Generally status within the object indicates when this state of life has been achieved or, perhaps, just the time since the object was last modified is sufficient. Can we just remove all revisions of the object from the active database and put them in an archive record?

The first problem is simply finding the old object because it might have been deleted. It might not even be in the list of current objects in a nonaudited list. For example, in Fig. 3 we had better not delete a Dept or delete it from the DeptList until the time comes to archive, or we will never be able to find the orphaned object. When archiving a Dept it would be an oversight to archive just the current Emps. What about the one that was deleted earlier in the life of the Dept and is referenced only in an old revision? Fig. 4 shows this to be the case for Emp2 in Dept1. Evidently, it will be necessary to search all the old revisions of all composite objects just to identify all candidates for archiving. A special key field to identify all components of a composite to be archived is a big help here.

The second, admittedly mechanistic, worry is how to remove an audited object, since deleting actually results in inserting a new pseudo-object, and we can't even access a deleted object at current time! Presumably some additional code design and implementation provides a mechanism for actually removing an audited object and all of its old revisions, as well as accessing deleted objects. This operation is called *transfer out* to distinguish it from deletion. Similarly, the database must allow *transfer in* of multiple object revisions, including pseudo-objects representing delete.

Now we can move on to the problem of other objects that access the archived object. Because archiving is not deleting, objects that reference an archived object need to retain these references in case the archived object must be accessed in the future. For example, we should retain an entry in the nonaudited DeptList for an archived Dept object even if it is not immediately accessible. One solution is to place a status object on each relationship in the DeptList. This status object can contain archive information. Another solution is to replace the archived Dept object (and its components) with a placeholder object that marks it as archived and could also contain archive information. Unless we want to start changing references in old objects, this new placeholder object will have the same OID (object identifier) as the old one. A variation on the second method is to record archive information within the ODBMS and trap references to archived objects.

These solutions work if the referencing object is not audited. But what if it is audited? Updating the current object or marking the status of its reference to the archived object may be satisfactory for current time access but will result in a load error if older revisions attempt to access the object using references that were valid back when the old revision was current. Unless we want to start updating old revisions (a scary idea if we want to trust the integrity of audited data), the archiving mechanism must handle these old references between audited objects without modification or qualification of the old references. The general solution to the archive-reference problem probably must be implemented at the database level. The database lock or load mechanism must be able to distinguish a reference to an object that never existed for the revision-time criteria specified from one that existed but is now archived. The user must be notified that the data is archived without disrupting normal processes.

Incremental (Time-Slice) Archiving. In some applications it may not be practical to archive entire objects. The life time of some archivable objects (actually composite objects with thousands of component objects) in some systems can be as long as five years, making archiving the object theoretically possible at some time but not very useful for reducing online data on a monthly or yearly basis. Clearly a mechanism for archiving just the aged revisions of objects is necessary in these applications.

The best way to specify incremental archiving is on a time basis, because time can be applied uniformly to all objects. In this scenario we could specify a list of candidate archive objects and a threshold archive time, such that all revisions of these objects found with a commit timestamp equal to or earlier than the archive threshold would be moved to the archive. Well, actually, not quite all of them! Since we must satisfy requests by the active database for the object at the threshold time, we must keep the one object revision with a commit timestamp before the threshold time because this revision is current at the threshold time (unless the object was deleted, of course).

To implement this incremental archive mechanism, as described so far, the system must keep track of the threshold time and archive information about the revisions of each object. Attempted access to revisions extant before the archive time should receive an archive error and perhaps supply the archive information so that the user knows where the data can be found.

In this scenario, archiving probably is not a one-time operation. What do we do with the remaining revisions of the object when the archive operation is repeated a month later, specifying a threshold archive time one month later than that in the previous operation? From a bookkeeping point of view, it would make sense to simply append the new archive revisions of an object to the old ones in the archive and update archive information in the active database. In practice most customers will not find this method any more acceptable than filing tax records by subject rather than date. Most archive time slices will be kept as an archive record labeled by the date range of the data it contains; it could be a tape collecting dust in a rack. If we needed to append to an archive whenever more revisions of a long-lived object were archived, the archive operation would eventually require mounting many archives. Thus, a practical archive mechanism must allow various revisions of an audited object to be scattered in multiple archive databases.

If a single object can be contained in multiple archives, we must know which archive might contain the requested data. Moreover, it would be nice to guarantee that the load request could be satisfied if the archive were made available. A customer will be upset if the archive supposedly containing the missing data is found and mounted and then the customer is told that the data still missing! Thus, it will be most convenient to retain in the active database complete information about the range of revisions and commit timestamps of an object in each archive. This archive record, called an *archive unit*, contains information about the continuous sequence of object revisions of an object that were transferred in the archive operation.

An example of time-slice archiving is presented in Fig. 5. An audited object identified by ObjNum 101 has created 10 revisions in the active database. At some time in the past, an archive database was created, designated as 1995 here. The first time-slice operation moved revision 1 to the archive database and left an archive record in the active database. The archived object acquired a new identifier, shown as 23, because an ObjNum is unique only within a single database. Subsequently, another archive operation moved revisions 2 and 3 to the same database, leaving another archive record. The following year, another archive database was created and revisions 4, 5, and 6 were archived here.

Dearchiving and Archive Access

Dearchive Operation. The process of dearchiving is just the reverse of archiving, whether the archive medium is a compressed file or a remote database. If incremental archiving is used and an archive record is maintained in the active database, it reduces bookkeeping to dearchive an archive unit (group of continuous object revisions) and remove the archive record from the active database. It is also necessary to dearchive archive units continuously from the youngest one to the target one to ensure the integrity of the time-retrieval mechanism. There must be a continuous revision sequence from the current timestamp to the timestamp preceding or equal to the target timestamp.

Indirect Access to Archive Data. Of greater interest is the possibility that dearchiving may not be necessary. If archived data resides on archive databases in a distributed database system, it is possible for a sophisticated object manager to access archived data in remote archive databases and integrate it with the active data. Important advantages of this mechanism are:

- Reduced resources for the active database because dearchiving is not necessary
- Transparent access to archived data by ordinary users
- Reduced administration, because the archive and dearchive processes become simply distributed transactions without introducing special mechanisms into the life of a system administrator.

This mechanism relies on maintenance of an archive record in the active database that records information about each archive unit placed in an archive database. The existence of an archive record in the active database allows the active database to return a *forwarding reference* instead of a load error when a requested revision or time of an object has been archived. The reference contains the address of the archive database, allowing the object manager to proceed to indirectly load the archived object as an alias for the requested one. Obviously, alias objects must be marked to prohibit update. The object manager can take the appropriate action to access archived objects (or revisions of objects) depending on the wishes of the user and system policy. In our system, the object manager recognizes several access modes to indicate how to treat archived data for each application operation.

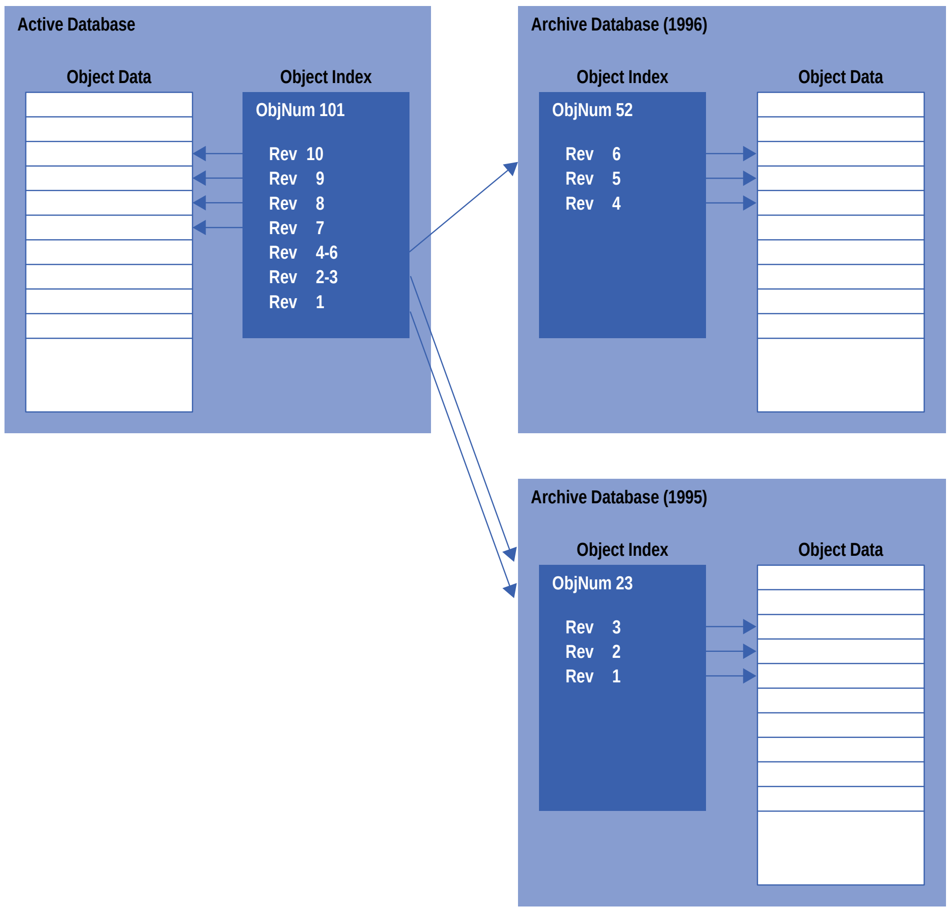

Fig. 5. *Time-slice archive example.*

Conclusion

The trend towards requiring audit trails of more and more processes is driving new database capabilities. Old models of audit logging and periodic archives do not provide routine access to audit data and are not scalable to large systems. We should not view auditing as a specialized, application-specific capability to be overlaid on a general-purpose database.

Object database systems are well-suited to implement this new technology because much of the technology can be incorporated efficiently within the DBMS, freeing the designer and programmer from many of the new complexities introduced in the discussion above. Ad hoc implementations using stored procedures, triggers, or other enhancements of relational databases will have difficulty matching the efficiency of systems in which auditing is an implicit capability.

Auditing objects in complex schemas and archiving the data in a distributed environment are complex processes that would appear to be difficult to implement in ordinary applications. On the contrary, we have found that these capabilities can be used reliably by application developers because most of the complexity can be concentrated in the object manager of an ODBMS and core class code. Similarly, access to archived data can be nearly transparent to most application code with judicious use of access modes and exception traps if the object manager implements automatic indirect access to archive databases.

The ambitious goals of rapid access to active data, convenient access to old data, practical database size, and reasonable application complexity can be achieved in an internally audited system by careful design of a distributed database system.

References

1. N. Kline, “An Update of the Temporal Database Bibliography,” *SIGMOD Record*, Vol. 22, no. 4, 1993, pp. 66-80.
2. A.U. Tansel, et al, *Temporal Databases*, Benjamin/Cummings, 1993.
3. *Illustra TimeSeries DataBlade*, Illustra Information Technologies Data Sheet, January 1996.
4. J. Rumbaugh, et al, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

Online

More information about the products associated with this article can be found at:

http://www.dmo.hp.com/apg/products/chemlms.html

▶ Go to Subarticle 10a

▶ Go to Next Article

▶ Go to Journal Home Page