A Shared, Segmented Memory System for an Object-Oriented Database

MARK F. HORNICK and STANLEY B. ZDONIK
Brown University

This paper describes the basic data model of an object-oriented database and the basic architecture of the system implementing it. In particular, a secondary storage segmentation scheme and a transaction-processing scheme are discussed. The segmentation scheme allows for arbitrary clustering of objects, including duplicates. The transaction scheme allows for many different sharing protocols ranging from those that enforce serializability to those that are nonserializable and require communication with the server only on demand. The interaction of these two features is described such that segment-level transfer and object-level locking is achieved.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*; *data types and structures*; *modules and packages*; D.4.2 [**Operating Systems**]: Storage Management—*segmentation*; *virtual memory*; H.2.2 [**Database Management**]: Physical Design—*deadlock avoidance*; H.2.4 [**Database Management**]: Systems—*distributed systems; transaction processing*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*file organization*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*clustering, retrieval models*

General Terms: Design, Experimentation, Languages, Performance

Additional Key Words and Phrases: Asynchronous communication, CAD transaction processing, data models, locking, object clustering, object-oriented databases, object server

1. INTRODUCTION

Modern workstation technology has made possible a new set of applications. These applications can be characterized as interactive and design based. The basic model is of a worker designing artifacts by using a set of intelligent tools. The artifacts will vary depending on the application, but the common activity seems to be design. Examples of these design environments are electronic and mechanical computer-aided design (CAD) programming environments and office information systems. For the latter, the office worker designs reports, graphics, slide presentations, and decision models.

This research was supported in part by the National Science Foundation under grant DCR 8605597, by the International Business Machines Corporation under contract 55917 and amendment contract 643513, by the Office of Naval Research under contract N00014-86-K-0621, and by DARPA under ONR contract N00014-83-K-0146, ARPA order 4786.
Authors' address: Brown University, Department of Computer Science, Providence, R.I. 02912. M. Hornick: mfh%cs.brown.edu; S. Zdonik; sbz%cs.brown.edu

In order to support the complex software tools that are needed in these environments, we need a powerful support platform. This platform must be capable of providing the glue that makes these applications function as an integrated unit. Database systems have been successful at providing this service for data processing applications. We strive to achieve the same goals for the domain of interactive design.

We believe that object-oriented databases are a step in this direction. They provide more flexible modeling tools than traditional database systems. They also incorporate some of the software engineering methodologies, such as data abstraction, that have proved to be effective in the design of large-scale software systems.

This paper describes one such system. It further raises a series of issues that must be addressed in building an object-oriented database. It sketches the solutions with which we are currently experimenting and focuses on the implementation of a sophisticated segmenting or data clustering scheme that we are using to achieve acceptable performance.

2. THE DATABASE MODEL

The database system that forms the basis for this work supports an object-oriented model of data [25]. It is in the tradition of much of the work on high-level semantic models [2, 3, 7, 12, 16, 20], but it takes a view of data that is very closely aligned with many of the object-oriented programming languages [1, 6, 10]. It illustrates a new direction in database research characterized as object-oriented databases [4, 5, 8, 9, 14, 22].

In the ENCORE database system [25], all objects are instances of some *type* that describes the behavior of its instances. A type T is a specification of behavior. As such, it describes a set of operations O, a set of properties P, and a set of constraints C that pertain to any of the instances of T. Intuitively, an operation is a program that is used to access or manipulate objects of the given type, a property relates objects of the given type to other objects in the database, and a constraint is a predicate that is used to restrict the legal states of objects. If x is an instance of T, any operation o in O can legally be applied to x, any property p in P is defined for x, and any constraint c in C must be satisfied for x. Types, operations, and properties are all objects in their own right and as such have a type that describes their behavior.

Types can be related to each other by means of a special property called *IS-A*. The IS-A property induces an inheritance relationship between types. If A IS-A B, then all operations, properties, and constraints that are defined on B will also be defined on A. In this case we say that A is a *subtype* of B and that B is a *supertype* of A. The system supports the ability for a type to have more than one supertype (i.e., multiple inheritance). It is possible for a subtype to redefine an operation or a property that is defined on its supertype. In this case, an instance of the subtype will not inherit that operation or property from its supertype.

Operations are active objects that are supported by code. Operation types correspond to a procedure definition, whereas instances of operation types correspond to procedure activations. All operation types have an *invoke* operation defined for them such that it is possible to invoke an operation defined on type

T on any object of type T. Operations are associated with a type. Each type defines a set of operation types that can be instantiated and invoked on its instances. A subtype may add operation types that are not defined on its supertype or may refine some of the operations that are defined on its supertype.

Operation refinement, as defined here, is distinguished from operation replacement, as in Smalltalk. In Smalltalk, a subtype method with the same name as a supertype method blocks the supertype method, thereby replacing it with the subtype method. In our system a subtype may provide an operation type that will substitute for an operation type that is defined on a supertype. Here, however, that operation type must be a subtype of the operation type that is being blocked in the supertype. The name of the refinement need not be the same as the operation type that it is refining. If the supertype A defines an operation f and the subtype B defines an operation g that is a subtype of f, an invocation of f on an instance of B will actually use the operation g.

Properties are objects that are used to relate other objects [23]. For example, a property called *works-for* might be defined on the type *Person*. Works-for would relate a given person to the company object for which he or she works. As a first-class object, it is possible for properties to have properties. A common constraint on property types limits the acceptable values for the property. We will call the set of all legal values for a property p its *value class*. Since properties are objects, there is a type called *Property* that describes how properties behave. There can be subtypes of this type, such as *Single-valued-properties* and *Multivalued-properties*. The first subtype restricts the value of the property to be a single entity, whereas the second allows a property value to be a set.

A subtype may refine a property that is defined by a supertype [23]. Just as in the case of operations, the property type that is defined on the subtype must be a subtype of the property type defined on the supertype.

Object-oriented databases are intended to support the development of large and complex applications. We believe that a strong view of encapsulation is essential for programming in the large. Each type has an implementation that is hidden. The implementation of a type includes a representation for instances and code that implements the operations and properties. Code outside of this type definition cannot access the representation of this type. Type definitions may only use the exported interface of other types. This includes a type and its subtypes. No subtype can make use of the implementation of any of its supertypes, and no supertype can make use of the implementation of any of its subtypes. A subtype may only interact with a supertype through the exported interface, just like any other type.

The concepts described above make up the kernel of the object-oriented database model. We view these as a minimal set of facilities for a database system of this kind. In addition to the kernel, we provide a set of facilities that are, in general, useful for design-oriented applications. These additional facilities are built out of the kernel facilities. The following paragraphs sketch a few of the additional facilities.

The ability to deal with change is one of the foremost requirements of any system that supports design activities. Change can occur at both the type level and the instance level. In order to deal with change at the instance level, we

introduce a version control mechanism [24]. This mechanism introduces two new types, *History-Bearing-Entity* (HBE) and *Version-Set*. HBE defines a set of properties that includes *next-version* and *previous-version*, which are used to express the appropriate temporal relationships between object versions. The *next-version* property can be multivalued, thereby allowing a given version object to have multiple successors. We call any version that is the value of a *next-version* property with cardinality greater than 1 an *alternative*. Any other type *T* can be defined to be a subtype of HBE (as well as any other logically related types), thereby giving instances of T the ability to record versions. *Version-Set* is a type that is used to collect all of the versions of an individual. It has an insert operation that can only add new versions at specific points in the version history. New versions can only be added at the end of a version sequence or as an alternative to an existing version.

The user-level transaction mechanism that is built on top of the kernel makes use of the version control mechanism. A transaction can add a new version to each of a set of version sets (i.e., its write set). This set of changes is called a *slice*, and only slices can be undone. The slice corresponds to a single atomic action, and undoing it corresponds to nullifying the effect of that transaction.

Since types are objects, we can use the version mechanism described above to keep track of changes to types. Each object retains its connection to the original version of the type under which it was created. If one needs to treat an object that is an instance of an old version of a type as if it were an instance of a newer version of that type, we use an exception-handling scheme [17, 18] to facilitate this operation. This scheme works equally well for the case in which we want to treat an object that is an instance of a new version of a type as if it were an instance of an older version of that type. We do not directly support the conversion of instances of old types to conform to new-type definitions. This process can cause old programs to stop working, is often very expensive, and, in some cases, loses information.

An object-oriented database needs to be able to model composite objects, that is, objects that are made up of other objects. In our view, there is a special property called *part-of* that is used to express this relationship. *Part-of* is a subtype of the type *Property*. The *part-of* property has special semantics. It is used by some operations to perform an action on an object and all of its pieces. An example of this is locking a whole object (e.g., a report) for update. This type of lock would first lock the high-level report object and then lock all other objects that are in the transitive closure of the *part-of* property. It is also used in the context of version sets to support version percolation [24].

It is important to realize that the model of data described above is part of a database system. As such, it governs the way in which persistent, sharable objects behave. Our system also addresses database notions of transaction, consistency, associative retrieval, and views.

It is also important to point out that in traditional data models there is always some fairly high level of abstraction below which programmers cannot have access. For example, with relational systems it is typically not possible to reprogram the basic file structures that are used to implement relations. In our view, an object-oriented database system should allow users to program at

whatever level best suits their needs. Everything is represented by types, and all types exist at the same level. Levels of abstraction will certainly be encouraged, and, in fact, the system provides several very high-level abstractions such as version sets. However, the very lowest level types, like the type *byte-string*, are available to programmers to build their own abstract types as needed. This does not, however, mean that data abstractions can be compromised. This is critical in an environment like CAD where performance is key. Programmers have the choice of using the system-provided higher level abstraction, or, for cases in which the performance of these types is not adequate, they may choose to create structures that fit their application more closely.

3. THE ARCHITECTURE

The main focus of this paper is on the storage management aspects of an efficient object-oriented database. To achieve a better understanding of some of our choices, we also describe how the storage management function fits into our overall architecture. The rest of this section describes the main system modules and the way in which objects are mapped through the various levels of abstraction.

3.1 The Module Structure

The database system is decomposed into two distinct subsystems. One subsystem is a typeless backend that is responsible for managing the use of the persistent object store, and the other piece is responsible for the enforcement of the type system.

The OBject SERVER, known as ObServer, reads and writes chunks of memory from secondary storage. These chunks are used by the higher level module to store the state of objects. ObServer also has a primitive notion of transactions, which includes a subset of Moss's nested transactions [11]. Through the transaction mechanism, it is possible to lock and unlock objects to ensure an appropriate level of noninterference. The transaction mechansim can be used in a way that provides for resilient storage in that if it is used properly, it will not allow the changes of an aborted or crashed transaction to be permanently installed in the database.

The transaction scheme makes it possible to support a variety of shared memory applications. The server is currently being used at Brown University for two distinct purposes, the backend of an object-oriented database system and the storage system for an object-oriented, interactive programming environment. Other examples of systems that could be implemented on top of our server include mail or blackboard systems.

The type level is normally referred to as ENCORE (Extensible and Natural Common Object REsource). It is this level that deals with the semantics of objects through type definitions. This higher level module supports the type system that was described earlier as the ENCORE data model. It should be noted that ObServer can support other type systems as well. For example, the GARDEN programming environment defines its own type system, yet it uses the facilities of ObServer to store its persistent objects.

The type level communicates with the server through the UNIX[1] remote procedure call (RPC) mechanism. The communication channel is asynchronous in the sense that ENCORE (or any application process) sends messages to ObServer requesting services and does not wait for ObServer to reply. When ObServer replies, ENCORE may or may not choose to service the reply message.

A client sends a request to the server and is not suspended while the server processes the request. At a future time, the client takes reply messages from its message queue. Reply messages may be delivered soon after a request has been made or after some delay. For example, lock requests may be granted immediately, or they may wait in a queue and be granted or denied later. Similarly, messages for a client regarding changes in objects on which the client holds notify locks arrive periodically from the server.

3.2 Server Overview

The server is a resource for any application system that needs to manage chunks of memory allocated in a shared memory space. Here, a chunk is any contiguous string of bytes. The server must allocate space and a Unique IDentifier (UID) for each chunk that it stores. The UID is similar to a laundry ticket that is given out when the object is stored and that guarantees delivery of the same object when the UID is presented to the server. One of the principal functions of the server is to maintain the correspondence between UIDs and chunks of memory.

The setting for our system is a network of workstations (i.e., nodes), each running independent processes. We have adopted a model in which a server and its data reside on a single node. It is possible for processes on other nodes to access this server. Concurrent access to the shared memory is accomplished by means of UNIX remote procedure calls to possibly remote UNIX processes. The server also supports transaction processing in a manner that is flexible enough to handle long, interactive transactions, as well as the more traditional type. The nested transaction processing facility supports atomicity and recovery and deadlock detection. Our approach to locking has several novel features that are discussed in a later section.

Each process that wants to communicate with the server must bind a module called the *client* into its image. It is, therefore, possible for the client and the server to reside on different machines. When a process needs to request a service from the server, it makes a call on the client code that hides the details of the RPC interface. The ENCORE module uses the object server as a backend. It makes calls directly on its own copy of the client module. Notice that if there are two different processes on two different machines using the ENCORE database, separate copies of ENCORE must reside on each machine (see Figure 1). As will be seen (in a future section), we can achieve some performance enhancement by making the client an intelligent partner in the communication. It can often make certain decisions locally, thereby minimizing the amount of communication.

[1] UNIX is a trademark of AT&T Bell Laboratories.

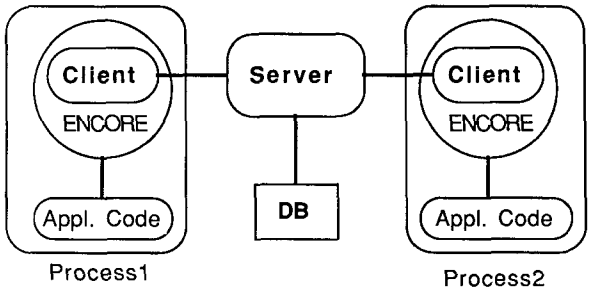

Fig. 1. The basic module structure.

3.3 ENCORE Overview

The chunks of memory that are managed by the server can be used to implement type objects as presented by the ENCORE interface. In an object-oriented database the type lattice introduces the problem of an object's being an instance of more than one type. If we have the type *Toyota* as a subtype of the type *Car*, then an instance x of the type *Toyota* is also an instance of the type *Car*. Since our system enforces a strong notion of data abstraction, there will be a chunk of storage that represents the part of x that is an instance of *Toyota*, and a chunk of storage that represents the part of x that is an instance of *Car*. We use the term *instance* to refer to each chunk and the term *object* to refer to the aggregate of all instances that make up x.

The system deals with object creation and modification in a way that is designed to optimize its interaction with the file system and the RPC facility. The reading and writing of objects is done on a block basis. That is, the application may request that an aggregate of UIDs be read or that a collection of objects be written in a single interaction with the server. This generates only one IPC transfer and also allows the server to optimize the way in which it interacts with the file system. Upon object creation, UID allocation is separated from storage allocation. This allows an application to request UIDs in anticipation of their use without reserving space for them in the file. Space is not allocated until objects are actually written.

3.4 Multiple Databases

In order to allow multiple databases to be accessed, we have adopted a scheme by which a separate *binder* process provides a client with a connection to the desired database. The interaction between the client, binder, and server allows both the creation of new databases and a connection to existing databases.

Each database that is being accessed will have a separate server process that mediates its requests. When a client wants to access a database, it issues a request to the binder. The binder returns enough information for the client to connect to the appropriate server. All further requests from the client will subsequently go directly to that server. The requests to a given server can come from any of several clients that are possibly on different machines.

3.5 Transaction Management

The transaction-processing facilities of the system were designed for transactions that are potentially long, interactive processes controlled by a user who is sitting at a workstation. Conventional transaction-processing schemes are designed for relatively short transactions that are implemented by a program. For this reason, we have made choices that are different from what one might expect of a database transaction facility. Our general philosophy is to provide the proper level of primitives so that applications built on top of our system can present the transaction mechanism that best suits their environment. For example, using our system it is possible to build a set of transactions that are serializable. It is also possible to use the primitives in a way that does not make such a strong guarantee about the results of a set of concurrent transactions.

A later section of this paper focuses on the transaction-processing capabilities provided by ObServer. ENCORE can make use of these primitives to construct transaction mechanisms of its own. We would model ENCORE transactions as instances of a type called *Transaction*. The type *Operation* would be a subtype of Transaction. We have not included definitions for the Transaction type in the current ENCORE kernel. If this type were to be built, it would make use of the ObServer facilities and may choose to let some or all of the transaction facilities show through.

3.6 Storage Mapping

ENCORE deals with abstract objects that are instances of types. These types participate in inheritance relationships and allow for the implementation of an object to be distributed across several type definitions. How are these levels of abstraction mapped onto the basic storage structures provided by ObServer?

At the type level, every object might consist of several instances, one for each type in which it participates. For example, if *Toyota* is a subtype of *Car*, *Car* is a subtype of *Vehicle*, and *Vehicle* is a subtype of *Object*, then a given Toyota will be an instance of all four types. Since each type has its own private representation, as required by our abstract data type scheme, the Toyota object would need four chunks of storage for its representation. Each of these chunks would be accessible through the operations of the corresponding type.

We must next ask how these chunks (i.e., one for each instance) are held together. A single UID is associated with each object. When a UID is dereferenced, it leads to a header block for that object. Conceptually, the header is a part of the chunk for the instance of type Object that every object must have. The header for object x contains some general bookkeeping information, as well as a set of pairs of the form (t, p), where t is a pointer to a type object, and p is a pointer to the beginning of the chunk that holds the representation for the instance of t that is a part of x.

Most often, these chunks are allocated contiguously such that the pointer p is the offset into that contiguous storage at which the chunk for t begins. In this case there would be a single UID for the large chunk that contains the instance chunks. This UID is the one that is used by ENCORE to represent object identity.

It is also possible for the chunks to be noncontiguous. Since *p* can be a UID, the chunks can be stored in any physical location. This allows for a vertical partitioning scheme in which instances of different types for the same object can be stored in different storage areas. The decision to perform this type of partitioning would depend on the access patterns for objects of the given type.

Every chunk that is stored by ObServer has an ObServer-level UID. Only some of these are exported by the ENCORE interface to application programs as object surrogates. If one of the internal pointers that binds together the type instances for an object is a UID, then this UID is never available to be passed to application programs. It is useful to allow ObServer to find the chunk, but since it does not represent a whole object, it has no semantic meaning at the ENCORE level.

Once we have an object decomposed into its proper storage pattern such that the chunk or chunks contain all of the necessary instance blocks, we can use ObServer to store those chunks with the appropriate UIDs. Notice that, if all the instances are stored contiguously, there is only one chunk to store, and the UID of the instance of Object is used.

4. SEGMENTS

In an environment in which many objects must be frequently accessed, efficiency becomes a principle design criterion. One approach to improving performance in a database involves clustering groups of related objects on the disk. The *segment* provides this facility. A segment contains objects that the object-oriented database management system expects a client to access during a transaction, thus eliminating frequent diskhead motion and single object transfers. Thus a segment clusters a logically related set of objects into a variable-sized single package. Since we expect a client to access other objects in a transferred segment, greater system performance results from preloading required objects. A segment is the unit of transfer for objects between client and server and from secondary storage to main memory.

Segment objects are only read or modified through the segment operators: install, find, update, delete. Find, update, and delete have the conventional meaning. Installing an object refers to inserting an object into a segment. Migrating an object involves deleting an object from one segment and installing it into another.

Once a client receives a segment, the objects are individually placed in an object hash table and the segment is freed. The client has no further use for the segment structure once it has acquired its objects. Part of the gain in performance involves the placement of objects on disk. Storing a segment's objects contiguously on disk allows faster disk access, since the segment may be read into main memory without random diskhead movement. Since the UNIX file system does not guarantee contiguous storage of segments, ObServer employs its own file mechanism.

The server receives a set of object changes from the client containing a client's operations (install, update, delete) and other information necessary (e.g., the object) to install the changes in the server's copy of the segment. By returning only the final changes to the server in one package, we minimize the amount of network traffic and reduce server processing. If changes are transmitted individ-

ually, the server not only installs the changes but must access the communication network for each change. It may seem that the entire segment should be sent to the server, thereby eliminating having the server install client changes into the server's segments. Since we wish to allow many clients to use copies of the same segment, the server would then have to merge the returned segments, which is a much more costly operation. As a result, our segment becomes a unidirectional unit of transfer, in the direction of the client, for reducing communication from the client to the server when objects are requested.

It is useful for a client to create different segment groupings or contexts in which to work. This allows individual segment sizes to remain small and at the same time define larger working sets above the level of transfer. To allow clients to retrieve different sets of related segments, we introduce the *segment group* (SG). To reference an SG, a unique name is assigned by the client creating the SG. This notion of grouping a set of segments facilitates having small, very strongly related sets of objects in segments, while allowing several alternative larger groupings to be specified. All segments are themselves uniquely named SGs, and an SG contains one or more SGs. Reading an SG provides a set of segments. As an example, Figure 2 illustrates nine segments that are involved in various groupings. Reading SG4 provides segments s2, s3, s4, s5, and s8. As indicated in Figure 2, a given segment may occur in several segment groups. Each database maintains its own SG forest, and an SG may only contain the segments within a database.

When a client requests an object, the server returns the segment s in which the object resides. The client may further specify an SG that indicates the context in which it is working. The SG may be selected by the ENCORE module on the basis of knowledge about how types are used and storage pragmas. In this case the server returns the other members of the SG asynchronously, while the client is working on the objects contained in the original segment s. This provides another level of preloading that can occur in the background.

4.1 Object Access

The object server maintains *master segments* containing the current versions of all objects resulting from committed object changes. A client obtains from the server *copy segments* that the client accesses locally. Clients may share the same copy segments by each having a copy at their location; however, object locks may prohibit specific object accesses.

Whereas segments provide access to objects in groups, the *unique identifier* (UID) provides individual object access. Our segmentation scheme employs two types of UIDs: *external* and *internal*. An external UID provides a user with a constant reference to a database object. When the server dereferences a valid external UID, there results an internal UID, manipulated by the system to locate an object physically. Both internal and external UIDs have the same length, but their internal structures differ. Each external UID maps either directly or indirectly onto one or more internal UIDs. A mapping to multiple internal UIDs results from replicating objects (discussed below). The server sequentially allocates external UIDs that are not recycled when objects are deleted. Deleted objects have external UIDs that map to a *tombstone* internal UID. This makes it

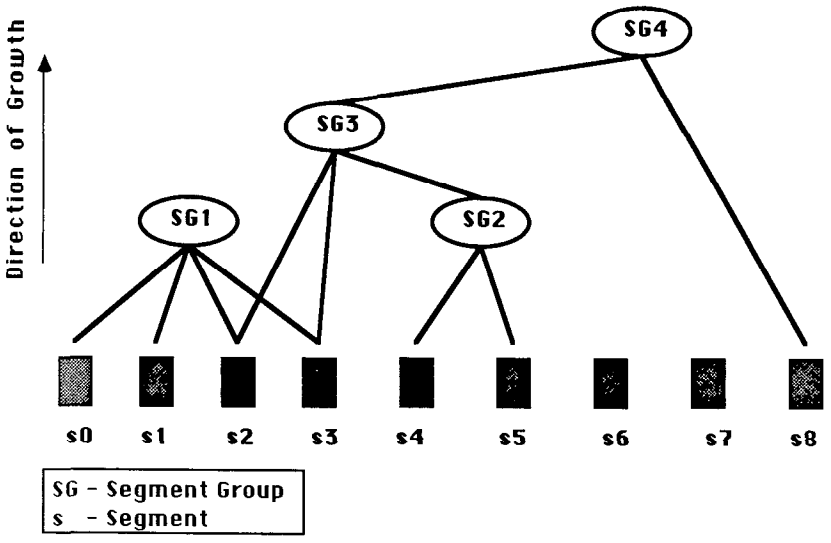

Fig. 2. Example segment groups.

possible to detect a reference to an object that no longer exists. Figure 3 depicts the dereferencing process from an external UID to an object. The various mappings are maintained in files called the *Object Location Table* (OLT) and *Duplicate Object Table* (DOT). In Figure 3, the *code* field in the UID structure indicates the UID type, either internal or external. This information is used in both the client and server processes. The OLT maintains the external-to-internal UID mapping. The DOT is described in more detail in the next section.

4.2 Object Replication

In most clustering schemes, it is only possible to place an object in one group. The case may arise in which there is more than one reasonable way to cluster a given object. To resolve such conflicts, we provide an object replication facility. This scheme, of course, incurs a penalty for update but is extremely useful for objects that are either seldom updated or read only.

The implementation of replicated objects requires the introduction of a level of indirection between the external UID and the internal UID. Here, an external UID maps to an index in the *Duplicate Object Table* (DOT) that is maintained by the server and provides the internal UIDs with all copies of a replicated object. When dereferencing an external UID that maps to a replicated object, the system checks whether a client already has a segment containing the object. If so, the corresponding internal UID is returned.

Updating a replicated object is a more costly operation, since the server must update the object in each segment containing a copy. However, the decision to maintain multiple copies of an object rests with the database designer at the time the object is created. The system guarantees that the update of all copies of a replicated object occurs atomically. Thus a client cannot obtain a segment that contains a duplicate copy of x until all segments containing x have been updated.

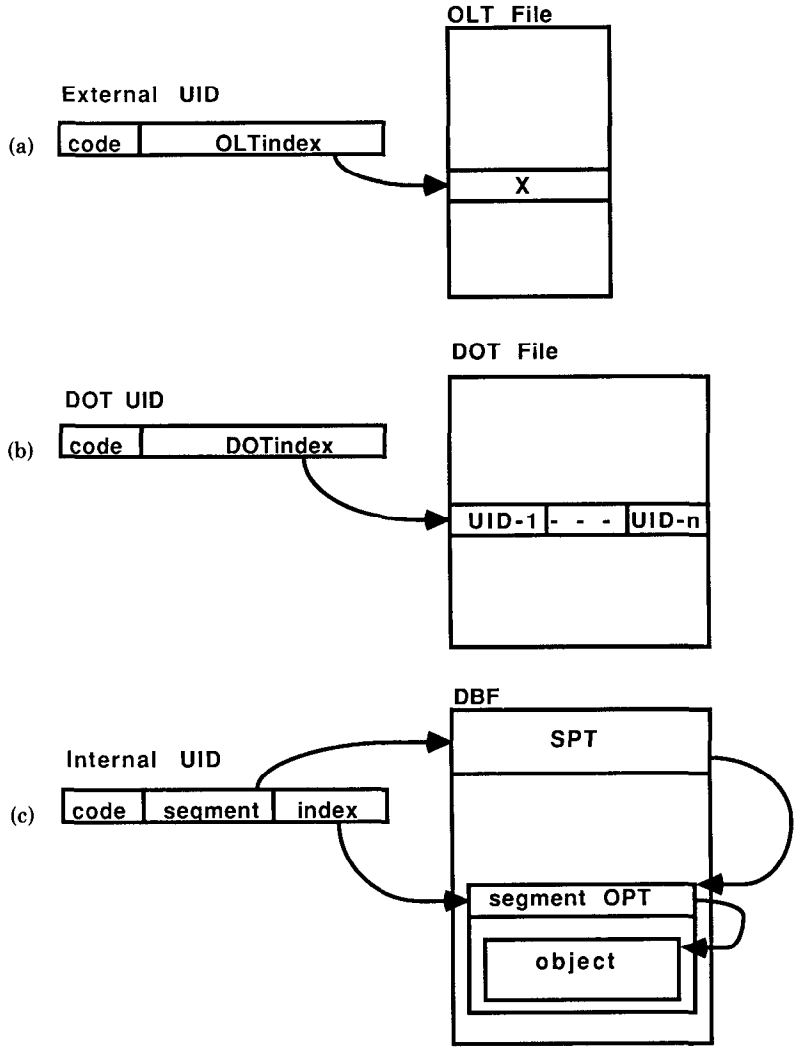

Fig. 3. Server UID to object mapping. In (a) X is either an internal UID or a DOT UID. In (b) one of the n internal UIDs is selected on the basis of the status of the corresponding segments. In (c) the segment field corresponds to an index in the SPT. The index field corresponds to an object within the given segment.

Object-level locking, however, introduces a problem with updating replicated objects. If two clients, C_a and C_b, have copies of the same segment, and C_a updates an object that C_b will use after C_a commits, C_b now has an outdated copy of the object in its address space. To solve this problem, the server generates new timestamps for each object in the transferred segment and for the segment itself. If other clients have copies of the same objects, the update of these objects by any client causes new timestamps to be associated with them. The server determines whether a client has an old copy of an object in its address space by

comparing the timestamp of the segment (when it was transferred) against that of the object. Timestamps are kept only for objects in use rather than for all database objects to reduce the amount of space required for timestamps in general. If a client tries to lock an object contained in a copy segment and the object is an old copy, the new object is sent to the client.

4.3 Clustering Objects

The segments collectively provide a *partition* of the objects within a database.[2] All database objects are contained in at least one segment. A *Database File* (DBF) represents a separate and independent set of objects and type specifications. It is often useful to partition objects by means of semantic properties. Some options for placement are the following:

—*One object per segment* is intended for very large objects, since they are costly to transfer and tend to be accessed individually.
—*Storing an object with its subobjects* transfers a package of related objects that are almost always accessed together.
—*Storing all instances of a type together* is used to satisfy queries requiring the search of all objects of a type.
—*Partitioning based on property values* is similar to indexing. In using properties, specific values, such as "red," or numeric intervals, such as $0 < n < 3$, may be specified. This method allows a client to separate objects containing a property value of particular interest into one segment.

At the discretion of the designer, any of these methods may be selected to tailor object placement to expected needs. Establishing an initial partition of objects, either through direct client specification (e.g., place object x into segment y) or by semantic criteria, a client may update an object, causing the original object placement to hold no longer. This mainly affects partitioning by property specifications. For example, changing an object's color from red to blue may violate the original specification of a segment whose objects were to have the COLOR property value of "red." We resolve this conflict of an inappropriate property value by specifying the *strictness* with which a segment adheres to the original specification. A segment designated to hold blue objects only holds blue objects if the segment is labeled as *strict*. If the object is updated with a property value violating the segment's strict specification, the object is moved to another, more appropriate segment. A segment labeled *nonstrict* accepts the appropriate objects when they are initially installed. However, updating the object does not cause the object to be moved out of the segment. Segments created by our automatic partitioning mechanism, the *Object to Segment Mapping* (OSM), have the segment specifications strictly enforced. Segments explicitly created by clients are labeled as nonstrict by default, yet may be altered by the client.

[2] Our use of the term partition does not imply mutually exclusive sets of database objects but a lower level clustering of both replicated and nonreplicated objects.

4.4 Segment Structure

A segment contains a pointer table and a set of objects. Each segment object is referenced by exactly one entry in the pointer table. Segments are stored in a *Database File* (DBF). The DBF structure is similar to that of the segment: a pointer table and a set of segments. The pointer table allows a reference to an object (or segment) without knowing its exact position. This makes it possible to move objects (or segments) within a segment (or DBF). The pointer table comprises one or more *pointer table blocks*, and additional fixed-size blocks are inserted as a segment acquires more objects. This feature reduces the frequency of segment expansion each time an object is installed. Figure 4 depicts the DBF and segment structures.

A DBF contains the number of *Segment Pointer Table Entries* (SPTEs), the *Segment Pointer Table* (SPT), and segments. The number of SPTEs represents the next available segment index to allocate. Each SPTE is composed of an *offset* and a *size*. The offset specifies the segment location within a file, and the size specifies the number of bytes occupied by the segment. The SPT index serves as the segment identification number and does not change for the life of the segment.

A segment in secondary storage likewise contains three sections: the number of *Object Pointer Table Entries* (OPTEs), an *Object Pointer Table* (OPT), and objects. The number of OPTEs represents the next available object index to allocate. Each OPTE contains an *offset*, *size*, and *OLTindex*, (Object Location Table index). The offset and size are the same as for the DBF. The OLT index provides a back pointer to the OLT that facilitates object migration.

Overflow blocks are main memory addenda to the segment structure. Upon the opening of a segment, space is allocated in main memory for the exact size of the segment. As new objects are installed or existing objects expanded, overflow blocks are allocated separately from the main segment. To reduce the frequency of creating overflow blocks, the system allocates enough memory so that several objects may fit in the same block. The allocation size is determined by a factor multiplied by the size of the first overflow object. Overflow blocks eliminate copying a segment each time objects are installed. Objects in overflow blocks are accessed as though the segment and overflow blocks were contiguous in main memory. When writing a segment back to its DBF, a new segment space is allocated in the file to reflect changes in the segment's size. If overflow blocks exist for a segment in main memory, the segment is first compacted in memory and then written to the disk.

The *Main Memory Segment Table* (MMST) contains information about an open segment throughout its duration in main memory. System routines referencing a segment use an MMST node as a *handle* for segment access. Upon opening a segment, an MMST node is created, initialized, and inserted into the MMST hash table. An MMST node maintains the overflow block information as objects are installed. Independent MMSTs are maintained at each client and at the server.

Object structure depends on the user-defined type specification, but this does not affect the object server since ObServer handles an object as a string of bytes when installing and retrieving objects.

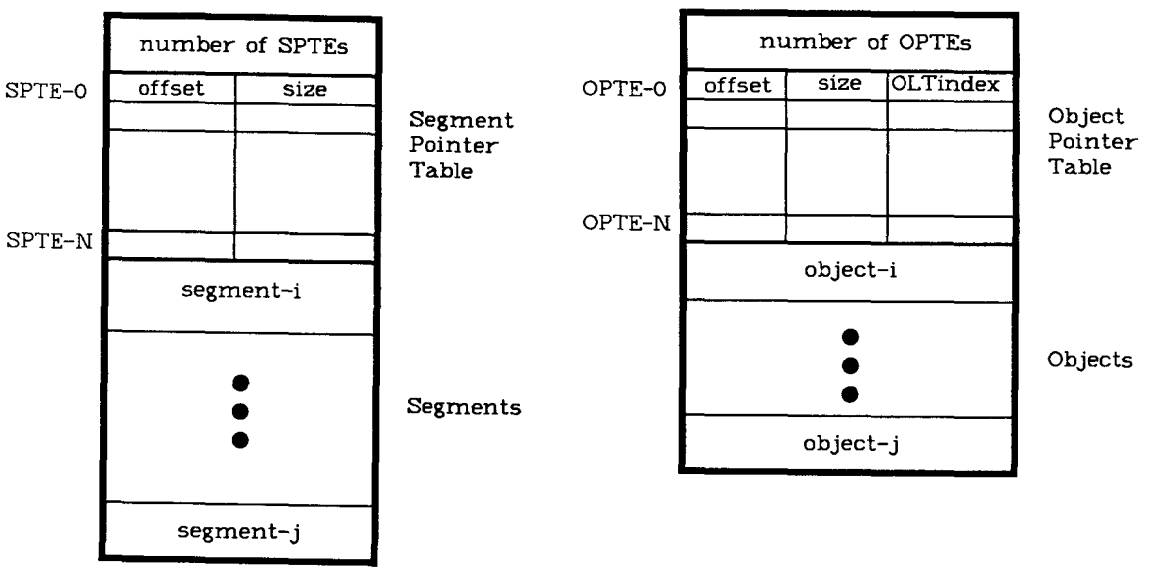

Fig. 4. Database file and segment structures.

Table I. Lock Modes and Compatibilities

	Lock request				
Current lock	NULL	NR-READ	R-READ	NR-WRITE	R-WRITE
NULL	T	T	T	T	T
NR-READ	T	T	T	T	T
R-READ	T	T	T	F	F
NR-WRITE	T	T	F	F	F
R-WRITE	T	F	F	F	F

5. SHARING

5.1 Lock Types

In conventional database systems, the lock set contains the generic read and write locks with well-defined protocols for their use (e.g., two-phase locking). The conventional lock types and protocols are too restrictive for the design environments that we want to support. For example, two-phase locking and serializability prevent a transaction from seeing the intermediate results of another transaction. Several designers who are in the middle of a design transaction (e.g., editing session) may need to share uncommitted results with their co-workers. Our objective is to provide a comprehensive lock set that allows users to define new protocols freely and easily.

We have identified two dimensions for lock definitions: *lock mode* and *communication mode*. Our scheme employs five lock modes, NULL, NR-READ, R-READ, NR-WRITE, and R-WRITE. NR stands for nonrestrictive and R for restrictive, in the sense of what they allow and disallow. The above ordering of the lock modes indicates their respective strengths from least to greatest. Their compatibility is specified in Table I.

The NR-READ lock mode allows a client to read an object without prohibiting the access privileges of other clients. The R-READ lock mode restricts other clients from writing to an object for the duration of the lock. The NR-WRITE lock mode prohibits other clients from obtaining R-READ or R-WRITE lock-mode locks but allows the reading of an object through the NR-READ lock mode. The R-WRITE lock provides a client with exclusive access to an object, which in essence removes the object from the database while the lock holder uses the object. This lock type is particularly useful when an object or operations on an object are malfunctioning. As an example, consider an operation that inadvertently overwrites random elements in main memory. To prevent further damage as a result of other clients invoking the operation, a system programmer wishes to stop all access to this operation while it is being updated. The lock mode NULL is useful when specifying soft locks (see below) or in conjunction with the communication-mode dimension.

The communication-mode dimension refers to communication between clients as the result of another client's action. Lock holders may wish to be notified of the status of an object, including requests from other clients for that object or committed updates from another client. The five communication modes are

Table II. Locks and Validity of Mode Combinations

	Communication modes				
Lock modes	U-NOTIFY	R-NOTIFY	W-NOTIFY	RW-NOTIFY	N-NOTIFY
NULL	V	I	I	I	I
NR-READ	V	I	V	I	V
R-READ	I	I	V	I	V
NR-WRITE	I	V	V	V	V
R-WRITE	I	V	V	V	V

Note: V = valid; I = invalid.

U-NOTIFY—notify lock holder upon update, *R-NOTIFY*—notify lock holder if another client requests the object for reading, *W-NOTIFY*—notify lock holder if another client requests the object for writing, *RW-NOTIFY*—notify lock holder if another client requests the object for reading or writing, and *N-NOTIFY* indicating no notification.

By taking the cross product of the lock modes and communication modes, 25 locks result, as depicted in Table II. Of the 25, 11 are nonfunctional, in that the lock mode prohibits the associated communication mode. As an example, consider the NR-WRITE/U-NOTIFY combination. An object locked as such could never be updated while the NR-WRITE was held; hence, notification on update is meaningless. By using various subsets of the remaining 14 locks, applications from cooperative programming design environments to those requiring full serializability may be satisfied.

As an example, the GARDEN [13] system currently uses a hybrid lock, *WRITE-KEEP* [19] that, among its other semantics, informs the owner of the lock of other clients' lock requests on the locked object. This WRITE-KEEP lock is used in conjunction with a NOTIFY lock. From our lock set, the subset NR-WRITE/RW-NOTIFY and NR-READ/U-NOTIFY provides the same functionality. GARDEN uses a NOTIFY lock in place of a read lock and a WRITE-KEEP lock in place of a write lock. This establishes a demand-driven communication scheme. When an object that GARDEN has read is updated, GARDEN is notified, and it can reread the object if necessary. When an object that GARDEN has for writing purposes is needed, GARDEN is notified, and it can return the object if desired.

For a more traditional lock environment, the subset R-READ/N-NOTIFY, NR-WRITE/N-NOTIFY provides the basics for serializable transactions when used with the proper transaction options (see below). By allowing a rich lock set, applications may tailor a locking environment to their requirements for sharing. In an interactive, cooperative design environment, one may wish to employ only locks with the RW-NOTIFY communication mode. Lock holders, not wishing to impede other client's productivity by keeping locks on objects not currently in use, are notified of other client's lock requests. Consequently, the lock holder may opt to free the lock or commit the current object changes, thus allowing the other users to lock the object.

As another example of the use of the communication mode, consider several transactions cooperating on a task, each of which has an object x displayed on its screen. Any one of the transactions is allowed to change x, but the others

would like to update their screens when this occurs. If all transactions hold a NR-READ/U-NOTIFY lock on their objects, then one of the transactions may convert its NR-WRITE lock to a N-NOTIFY lock. When it writes x, the other transactions will be sent a notification of the change, and they can reread x and reset the new value on their displays.

A rich lock set provides flexibility to the user but also greater responsibility to select a lock subset and consistently use locks from that subset. A lock subset is equivalent to the complete lock set with the undesirable lock types filtered out, so we now introduce a lock *filter*. Lock subsets are specified by selecting from one of the system-predefined filters (e.g., to ensure serializable transactions) or by dynamically creating such a filter. Filters guarantee that the specified lock environment is maintained and allows only the permissible lock requests to reach the server. Hence, the user has the ability to tailor an environment with the exact lock desired. A client may set a filter as part of a normal transaction operation. Later versions of the system will allow a database server to have many filters that apply to various user categories (e.g., read-only users or full-privileged users).

5.2 Processing Lock Requests

The server interacts with the client to process both object and nonobject operations and maintains the files necessary for accessing master segments (see Section 6). Commands processed from clients involve DBF operations such as create and open, segment and segment group operations and lock requests. In satisfying client lock requests, the server must determine three things: (1) Does the client already have the object requested in the client's copy segment? (2) Does the client have the most recent copy of the object? (3) If the client has neither (1) nor (2), which segment should be sent to the client? We assume that a client locking an object does so only if it intends to use the object within the current transaction. Therefore, the object is sent in its segment if the lock is granted. The server answers the first question by checking the *Client Segment List* (CSL) maintained at the server for all clients. If the client already holds the required segment, the client receives only the object-access information. Otherwise, the client acquires an appropriate segment from the server. If the client has an outdated local copy of the object as determined by the object timestamp, the server sends the current copy of the object to the client and replaces the local copy.

5.3 Deadlock Detection

Deadlock can occur whenever two conflicting locks have been requested on the same object from two distinct transactions. This situation requires that one transaction wait until the other commits. In our model, the lock compatibilities that conflict are given in Table I. Our definition of deadlock is somewhat different from the usual definition. The server allows certain cycles to remain in the waits-for graph. Here, a deadlock requires that the deadly embrace be between two transactions that have actually been granted locks for each other's objects. If there is a cycle between two transactions that are queued, this cycle is allowed to remain. The system will not grant a lock request if that request would cause a

deadlock. This potential for deadlock could happen at the time that the lock is requested, or it could happen at the time that a pending request is finally serviced (i.e., removed from the queue). Queued lock requests are serviced whenever a transaction commits or aborts and its locks are released, making them available to other waiting transactions.

Consider the following example. Transaction T_c has objects x_1 and x_2 locked in write mode. Transactions T_a and T_b request locks for x_1 and are queued in that order. Similarly, transactions T_a and T_b request locks for x_2 and are queued in the opposite order. The system will not consider this to be a deadlock, since T_a, T_b, or both of them may abort before T_c completes. If they do not abort, the system will detect deadlock when T_c completes, and either T_a or T_b will be informed that its lock request has been dequeued. Notice that in an interactive environment it is important that a deadlock not cause a transaction to abort.

6. LOCK AND SEGMENT INTERACTION

In our segmentation scheme, we present two levels of granularity for locking: *object-level* and *segment-level.* Locking at the object level implies that a client must request locks on individual UIDs, whereas when locking at the segment level, the client could lock all the objects in a segment with a single specification. Note that, locking an object by its UID locks all copies of the object. Since our system allows replicated objects to reside in separate segments, it would be possible with segment-level locking to lock large sections of the database by locking a single object. In the general case concurrency among clients is significantly reduced with segment-level locking, since a client using a single object in a segment prohibits other clients from obtaining other objects in that segment.

Locking purely at the object level allows clients to share segments, thus increasing concurrency. Our system supports object-level locking with the additional facility for locking all objects in a segment easily. This is more efficient from the user's standpoint in that procuring locks on all segment objects at the outset eliminates having the client repeatedly ask the server for locks or individually specify locks for each object in a segment. To lock all the objects within a segment, each object must acquire the desired lock. Locks that cannot be granted are queued, and the segment is sent with the objects that have acquired their locks.

Recall that our segmentation scheme sends a client an entire segment when the client requests even a single object. This clustering provides objects expected to be used by the client during a transaction. Because client/server interaction is asynchronous, frequent requests to the server for locks impedes client productivity, since the client must wait for the server to reply. Frequent requests also increase the work load at the server, hence reducing performance for all clients.

The server makes a distinction between objects explicitly requested by a client and objects in the remainder of a segment. Objects explicitly requested use *hard locks*, and the remaining segment objects in the segment use *soft locks*. In specifying a lock mode for the remainder of objects in a segment, the client does not know exactly which objects it will be getting. Therefore, we view soft locks as a convenience rather than a necessity (from the client's point of view). If a hard lock cannot be granted, it is queued; soft locks are not queued. The client

requesting a hard lock is notified whether the lock was granted or denied, but is only notified if the lock was granted for soft locks. Using soft locks, we reduce the size of the lock queue and minimize the amount of information returned to the client.

When requesting objects, the object and segment-lock specifications need not be alike, and all locks from the current lock filter are valid for either. A client specifies a lock request as a quadruple: the object UID to be locked, its lock, a segment, and the lock for the remainder of the objects in the segment.

7. TRANSACTION MODEL

The traditional transaction model [6] has well-defined features, such as two-phase locking, that arise from the guarantee that all transactions are atomic. The measure of correctness here is that all resulting schedules are serializable. We attempt to identify the basic building blocks that can be used to build all interesting transaction schemes. We begin by defining a transaction as a series of operations that occur during some period of time in a *well-defined* frame, that is, a frame that is marked by specific delimiters (e.g., *begin* and *end*). These transactions may be nested [11]. All operations occur during a transaction and are associated with an individual transaction. Since transactions at the same client may have different restrictions and allowable lock modes, the operation must be screened to determine whether it is using a validly locked object for that transaction and whether the operation itself is valid (e.g., unlocking an object in the middle of a transaction).

We introduce a set of constraints that may be applied to the skeleton transaction to tailor it to the environment in which it is used. The two essential facilities that a transaction provides are acquiring or releasing a lock and making changes visible (i.e., committed).

With respect to locking, unlock all objects associated with a transaction when it ends with the following two options:

—allow explicit unlocking of objects during a transaction,

—disallow explicit unlocking during a transaction.

With respect to committing, atomically commit all object changes at the end of a transaction with the following two options:

—allow explicit committing of objects during a transaction,

—disallow explicit committing during a transaction.

In general, all explicitly committed object changes are made visible and cannot be aborted. A transaction may be aborted at any time such that any changes not committed are thrown away and all objects are unlocked. From these building blocks, many transaction environments may be created. It should be obvious how the conventional transaction model can be created out of these options.

8. AN EXAMPLE

Figure 5 is an example depicting the role of hard and soft locks, the reduction of communication between the client and server as a result of segment transfers, and the general procedure for accessing objects. Note the following abbreviations:

LOCK MODES	COMMUNICATION MODES
0—NULL	0—N-NOTIFY
1—NR-READ	1—U-NOTIFY
2—R-READ	2—R-NOTIFY
3—NR-WRITE	3—W-NOTIFY
4—R-WRITE	4—RW-NOTIFY

SERVER STRUCTURES

Master Segments:

Segment S_x contains object O_y

S_1: O_1, O_2, O_3, O_4

S_2: O_1, O_5, O_6

S_3: O_3, O_7

Lock Table:

Transaction T_z locks object O_y with lock $L_{i,j,k}$

T_1: $\langle O_1, L_{3,2,H}\rangle$, $\langle O_5, L_{2,3,H}\rangle$, $\langle O_6, L_{2,0,S}\rangle$

T_2: $\langle O_3, L_{1,0,H}\rangle$

Client Segment List

Client C_w has segment S_x

C_1: S_2

C_2: S_3

Figure 5

S, segment; O, object; T, transaction; C, client; $L_{i,j,k}$, $0 < i < 4$ lock modes, $0 < j < 4$ communication modes; k: H hard; S soft.

The following are a subset of the client operations that are relevent to this example: LOCKquery (O_y) returns the type of lock $L_{i,j,k}$ currently held on the object or a NO-LOCK signal if no lock is held. LOCKobjects (O_y, O, S_x, S) make the request for object O_y, with the lock O-$L_{i,j,k}$ and lock the accompanying segment S_x with the lock S-$L_{i,j,k}$. This operation informs the server of the request to which the server later responds. The client then takes the segment and the corresponding object-access information into its address space. OBJECTread (O_y, buffer) finds the object O_y in the client's address space and places it in the provided buffer. Not finding the object results in an error signal. SEGMENTfind (O_y) is separate from the standard routines in that it maintains a mapping, possibly at the type level, of where objects are stored. Hence, it determines which segment to request. This provides the client with an *intelligent* segment choice rather than a less informed choice from the server.

From here, client C_2 in transaction T_2 requires object O_2 and checks whether it has a lock on it by making the local procedure call LOCKquery (O_2). Since it does not have the lock and is interested in working on other objects in a specific segment, T_2 finds the segment it wants by the call SEGMENTfind (O_2) and makes a lock request LOCKobjects (O_2, $L_{3,0,H}$, S_1, $L_{2,0,S}$), asking for a specific object O_2 and segment S_1. If the client had not specified a segment, one would have been selected by the server.

At the server, the hard lock for O_2 is granted. The remaining objects on which to acquire soft locks are O_1, O_3, O_4. The soft lock on O_1 is denied, since T_1 holds the lock $L_{3,2,H}$. The soft lock on O_3 is not granted, since a hard lock already exists on the object. If O_3 had been soft locked by T_2, then, if possible, the existing lock would have been upgraded. In either case no additional information needs to be sent to the client about O_3. Object O_4 acquires the soft lock, and the corresponding

access information is sent to the client. Having completed the locking phase, S_1 is sent to the client.

At the client, the object O_2 may be accessed using the call OBJECTread (O_2, O_2-buffer). Suppose C_2 wanted to access O_4. C_2 makes the call LOCKquery (O_4) in T_2 and finds that it has an $L_{2,0,S}$ lock on it, which is sufficient for its current needs. Since the object is already at the client, and the lock is valid, the client need not request it from the server, thus giving the client instant access to the object.

If T_2 further wants to write to O_1, it finds that it does not have the appropriate lock. Making the lock request LOCKobjects (O_1, $L_{3,0,H}$, ANY_SEGMENT, and $L_{0,0,S}$) involves the following operations: The server finds that T_1 committed and freed the lock it had on O_1. T_2 is granted the lock and, through the timestamp mechanism, it is found to have an outdated copy of the object, so the server sends the individual updated object to C_2. If O_1 had not been updated, then the server informs the client that the lock is granted.

9. ATTAINING AN OPTIMAL PARTITION

The optimal partition of objects within a database results in the transfer of only those objects a user will access in one transaction. Although users provide a partition for objects either by *Object-Segment Mapping* (OSM) or manual specification, these may not result in an optimal partition. Therefore, heuristics provide a more flexible approach to maintaining an optimal object partition based on observations of object usage over time. The heuristics employ a *migration* facility for moving objects from one segment to another. The object migration process involves the system's monitoring object usage within segments and moving objects from one segment to another within the same DBF to aid database performance.

When objects are created, OSM partitions them by semantic criteria or user-specified segments. If an object changes logical association with the other objects in its segment, it needs to migrate or move to another segment to reduce the number of segments transferred, that is, to improve performance. The database administrator and system are responsible for this process.

We consider two types of heuristics, *transaction-oriented* and *single-object* evaluation of object usage. Transaction-oriented heuristics involve monitoring object usage within the context of a transaction. That is, transaction-oriented heuristics involve monitoring how objects are used together. Single-object heuristics involve using measurements amassed over a period of time. As an example of a time-interval heuristic, consider that natural partitions may form within a segment on the basis of an access count for individual objects, namely, some objects are used very frequently, and others not at all. If enough objects exist in each group so that creating another segment is justified, the objects in the smaller group migrate to the new segment.

Three measures are currently employed for monitoring: the *access count*, *open count*, and *access ratio*. The access count refers to the number of times an object was accessed in a given segment. The open count refers to the number of times the segment was opened. The access ratio is the quotient between the access count and the open count.

Some of these heuristics require keeping detailed statistics on both segments and objects. These are maintained in files at the server and used initially to fine-tune the database.

10. RELATED WORK

Attaining an optimal data organization to minimize retrieval costs is certainly not a new idea [21]. Schkolnick devised a clustering algorithm to perform this task [15]. Although our desired result is the same, our data structures (hierarchical data represented in a tree versus objects), work environment, and means are quite different.

Schkolnick chose to group all instances of a type into a segment. These segments were then grouped in a hierarchical tree structure. To apply his algorithm, Schkolnick obtains usage patterns of data access that are basically the most frequently used access paths in the hierarchy. Examining these patterns allows the algorithm to determine a partition of the tree segments. Once subtrees have been produced, the available disk space is divided into linear address spaces (LASs), one for each partition. The instances of all segments for a given partition are placed in their corresponding LASs in the same order as they appear in the hierarchical order. This idea is equivalent to our method of storing objects with their subobjects. Next, each LAS is divided into blocks of equal length called pages. The objective is to minimize page faults. The system paging mechanism acquires the data from secondary storage, and the data are accessed directly through these pages. A page fault is noted whenever a record is not found on a page already in the buffer pool. Schkolnick has shown that, for a given access pattern, the access time can be minimized on the basis of the storage method for a hierarchical structure. He states, "The predicted optimal storage allocation does in fact significantly reduce the average number of page faults over that obtained when the structure is stored in the conventional hierarchical order" [15, p. 43–44].

This method has several similarities to our segmentation scheme. As mentioned, the notion of storing objects with their subobjects is a common thread. In terms of accessing data, we read a set of related objects expected to be accessed during a transaction. In our method of fathoming a better partition, we analyze usage patterns amassed over time. To clarify the term *segment* in the two contexts, we allow a melange of instances of object types to reside within a segment, whereas Schkolnick's segments are a grouping of type instances which themselves are grouped into LASs. This latter grouping into LASs is similar to our use of *segment groups*, where related segments can be accessed as one larger unit. However, our segment groups maintain their individual segment identities. With Schkolnick's LASs, objects along the same access path within a subtree are stored contiguously, which shatters the original segment boundaries.

We allow more options for object grouping; hence, a more tailored set of objects may be placed in one segment. However, the option for all instances of a type is also provided. Since we devised our method for an object-oriented system, we pose virtually no restrictions on the overall data configuration and dependencies between objects. Our method has no immediate concern for the underlying paging

mechanism, which makes it freer from hardware idiosyncracies. We also provide specific segment operators that give exclusive access to database objects.

11. SUMMARY

In this paper we have given an overview of our object-oriented database system. Within that discussion, our central focus has been the implementation of a typeless object server that is used as the backend. Specifically, we have described our segmentation scheme and the mechanisms that are used for controlled sharing. In a final section we have pointed out the interactions and problems encountered in building these two facilities, and we have sketched our solution.

A prototype of this system has been implemented. We have linked the prototype of ObServer with the GARDEN [11] programming environment. GARDEN is a system for visual programming. It allows programmers to construct their programs in terms of pictures. GARDEN also contains a set of tools for easily constructing new pictorial languages. GARDEN treats everything as an object and, as such, provides an excellent testbed for our system. GARDEN views static program pieces, such as modules, statements, and variables, as objects. It also views dynamic structures, such as stack frames, as objects as well. Future versions of GARDEN will make use of the ENCORE database system to take advantage of some of the more advanced features, such as version control.

There are many important research issues that need to be investigated. We view implementation issues as among the most important. This technology will only succeed to the extent that it can be made to operate efficiently. Many of the ideas expressed in this paper were derived from experience with our prototype in the GARDEN environment. We expect this kind of refinement to continue.

The issue of being able to handle objects of widely differing sizes is very important. How can we manage huge objects, such as bit maps or large programs, in a homogeneous way with small objects, such as characters or integers? The data model presents no problem here, but the implementation problems of managing these objects on a disk are yet to be solved.

At the model level, it is tempting and useful to be able to treat everything as an object. For example, we might treat paragraphs, sentences, and characters as objects. It is not unreasonable to incur the overhead associated with an object for paragraphs and sentences, but this overhead at the character level would be completely unreasonable. We therefore need a scheme whereby characters can be conceptually stored as objects, but not as full-fledged objects at the implementation level. A scheme such as this would require that the container objects know something about the form of the objects that are contained in them, and that inbound references be handled specially.

Other research areas include topics like extending and enhancing the data model to include facilities like triggers and views and designing a more complete transaction management scheme that supports concurrency control and recovery differently for different types. The issues involved in effectively supporting the management of change still require further study. Designing a databased programming language whose model of data is precisely the model that we have described above is currently underway. We are also interested in extending our

database system to operate in a distributed database environment and to run on parallel-processor machines.

ACKNOWLEDGMENTS

The authors wish to thank David Babson, Steve Reiss, and Andrea Skarra for many invaluable discussions about the design of this system. We would also like to recognize their help, as well as the help of Mark Lellouch and Wlodek Nakonieczey, in implementing the prototype.

REFERENCES

1. BOBROW, D., AND STEFIK, M. The Loops Manual. Xerox Corp., Palo Alto, Calif., 1983.
2. CHEN, P. P. S. The entity-relationship model: Towards a unified view of data. *ACM Trans. Database Syst. 1*, 1 (Mar. 1976), 9–36.
3. CODD, E. F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (Dec. 1979), 397–434.
4. COPELAND, G., AND MAIER, D. Making smalltalk a database system. In *Proceedings of the ACM SIGMOD* (Boston, June 18–21). ACM, New York, 1984, 316–325.
5. DITTRICH, K., GOTTHARD, W., AND LOCKEMANN, P. C. DAMOKLES—A database system for software engineering environments. In *Proceedings of the IFIP 2.4 Workshop on Advanced Programming Environments* (Trondheim, Norway, June), 1986.
6. GRAY, J. The transaction concept: Virtues and limitations. In *Proceedings of the Very Large Database Conference* (Cannes, France, Sept.), 1981.
7. HAMMER, M., AND MCLEOD, D. Database description with SDM: A semantic database model. *ACM Trans. Database Syst. 6*, 3 (Sept. 1981), 351–387.
8. MAIER, D., AND STEIN, J. Indexing in an object-oriented DBMS. Tech. Rep. CS/E-86-006, Oregon Graduate Center, Univ. of Oregon, Beaverton, Oreg., (May, 1986).
9. MAIER, D., STEIN, J., OTIS, A., AND PURDY, A. Development of an object-oriented DBMS. Tech. Rep. CS/E-86-005, Oregon Graduate Center, Univ. of Oregon, Beaverton, Oreg., (Apr. 1986).
10. MOON, D., STALLMAN, R., AND WEINREB, D. The Lisp Machine Manual, chap 20. MIT AI Laboratory, Jan. 1983, 321–361.
11. MOSS, E. The theory of nested transactions. Tech. Rep., University of Massachusetts, 1986.
12. MYLOPOULOS, J., BERNSTEIN, P. A., AND WONG, H. K. T. A language facility for designing database-intensive applications. *ACM Trans. Database Syst. 5*, 2 (June 1980), 185–207.
13. REISS, S. P. An object-oriented framework for graphical programming. *SIGPLAN Not. 21*, 10 (Oct. 1986) 49–57.
14. RUDMIK, A. Choosing an environment data model. In *Proceedings of the IFIP 2.4 Workshop on Advanced Programming Environments* (Trondheim, Norway, June), 1986.
15. SCHKOLNICK, M. A. A clustering algorithm for hierarchical structures. *Trans. Database Syst. 12*, 1 (Mar. 1977) 27–44.
16. SHIPMAN, D. W. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 6*, 1 (Mar. 1981), 140–173.
17. SKARRA, A. H., AND ZDONIK, S. B. The management of changing types in an object-oriented database. In *Proceedings of the First Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oreg., Sept. 29–Oct. 2). ACM, New York, 1986, 483–495.
18. SKARRA, A. H., AND ZDONIK, S. B. Type evolution in an object-oriented database. In *Research Directions in Object-Oriented Programming.* Addison-Wesley, Reading, Mass., 1987.
19. SKARRA, A. H., ZDONIK, S. B., AND REISS, S. P. An object server for an object-oriented database system. In *International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept.). ACM, New York, 1986, pp. 196–204.
20. SMITH, J. M., FOX, S., AND LANDERS. T. *ADAPLEX: Rational and Reference Manual.* 2nd ed. Computer Corporation of America, Cambridge, Mass., 1983.
21. STAMOS, J. W. On object grouping experiments in LOOM. Xerox PARC report SC 6-82-2, Xerox Corp., Palo Alto, Calif.

22. ZDONIK, S. B. Object management system concepts. In *Proceedings of the 2nd ACM-SIGOA Conference on Office Information Systems* (Toronto, Canada, June 25–27). ACM, New York, 1984, 13–19.
23. ZDONIK, S. B. Why properties are objects or some refinements of Is-a. In *Proceedings of the National Computer Conference* (Austin, Tex.). ACM, New York, 1986.
24. ZDONIK, S. B. Version management in an object-oriented database. In *Proceedings of the International Workshop on Advanced Programming Environments* (Trondheim, Norway, June), 1986.
25. ZDONIK, S. B., AND WEGNER, P. Language and methodology for object-oriented database environments. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences* (Honolulu, Jan.) 1986, 378–387.

Received September 1986; revised December 1986; accepted December 1986