A Shared Object Hierarchy†

Lawrence A. Rowe

Computer Science Division - EECS
University of California
Berkeley, CA 94720

Abstract

This paper describes the design and proposed implementation of a shared object hierarchy. The object hierarchy is stored in a relational database and objects referenced by an application program are cached in the program's address space. The paper describes the database representation for the object hierarchy and the use of POSTGRES, a next-generation relational database management system, to implement object referencing efficiently. The shared object hierarchy system will be used to implement OBJFADS, an object-oriented programming environment for interactive multimedia database applications, that will be the programming interface to POSTGRES.

1. Introduction

Object-oriented programming has received much attention recently as a new way to develop and structure programs [GoR83, StB86]. This new programming paradigm, when coupled with a sophisticated interactive programming environment executing on a workstation with a bit-mapped display and mouse, improves programmer productivity and the quality of programs they produce.

A program written in an object-oriented language is composed of a collection of objects that contain data and procedures. These objects are organized into an *object hierarchy*. Previous implementations of object-oriented languages have required each user to have his or her own private object hierarchy. In other words, the object hierarchy is not shared. Moreover, the object hierarchy is usually restricted to main memory. The LOOM system stored object hierarchies in secondary memory [KaK83], but it did not allow object sharing. These restrictions limit the applications to which this new programming technology can be applied.

There are two approaches to building a shared object hierarchy capable of storing a large number of objects. The first approach is to build an object data manager [Afe85, CoM84, Dae85, Dee86, KhV87, MaS86, Tha86]. In this approach, the data

† This research was supported by the National Science Foundation under Grant DCR-8507256 and the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089.

manager stores objects that a program can fetch and store. The disadvantage of this approach is that a complete database management system (DBMS) must be written. A query optimizer is needed to support object queries (e.g., ‘‘fetch all *foo* objects where field *bar* is *bas*’’). Moreover, the optimizer must support the equivalent of relational joins because objects can include references to other objects. A transaction management system is needed to support shared access and to maintain data integrity should the software or hardware crash. Finally, protection and integrity systems are required to control access to objects and to maintain data consistency. These modules taken together account for a large fraction of the code in a DBMS. Proponents of this approach argue that some of this functionality can be avoided. However, we believe that eventually all of this functionality will be required for the same reasons that it is required in a conventional database management system.

The second approach, and the one we are taking, is to store the object hierarchy in a relational database. The advantage of this approach is that we do not have to write a DBMS. A beneficial side-effect is that programs written in a conventional programming language can simultaneously access the data stored in the object hierarchy. The main objection to this approach has been that the performance of existing relational DBMS’s has been inadequate. We believe this problem will be solved by using POSTGRES as the DBMS on which to implement the shared hierarchy. POSTGRES is a next-generation DBMS currently being implemented at the University of California, Berkeley [StR86]. It has a number of features, including data of type procedure, alerters, precomputed procedures and rules, that can be used to implement the shared object hierarchy efficiently.

Figure 1 shows the architecture of the proposed system. Each application process is connected to a database process that manages the shared database. The application program is presented a conventional view of the object hierarchy. As objects are referenced by the program, a run-time system retrieves them from the database. Objects retrieved from the database are stored in an object cache in the application process so that subsequent references to the object will not require another database retrieval. Object updates by the application are propagated to the database and to other processes that have cached the object.

Other research groups are also investigating this approach [AbW86, Ane86, KeS86, Mae87, Mey86, Ske86]. The main difference between our work and the work of these other groups is the object cache in the application process. They have not addressed the problem of maintaining cache consistency when more than one application process is using an object. Research groups that are addressing the object cache problem are using different implementation strategies that will have different performance characteristics [KhV87, Kra85, MaS86].

This paper describes how the OBJFADS shared object hierarchy will be implemented using POSTGRES. The remainder of this paper is organized as follows. Section 2 presents the object model. Section 3 describes the database representation for the

Figure 1. Process architecture.

shared object hierarchy. Section 4 describes the design of the object cache including strategies for improving the performance of fetching objects from the database. Section 5 discusses object updating and transactions. Section 6 describes the support for selecting and executing methods. And lastly, section 7 summarizes the paper.

2. Object Hierarchy Model

This section describes the object hierarchy model. The model is based on the Common Lisp Object System (CLOS) [BoK87] because OBJFADS is being implemented in Common Lisp [Ste84].

An *object* can be thought of as a record with named *slots*. Each slot has a data type and a default value. The data type can be a primitive type (e.g., *Integer*) or a reference to another object.[1] The type of an object is called the *class* of the object. Class information (e.g., slot definitions) is represented by another object called the *class object*.[2] A particular object is also called an *instance* and object slots are also called *instance variables*.

A class inherits data definitions (i.e., slots) from another class, called a *superclass*, unless a slot with the same name is defined in the class. Figure 2 shows a class hierarchy (i.e., type hierarchy) that defines equipment in an integrated circuit (IC) computer integrated manufacturing database. [RoW87]. Each class is represented by a labelled node (e.g., *Object*, *Equipment*, *Furnace*, etc.). The superclass of each class is indicated by the solid line with an arrowhead. By convention, the top of the hierarchy is an object named *Object*. In this example, the class *Tylan*, which represents a furnace produced by a particular vendor, inherits slots from *Object*, *Equipment*, and *Furnace*.

As mentioned above, the class is represented by an object. The type of these class objects is represented by the class named *Class*. In other words, they are instances of the class *Class*. The *InstanceOf* relationship is represented by dashed lines in the figure. For example, the class object *Equipment* is an instance of the class *Class*. Given an object, it is possible to determine the class of which it is an instance. Consequently, slot definitions and, as described below, procedures that operate on the object can be looked-up in the class object. For completeness, the type of the class named *Class* is a class named *MetaClass*.

Figure 3 shows class definitions for *Equipment*, *Furnace*, and *Tylan*. The definition of a class specifies the name of the class, the metaclass, the superclass, and the slots. The metaclass is specified explicitly because a different metaclass is used when the objects in the class are to be stored in the database. In the example, the class *Tylan* inherits all slots in *Furnace* and *Equipment* (i.e., *Location*, *Picture*, *DateAcquired*, *NumberOfTubes*, and *MaxTemperature*).

Variables can be defined that are global to all instances of a class. These variables, called *class variables*, hold data that represents information about the entire class. For

---

[1] An object reference is represented by an *object identifier* (*objid*) that uniquely identifies the object.

[2] The term *class* is used ambiguously in the literature to refer to the type of an object, the object that represents the type (i.e., the class object), and the set of objects of a specific type. We will indicate the desired meaning in the surrounding text.

Figure 2: Equipment class hierarchy.

Class *Equipment*
MetaClass *Class*
Superclass *Object*
Slots

Location	*Point*
Picture	*Bitmap*
DateAcquired	*Date*

Class *Furnace*
MetaClass *Class*
Superclass *Equipment*
Slots

NumberOfTubes	*Integer*
MaxTemperature	*DegreesCelsius*

Class *Tylan*
MetaClass *Class*
Superclass *Furnace*
Slots

Figure 3: Class definitions for equipment.

example, a class variable *NumberOfFurnaces* can be defined for the class *Furnace* to keep track of the number of furnaces. Class variables are inherited just like instance variables except that inherited class variables refer to the same memory location. For example, the slot named *NumberOfFurnaces* inherited by *Tylan* and *Bruce* refer to the same variable as the class variable in *Furnace*.

Procedures that manipulate objects, called *methods*, take arguments of a specific class (i.e., type). Methods with the same name can be defined for different classes. For example, two methods named *area* can be defined: one that computes the area of a *box* object and one that computes the area of a *circle* object. The method executed when a program makes a call on *area* is determined by the class of the argument object. For example,

    area(x)

calls the *area* method for *box* if *x* is a *box* object or the *area* method for *circle* if it is a *circle* object. The selection of the method to execute is called *method determination*.

Methods are also inherited from the superclass of a class unless the method name is redefined. Given a function call ''$f(x)$'', the method invoked is determined by the following algorithm. Follow the *InstanceOf* relationship from x to determine the class of the argument. Invoke the method named f defined for the class, if it exists. Otherwise, look for the method in the superclass of the class object. This search up the superclass hierarchy continues until the method is found or the top of the hierarchy is reached in which case an error is reported.

Figure 4 shows some method definitions for *Furnace* and *Tylan*. Furnaces in an IC fabrication facility are potentially dangerous, so they are locked when they are not in use. The methods *Lock* and *UnLock* disable and enable the equipment. These methods are defined for the class *Furnace* so that all furnaces will have this behavior. The argument to these methods is an object representing a furnace.[3] The methods *CompileRecipe* and *LoadRecipe* compile and load into the furnace code that, when executed by the furnace, will process the semiconductor wafers as specified by the recipe text. These methods are defined on the *Tylan* class because they are different for each vendor's furnace. With these definitions, the class *Tylan* has four methods because it inherits the methods from *Furnace*.

Slot and method definitions can be inherited from more than one superclass. For example, the *Tylan* class can inherit slots and methods that indicate how to communicate with the equipment through a network connection by including the *NetworkMixin* class in

---

method Lock(self: *Furnace*)
 . . .

method UnLock(self: *Furnace*)
 . . .

method CompileRecipe(self: *Tylan*, recipe: *Text*)
 . . .

method LoadRecipe(self: *Tylan*, recipe: *Code*)
 . . .

Figure 4: Example method definitions.

---

[3] The argument name *self* was chosen because it indicates which argument is the object.

the list of superclasses.[4] Figure 5 shows the definition of *NetworkMixin* and the modified definition of *Tylan*. With this definition, *Tylan* inherits the slots and methods from *NetworkMixin* and *Furnace*. A name conflict arises if two superclasses define slots or methods with the same name (e.g., *Furnace* and *NetworkMixin* might both have a slot named *Status*). A name conflict is resolved by inheriting the definition from the first class that has a definition for the name in the superclass list. Inheriting definitions from multiple classes is called *multiple inheritance*.

3. Shared Object Hierarchy Database Design

The view of the object hierarchy presented to an application program is one consistent hierarchy. However, a portion of the hierarchy is actually shared among all concurrent users of the database. This section describes how the shared portion of the hierarchy will be stored in the database.

Shared objects are created by defining a class with metaclass *DBClass*. All instances of these classes, called *shared classes*, are stored in the database. A predefined

---

Class *NetworkMixin*
MetaClass *Class*
Superclass *Object*
Instance Variables
    HostName *Text*
    Device *Text*
Methods
    SendMessage(self: *NetworkMixin*; msg: *Message*)
    ReceiveMessage (self: *NetworkMixin*) **returns** *Message*

Class *Tylan*
MetaClass *Class*
Superclass *Furnace NetworkMixin*
. . .

Figure 5: Multiple inheritance example.

---

[4] The use of the suffix *Mixin* indicates that this object defines behavior that is added to or mixed into other objects. This suffix is used by convention to make it easier to read and understand an object hierarchy.

shared class, named *DBObject*, is created at the top of the shared object hierarchy. The relationship between this class and the other predefined classes is shown in figure 6. All superclasses of a shared object class must be shared classes except *DBObject*. This restriction is required so that all definitions inherited by a shared class will be stored in the database.

The POSTGRES data model supports attribute inheritance, user-defined data types, data of type procedure, and rules [RoS87, StR86] which are used by OBJFADS to create the database representation for shared objects. System catalogs are defined that maintain information about shared classes. In addition, a relation is defined for each class that contains a tuple that represents each class instance. This relation is called the *instance relation*.

OBJFADS maintains four system catalogs to represent shared class information: *DBObject*, *DBClass*, *SUPERCLASS*, and *METHODS*. The *DBObject* relation identifies objects in the database:

Figure 6: Predefined classes.

CREATE DBObject(Instance, Class)

where

Instance	is the *objid* of the object.
Class	is the *objid* of the class object of this instance.

This catalog defines attributes that are inherited by all instance relations. No tuples are inserted into this relation (i.e., it represents an abstract class). However, all shared objects can be accessed through it by using transitive closure queries. For example, the following query retrieves the *objid* of all instances:

RETRIEVE (DBObject*.Instance)

The asterisk indicates closure over the relation *DBObject* and all other relations that inherit attributes from it.

POSTGRES maintains a unique identifier for every tuple in the database. Each relation has a predefined attribute that contains the unique identifier. While these identifiers are unique across all relations, the relation that contains the tuple cannot be determined from the identifier. Consequently, we created our own object identifier (i.e., an *objid*) that specifies the relation and tuple. A POSTGRES user-defined data type, named *objid*, that represents this object identifier will be implemented. *Objid* values are represented by an identifier for the instance relation (*relid*) and the tuple (*oid*). *Relid* is the unique identifier for the tuple in the POSTGRES catalog that stores information about database relations (i.e., the *RELATION* relation). Given an *objid*, the following query will fetch the specified tuple:

RETRIEVE (o.all)
FROM o IN *relid*
WHERE o.oid = *oid*

This query will be optimized so that fetching an object instance will be very efficient.

The *DBClass* relation contains a tuple for each shared class:

CREATE DBClass(Name, Owner) INHERITS (DBObject)

This relation has an attribute for the class name (*Name*) and the user that created the class (*Owner*). Notice that it inherits the attributes in *DBObject* (i.e., *Instance* and *Class*) because DBClass is itself a shared class.

The superclass list for a class is represented in the *SUPERCLASS* relation:

CREATE SUPERCLASS(Class, Superclass, SeqNum)

where

Class	is the name of the class object.
Superclass	is the name of the parent class object.
SeqNum	is a sequence number that specifies the inheritance order in the case that a class has more than one superclass.

The superclass relationship is stored in a separate relation because a class can inherit variables and methods from more than one parent (i.e., multiple inheritance). The sequence number is required to implement the name conflict resolution rule.

Methods are represented in the METHODS relation:

CREATE METHODS(Class, Name, Source, Binary)

where

Class	is the *objid* of the class that defines the method.
Name	is the name of the method.
Source	is the source code for the method.
Binary	is the relocatable binary code for the method.

Method code is dynamically loaded into the application program as needed. Method determination and caching are discussed below.

Object instances are represented by tuples in the instance relation that has an attribute for each instance variable. For example, if the classes *Equipment*, *Furnace*, and *Tylan* shown in figure 3 were defined with metaclass *DBClass*, the relations shown in figure 7 would be created in the database. When an OBJFADS application creates an instance of one of these classes, a tuple is automatically appended to the appropriate instance relation. Notice that to create a shared class, the superclass of *Equipment* must be changed to *DBObject*.

The POSTGRES data model uses the same inheritance conflict rules for attributes that CLOS uses so attribute inheritance can be implemented in the database system. If the rules were different, OBJFADS would have to simulate data inheritance in the database or POSTGRES would have to be changed to allow user-defined inheritance rules as in CLOS.

Thus far, we have not described how OBJFADS data types (i.e., Common Lisp data types) are mapped to POSTGRES data types. Data types will be mapped between the

```
CREATE Equipment(Location, Picture, DateAcquired)
INHERITS (DBObject)

CREATE Furnace(NumberOfTubes, MaxTemperature)
INHERITS (Equipment)

CREATE Tylan()
INHERITS (Furnace)
```

Figure 7: Shared object relations.

two environments as specified by type conversion catalogs. Most programming language interfaces to database systems do not store type mapping information in the database [Ale85, Ale78, Ate83, Mye85, RoS79, Sch77]. We are maintaining this information in catalogs so that user-defined data types in the database can be mapped to the appropriate Common Lisp data type.

The type mapping information is stored in three catalogs: *TYPEMAP*, *OFTOPG*, and *PGTOOF*. The *TYPEMAP* catalog specifies a type mapping and procedures to convert between the types:

CREATE TYPEMAP(OFType, PGType, ToPG, ToOF)

where

OFType	is an OBJFADS type.
PGType	is a POSTGRES type.
ToPG	is a procedure that converts from the OBJFADS type to the POSTGRES type.
ToOF	is a procedure that converts from the POSTGRES type to the OBJFADS type.

The table in figure 8 shows the mapping for selected Common Lisp types. Where possible, Common Lisp values are converted to equivalent POSTGRES types (e.g., *fixnum* to *int4*). In other cases, the values are converted to a print representation when they are stored in the database and recreated by evaluating the print representation when they are

Common Lisp	**POSTGRES**	**Description**
fixnum	int4	4 byte integer.
float	float	4 byte floating point number.
(simple-array string-char)	char[]	Variable length character string.
symbol	char[]	A string that represents the symbol (e.g., ‘‘’x’’ for the symbol *x*).
(local) object	char[]	A string that contains a function call that will recreate the object when executed.

Figure 8: Data type mapping examples.

fetched into the program (e.g., symbols and functions). We expect over time to build-up a set of user-defined POSTGRES types that will represent the commonly used Common Lisp types (e.g., *list*, *random-state*, etc.). However, we also expect application data structures to be designed to take advantage of the natural database representation. For example, it makes more sense to store a list as a separate relation with a common attribute (e.g., a *PO#* that joins a purchase order with the line items it contains) than as an array of *objid*'s in the database.

Class variables are more difficult to represent than class information and instances variables. The straightforward approach is to define a relation *CVARS* that contains a tuple for each class variable:

```
CREATE CVARS(Class, Variable, Value)
```

where *Class* and *Variable* uniquely determine the class variable and *Value* represents the current value of the variable. This solution requires a union type mechanism because the attribute values in different tuples may have different types. POSTGRES does not support union types because they violate the relational tenet that all attribute values must have the same type.

Two other representations for class variables are possible with POSTGRES. First, a separate relation can be defined for each class that contains a single tuple that holds the current values of all class variables. For example, the following relation could be defined for the *Furnace* class:

```
FurnaceCVARS(NumberOfFurnaces)
```

Unfortunately, this solution introduces representational overhead (the extra relation) and requires another join to fetch the slots in an object. Moreover, it does not take advantage of POSTGRES features that can be used to update the count automatically.

The second alternative uses POSTGRES rules. A rule can be used to define an attribute value that appears to the application as if it was stored [SHH87]. For example, the following command defines a rule that computes the number of furnaces:

```
REPLACE ALWAYS Furnace*(
  NumberOfFurnaces = COUNT{Furnace*.Instance})
```

A reference to *Furnace.NumberOfFurnaces* will execute the COUNT aggregate to compute the current number of furnaces. The relation variable *Furnace** in the aggregate specifies that tuples in *Furnace* and all relations that inherit data from *Furnace* (e.g., *Tylan* and *Bruce*) are to be counted. With this representation, the database maintains the correct count. Notice that the command replaces this value in *Furnace** which causes the rule to be inherited by all relations that inherit data from *Furnace*. The disadvantage of this approach is that the COUNT aggregate is executed every time the class variable is referenced.

POSTGRES provides another mechanism that can be used to cache the answer to this query so that it does not have to be recomputed each time the variable is referenced. This mechanism allows the application designer to request that a rule be evaluated early (i.e., precomputed) and cached in the appropriate relation. In other words, the furnace count will be cached in the relations *Furnace*, *Tylan*, and *Bruce* so that references to the variable will avoid recomputation. Updates to *Furnace* or subclasses of *Furnace* will cause the precomputed value to be invalidated. POSTGRES will recompute the rule off-line or when the class variable is next referenced whichever comes first.

Class variables that are not computable from the database can be represented by a rule that is assigned the current value as illustrated in the following command:

REPLACE ALWAYS Furnace(x = *current value*)

Given this definition, a reference to *Furnace.x* in a query will return the current value of the class variable. The variable is updated by redefining the rule. We plan to experiment with both the single tuple relation and rule approaches to determine which provides better performance.

This section described the object hierarchy model and a database design for storing it in a relational database. The next section describes the application process object cache and optimizations to improve the time required to fetch an object from the database.

4. Object Cache Design

The object cache must support three functions: object fetching, object updating, and method determination. This section describes the design for efficiently accessing objects. The next section describes the support for object updating and the section following that describes the support for method determination.

The major problem with implementing an object hierarchy on a relational database system is the time required to fetch an object. This problem arises because queries must be executed to fetch and update objects and because objects are decomposed and stored in several relations that must be joined to retrieve it from the database. Three strategies will be used to speed-up object fetch time: caching, precomputation, and prefetching. This section describes how these strategies will be implemented.

The application process will cache objects fetched from the database. The cache will be similar to a conventional Smalltalk run-time system [Kae81]. An object index will be maintained in main memory to allow the run-time system to determine quickly if a referenced object is in the cache. Each index entry will contain an object identifier and the main memory address of the object. All object references, even instance variables that reference other objects, will use the object identifier assigned by the database (i.e., the *instance* attribute). These indirect pointers may slow the system down but they avoid the problem of mapping addresses when objects are moved between main memory and the database.[5] The object index will be hashed to speed-up object referencing.

---

[5] Most Smalltalk implementations use a similar scheme and it does not appear to be a bottleneck.

Object caching can speed-up references to objects that have already been fetched from the database but it cannot speed-up the time required to fetch the object the first time it is referenced. The implementation strategy we will use to solve this problem is to precompute the memory representation of an object and to cache it in an OBJFADS catalog:

CREATE PRECOMPUTED(Objid, ObjRep)

where

Objid	is the object identifier.
ObjRep	is the main memory object representation.

Suppose we are given the function *RepObject* that takes an object identifier and returns the memory representation of the object. Notice that the memory representation includes class variables and data type conversions. An application process could execute *RepObject* and store the result back in the *PRECOMPUTED* relation. This approach does not work because the precomputed representation must be changed if another process updates the object either through an operation on the object or an operation on the relation that contains the object. For example, a user could run the following query to update the values of *MaxTemperature* in all *Furnace* objects:

REPLACE Furnace*(MaxTemperature = *newvalue*)

This update would cause all *Furnace* objects in *PRECOMPUTED* to be changed.[6]

A better approach is to have the DBMS process execute *RepObject* and invalidate the cached result when necessary. POSTGRES supports precomputed procedure values that can be used to implement this approach. Query language commands can be stored as the value of a relation attribute. A query that calls *RepObject* to compute the memory representation for the object can be stored in *PRECOMPUTED.Objrep*:

RETRIEVE (MemRep = RepObject($Objid))

$Objid refers to the object identifier of the tuple in which this query is stored (i.e., *PRECOMPUTED.Objid*). To retrieve the memory representation for the object with *objid* ‘‘Furnace-123,’’ the following query is executed:

RETRIEVE (object = PRECOMPUTED.ObjRep.MemRep)
WHERE PRECOMPUTED.objid = ‘‘Furnace-123’’

The nested dot notation (*PRECOMPUTED.ObjRep.MemRep*) accesses values from the result tuples of the query stored in *ObjRep* [Zan83]. The constant ‘‘Furnace-123’’ is an external representation for the *objid* (i.e., the *Furnace* object with *oid* 123). Executing this query causes *RepObject* to be called which returns the main memory representation of the object.

---

[6] *Furnace* objects cached in an application process must also be invalidated. Object updating, cache consistency, and update propagation are discussed in the next section.

This representation by itself does not alter the performance of fetching an object. The performance can be changed by instructing the DBMS to precompute the query in *ObjRep* (i.e., to cache the memory representation of the object in the *PRECOMPUTED* tuple). If this optimization is performed, fetching an object turns into a single relation, restriction query that can be efficiently implemented. POSTGRES supports precomputation of query language command values similar to the early evaluation of rules described above.[7] Database values retrieved by the commands will be marked so that if they are updated, the cached result can be invalidated. This mechanism is described in greater detail elsewhere [Sto86, Sto87].

The last implementation strategy to speed-up object referencing is prefetching. The basic idea is to fetch an object into the cache before it is referenced. The *HINTS* relation maintains a list of objects that should be prefetched when a particular object is fetched:

CREATE HINTS(FetchObject, HintObject, Application)

When an object is fetched from the database by an application (*Application*), all *HintObject*'s for the *FetchObject* will be fetched at the same time. For example, after fetching an object, the following query can be run to prefetch other objects:

RETRIEVE (obj = p.ObjRep.MemRep)
FROM p IN PRECOMPUTED, h IN HINTS
WHERE p.Objid = h.HintObject
AND h.FetchObject = *fetched-object-identifier*
AND h.Application = *application-name*

This query fetches objects one-at-a-time. We will also investigate precomputing collections of objects, so called *composite objects* [StB86]. The idea is to precompute a memory representation for a composite object (e.g., a form or procedure definition that is composed of several objects) and retrieve all objects into the cache in one request. This strategy may speed-up fetching large complex objects with many subobjects.

We believe that with these three strategies object retrieval from the database can be implemented efficiently. Our attention thus far has been focussed on speeding up object fetching from the database. We will also have to manage the limited memory space in the object cache. An LRU replacement algorithm will be used to select infrequently accessed objects to remove from the cache. We will also have to implement a mechanism to ‘‘pin down’’ objects that are not accessed frequently but which are critical to the execution of the system or are time consuming to retrieve.

This section described strategies to speed-up object fetching. The next section discusses object updating.

---

[7] The POSTGRES server checks that the command does not update the database and that any procedures called in the command do not update the database so that precomputing the command will not introduce side-effects.

5. Object Updating and Transactions

This section describes the run-time support for updating objects. Two aspects of object updating are discussed: how the database representation of an object is updated (database concurrency and transaction management) and how the update is propagated to other application processes that have cached the object.

The run-time system in the application process specifies the desired update mode for an object when it is fetched from the database into the object cache. The system supports four update modes: local-copy, direct-update, deferred-update, and object-update. Local-copy mode makes a copy of the object in the cache. Updates to the object are not propagated to the database and updates by other processes are not propagated to the local copy. This mode is provided so that changes are valid only for the current session.

Direct-update mode treats the object as though it were actually in the database. Each update to the object is propagated immediately to the database. In other words, updating an instance variable in an object causes an update query to be run on the relation that represents instances of the object. A conventional database transaction model is used for these updates. Write locks are acquired when the update query is executed and they are released when it finishes (i.e., the update is a single statement transaction). Note that read locks are not acquired when an object is fetched into the cache. Updates to the object made by other processes are propagated to the cached object when the run-time system is notified that an update has occurred. The notification mechanism is described below. Direct-update mode is provided so that the application can view ‘‘live data.’’

Deferred-update mode saves object updates until the application explicitly requests that they be propagated to the database. A conventional transaction model is used to specify the update boundaries. A begin transaction operation can be executed for a specific object. Subsequent variable accesses will set the appropriate read and write locks to ensure transaction atomicity and recoverability. The transaction is committed when an end transaction operation is executed on the object. Deferred-update mode is provided so that the application can make several updates atomic.

The last update mode supported by the system is object-update. This mode treats all accesses to the object as a single transaction. An intention-to-write lock is acquired on the object when it is first retrieved from the database. Other processes can read the object, but they cannot update it. Object updates are propagated to the database when the object is released from the cache. This mode is provided so that transactions can be expressed in terms of the object, not the database representation. However, note that this mode may reduce concurrency because the entire object is locked while it is in the object cache.

Thus far, we have only addressed the issue of propagating updates to the database. The remainder of this section will describe how updates are propagated to other processes that have cached the updated object. The basic idea is to propagate updates through the shared database. When a process retrieves an object, a database alerter [BuC79] is set on the object that will notify the process when it is updated by another process. When the alerter is trigger by another process, the process that set the alerter is notified. The value returned by the alerter to the process that set it is the updated value of the object. Note that the precomputed value of the object memory representation will be invalidated by the update so that it will have to be recomputed by the POSTGRES server. The advantage of this approach is that the process that updates an object does not have to know which processes want to be notified when a particular object is updated.

The disadvantages of this approach are that the database must be prepared to handle thousands of alerters and the time and resources required to propagate an update may be prohibitive. Thousands of alerters are required because each process will define an alerter for every object in its cache that uses direct-, deferred-, or object-update mode. An alerter is not required for local-copy mode because database updates by others are not propagated to the local copy. POSTGRES is being designed to support large databases of rules so this problem is being addressed.

The second disadvantage is the update propagation overhead. The remainder of this section describes two propagated update protocols, an alerter protocol and a distributed cache update protocol, and compares them. Figure 9 shows the process structure for the alerter approach. Each application process (AP) has a database process called its POSTGRES server (PS). The POSTMASTER process (PM) controls all POSTGRES servers. Suppose that AP_i updates an object in the database on which $M \leq N$ AP's have set an alerter. Figure 10 shows the protocol that is executed to propagate the updates to the other AP's. The cost of this propagated update is:

2M+1	process-to-process messages
1	database update
1	catalog query
1	object fetch

The object fetch is avoidable if the alerter returns the changed value. This optimization works for small objects but may not be reasonable for large objects.

The alternative approach to propagate updates is to have the user processes signal each other that an update has occurred. We call this approach the *distributed cache update* approach. The process structure is similar to that shown in figure 9, except that each AP must be able to broadcast a message to all other AP's. Figure 11 shows the distributed cache update protocol. This protocol uses a primary site update protocol. If AP_i does not have the update token signifying that it is the primary site for the object, it sends a broadcast message to all AP's requesting the token. The AP that has the token sends it to AP_i. Assuming that AP_i does not have the update token, the cost of this protocol is:

Figure 9. Process structure for the alerter approach.

1. AP_i updates the database.
2. PS_i sends a message to PM indicating which alerters were tripped.
3. PM queries the alerter catalog to determine which PS's set the alerters.
4. PM sends a message to PS_j for each alerter.
5. Each PS_j sends a message to AP_j indicating that the alerter has been tripped.
6. Each PS_j refetches the object.

Figure 10. Propagated update protocol for the alerter approach.

1. AP_i acquires the update token for the object.
2. AP_i updates the database.
3. AP_i broadcasts to all AP's that the object has been updated.
4. Each AP_j that has the object in its cache refetches it.

Figure 11. Propagated update protocol for the distributed cache approach.

2	broadcast messages
1	process-to-process message
1	database update
1	object fetch

One broadcast message and the process-to-process message are eliminated if AP_i already has the update token. The advantage of this protocol is that a multicast protocol can be used to implement the broadcast messages in a way that is more efficient than sending N process-to-process messages. Of course, the disadvantage is that AP's have to examine all update signals to determine whether the updated object is in its cache.

Assume that the database update and object fetch take the same resources in both approaches and that the alerter catalog is cached in main memory so the catalog query does not have to read the disk in the alerter approach. With these assumptions, the comparison of these two approaches comes down to the cost of 2 broadcast messages versus 2M process-to-process messages. If objects are cached in relatively few AP's (i.e., M << N) and broadcast messages are efficient, the distributed cache update appears better. On the other hand, if M is larger, so the probability of doing 2 broadcasts goes up, and broadcasts are inefficient, the alerter approach appears better. We have chosen the alerter approach because an efficient multicast protocol does not exist but the alerter mechanism will exist in POSTGRES. If this approach is too slow, we will have to tune the alerter code or implement the multicast protocol.

This section described the mechanisms for updating shared objects. The last operation that the run-time system must support is method determination which is discussed in the next section.

6. Method Determination

Method determination is the action taken to select the method to be executed when a procedure is called with an object as an argument. Conventional object-oriented systems implement a cache of recently called methods to speed-up method determination [GoR83]. The cache is typically a hash table that maps an object identifier of the receiving object and a method name to the entry address of the method to be executed. If the desired object and method name is not in the table, the standard look-up algorithm is invoked. In memory resident Smalltalk systems, this strategy has proven to be very good because high hit ratios have been achieved with modest cache sizes (e.g., 95% with 2K entries in the cache) [Kra83].

We will adapt the method cache idea to a database environment. A method index relation will be computed that indicates which method should be called for each object class and method name. The data will be stored in the *DM* relation defined as follows:

```
CREATE DM(Class, Name, DefClass)
```

where

Class	is the class of the argument object.
Name	is the name of the method called.
DefClass	is the class in which the method is defined.

Given this relation, the binary code for the method to be executed can be retrieved from the database by the following query:

```
RETRIEVE (m.Binary)
FROM m IN METHODS, d IN DM
WHERE m.Class = d.DefClass
  AND d.Class = argument-class-objid
  AND d.Name = method-name
```

The DM relation can be precomputed for all classes in the shared object hierarchy and incrementally updated as the hierarchy is modified.

Method code will be cached in the application process so that the database will not have to be queried for every procedure call. Procedures in the cache will have to be invalidated if another process modifies the method definition or the inheritance hierarchy. Database alerters will be used to signal object changes that require invalidating cache entries. We will also support a check-in/check-out protocol for objects so that production programs can isolate their object hierarchy from changes being made by application developers [Kat83].

This section described a shared index that will be used for method determination.

7. Summary

This paper described a proposed implementation of a shared object hierarchy in a POSTGRES database. Objects accessed by an application program are cached in the application process. Precomputation and prefetching are used to reduce the time to retrieve objects from the database. Several update modes were defined that can be used to control concurrency. Database alerters are used to propagate updates to copies of objects in other caches. A number of features in POSTGRES will be exploited to implement the system, including: rules, POSTQUEL data types, precomputed queries and rules, and database alerters.

References

[AbW86] R. M. Abarbanel and M. D. Williams, A Relational Representation for Knowledge Bases, Unpublished manuscript, Apr. 1986.

[Afe85] H. Afsarmanesh and et. al., ‘‘An Extensible, Object-Oriented Approach to Databases for VLSI/CAD’’, *Proc. 11th Int. Conf. on VLDB*, Aug. 1985.

[Ale85] A. Albano and et. al., ‘‘Galileo: A Strongly-Typed, Interactive Conceptual Language’’, *ACM Trans. Database Systems*, June 1985, 230-260.

[Ale78] E. Allman and et. al., ‘‘Embedding a Relational Data Sublanguage in a General Purpose Programming Language’’, Proc. of a Conf. on Data: Abstraction, Definition, and Structure, *SIGPLAN Notices*,, Mar. 1978.

[Ane86] T. Anderson and et. al., ‘‘PROTEUS: Objectifying the DBMS User Interface’’, *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

[Ate83] M. P. Atkinson and et. al., ‘‘An Approach to Persistent Programming’’, *Computer Journal 26*, 4 (1983), 360-365.

[BoK87] D. Bobrow and G. Kiczales, ‘‘Common Lisp Object System Specification’’, Draft X3 Document 87-001, Am. Nat. Stand. Inst., February 1987.

[BuC79] O. P. Buneman and E. K. Clemons, ‘‘Efficiently Monitoring Relational Databases’’, *ACM Trans. Database Systems*, Sep. 1979, 368-382.

[CoM84] G. Copeland and D. Maier, ‘‘Making Smalltalk a Database System’’, *Proc. 1984 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, June 1984.

[Dae85] U. Dayal and et.al., ‘‘A Knowledge-Oriented Database Management System’’, *Proc. Islamorada Conference on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1985.

[Dee86] N. P. Derrett and et.al., ‘‘An Object-Oriented Approach to Data Management’’, *Proc. 1986 IEEE Spring Compcon*, 1986.

[GoR83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, MA, May 1983.

[Kae81] T. Kaehler, ‘‘Virtual Memory for an Object-Oriented Language’’, *Byte 6*, 8 (Aug. 1981).

[KaK83] T. Kaehler and G. Krasner, ‘‘LOOM – Large Object-Oriented Memory for Smalltalk-80 Systems’’, in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), Addison Wesley, Reading, MA, May 1983.

[Kat83] R. Katz, ‘‘Managing the Chip Design Database’’, *Computer Magazine 16*, 12 (Dec. 1983).

[KeS86] J. Kempf and A. Snyder, ‘‘Persistent Objects on a Database’’, Report STL-86-12, Sftw. Tech. Lab., HP Labs, Sep. 1986.

[KhV87] S. Khoshanfian and P. Valduriez, ‘‘Sharing, Persistence, and Object Orientation: A Database Perspective’’, DB-106-87, MCC, Apr. 1987.

[Kra85] G. L. Krablin, ‘‘Building Flexible Multilevel Transactions in a Distributed Persistent Environment’’, Persistence and Data Types, Papers for the Appin Workshop, U. of Glasgow, Aug. 1985.

[Kra83] G. Krasner, ed., *Smalltalk-80: Bits of History, Words of Advice*, Addison Wesley, Reading, MA, May 1983.

[MaS86] D. Maier and J. Stein, ‘‘Development of an Object-Oriented DBMS’’, *Proc. 1986 ACM OOPSLA Conf.*, Portland, OR, Sep. 1986.

[Mae87] F. Maryanski and et.al., ‘‘The Data Model Compiler: a Tool for Generating Object-Oriented Database Systems’’, Unpublished manuscript, Elect. Eng. Comp. Sci. Dept., Univ. of Connecticut, 1987.

[Mey86] N. Meyrowitz, ‘‘Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework’’, *Proc. 1986 ACM OOPSLA Conf.*, Portland, OR, Sep. 1986, 186-201.

[Mye85] J. Mylopoulos and et. al., ‘‘A Language Facility for Designing Interactive Database-Intensive Systems’’, *ACM Trans. Database Systems 10*, 4 (Dec. 1985).

[RoS79] L. A. Rowe and K. A. Shoens, ‘‘Data Abstraction, Views, and Updates in Rigel’’, *Proc. 1979 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, Boston, MA, May 1979.

[RoS87] L. A. Rowe and M. R. Stonebraker, ‘‘The POSTGRES Data Model’’, to appear in *Proc. 13th VLDB Conf.*, Brighton, England, Sep. 1987.

[RoW87] L. A. Rowe and C. B. Williams, ‘‘An Object-Oriented Database Design for Integrated Circuit Fabrication’’, submitted for publication, Apr. 1987.

[Sch77] J. Schmidt, ‘‘Some High Level Language Constructs for Data of Type Relation’’, *ACM Trans. Database Systems 2,* 3 (Sep. 1977), 247-261.

[Ske86] A. H. Skarra and et. al., ‘‘An Object Server for an Object-Oriented Database System’’, *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

[Ste84] G. L. Steele, *Common Lisp - The Language*, Digital Press, 1984.

[StB86] M. Stefik and D. G. Bobrow, ‘‘Object-Oriented Programming: Themes and Variations’’, *The AI Magazine 6*, 4 (Winter 1986), 40-62.

[StR86] M. R. Stonebraker and L. A. Rowe, ‘‘The Design of POSTGRES’’, *Proc. 1986 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, June 1986.

[Sto86] M. R. Stonebraker, ‘‘Object Management in POSTGRES Using Procedures’’, *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

[Sto87] M. R. Stonebraker, ‘‘Extending a Relational Data Base System with Procedures’’, to appear *ACM TOD*, 1987.

[SHH87] M. R. Stonebraker, E. Hanson and C. H. Hong, ‘‘The Design of the POSTGRES Rules System’’, *IEEE Conference on Data Engineering*, Los Angeles, CA, Feb. 1987.

[Tha86] S. M. Thatte, ‘‘Persistent memory: A Storage Architecture for Object-Oriented Database Systems’’, *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

[Zan83] C. Zaniola, ‘‘The Database Language GEM’’, *Proc. 1983 ACM-SIGMOD Conference on Management of Data*, San Jose, CA., May 1983.