THE OBJECTSTORE DATABASE SYSTEM

Charles Lamb
Gordon Landis
Jack Orenstein
Dan Weinreb

ObjectStore is an object-oriented database management system (OODBMS) that provides a tightly integrated language interface to the traditional DBMS features of persistent storage, transaction management (concurrency control and recovery), distributed data access, and associative queries. ObjectStore was designed to provide a unified programmatic interface to both persistently allocated data (i.e., data that lives beyond the execution of an application program) and transiently allocated data (i.e., data that does not survive beyond an application's execution), with object-access speed for persistent data usually equal to that of an in-memory dereference of a pointer to transient data.

These goals were derived from the requirements of ObjectStore's target applications, which are typically data-intensive programs that perform complex manipulations on large databases of objects with intricate structure, e.g., CAD, CAE, CAP, CASE, and geographic information systems (GIS). This structural complexity is generally realized by inter-object references, e.g., pointers from one object to another. Objects are located, possibly with the intent to update them, by traversing these references and by associative queries.

We selected C++ as the primary language through which ObjectStore is accessed because it is becoming a very popular language among the developers of ObjectStore's target applications. ObjectStore can also be used from C programs—providing access from C is easy because the data model of C is a subset of that of C++. Use of ObjectStore from other programming languages is discussed later.

The key to ObjectStore's integration with C++ is that persistence is not part of the type of an object. Objects of any C++ data type whatsoever can be allocated transiently (on the ordinary heap) or persistently (in a database), from built-in types such as integers and character strings, to arbitrary user-defined structures (which may contain pointers, use C++ virtual functions and multiple inheritance, etc). In particular, there is no need to inherit from a special "persistent object" base class. Different objects of the same type may be persistent or transient within the same program.

There are several motivations for our goal of making ObjectStore closely integrated with the programming language. These include:

Ease of learning: It was intentionally designed so that a C++ user would only have to learn a little bit more in order to try out ObjectStore and start to use it effectively. After that, a user can learn more, and take advantage of more of the capabilities the database offers. In particular, there is no need to learn a new type system or a new way to define objects. The declarative and procedural parts of the language are used for both kinds of objects. By providing a gradual learning path and making it easy for users to get started, we hope to make ObjectStore accessible to a wider range of developers, and help ease the transition into the use of object-oriented database technology.

No translation code: We wanted to save the programmer from having to write code that translates between the disk-resident representation of data, and the representation used during execution. For example, to store a C++ object into a relational database, the programmer must construct a mapping between the two, and write code that picks fields out of tuples and copies them into data members of objects. (This is part of the problem that has been called the "impedance mismatch" between a programming language and a database access language [2, 13].) With ObjectStore, no translating and no copying is needed. Persistent data is just like ordinary heap-allocated (transient) data: once a pointer is obtained to it, the user can just use it in the ordinary way. ObjectStore automatically takes care of locking, and keeps track of what has been modified.

Expressive power: We wanted the interface to persistently allocated objects to support all of the power of the host programming language. This contrasts with the traditional data manipulation capabilities of languages such as SQL, which are much less powerful than a general-purpose programming language.

Reusability: We wanted to promote reusability of code, by allowing the same code to operate on either persistent or transient data, and to allow libraries that were developed for manipulating transient data to work on persistent data without change. For example, if a programmer has a library routine that takes an array of floating-point numbers and computes the fast Fourier transform, he or she can pass it an ObjectStore persistent array, and it will work. Usually, if a library does not need to do any persistent allocation of its own, the library can be applied to persistent data without even being recompiled.

Conversion: Many programmers who are interested in using object-oriented DBMSs would like to add persistence to existing applications that deal with transient objects, rather than build new applications from scratch. We wanted to make it as easy as possible to convert an existing application to use persistent objects throughout. In particular, this means that basic data operations such as dereferencing pointers and getting and setting data members should be syntactically the same for persistent and transient objects, and that variables should not have to have their type declarations changed when persistent objects are used.

Type checking: We wanted the compile-time type-checking of C++ to apply to persistent data as well as transient data, with the entire application using a single type system. The compiler's type checking applies to objects in the database. For example, a variable referring to an object of class employee would have type 'employee *'. Such a variable could refer to a persistent employee or a transient employee, at different times during program execution. A function that takes a reference to an employee as an argument can therefore operate on a persistent or a transient employee.

The second goal of ObjectStore is to provide a very high performance for the kinds of applications to which ObjectStore is targetted. From the point of view of performance, the target applications are very different from traditional database applications such as payroll programs and on-line transaction processing systems, in several ways, as we found from interviewing developers of such applications.

Temporal locality: When many users access a shared database, very often the next user of a data item will be the same as the previous user. In other words, while concurrent access must be allowed and must work correctly, many data items will be used 'mostly' by one user over a short span of time.

Spatial locality: Often an application will use only a portion of a database, and that portion will be (or can be arranged to be) in a small section of the database that is contiguous, or mostly so.

Fine interleaving: Applications often interleave small database operations (i.e., go from one object to a reference object) with small amounts of computation. That is, there are many very small database operations rather than relatively few large ones. If every database operation required a significant per-operation overhead cost (such as the cost of sending a network message), overhead costs would become prohibitive.

Developers told us that it is imperative that ordinary data manipulation be as fast as possible. For example, an ECAD circuit simulation is CPU-intensive, traversing a network of objects representing a circuit, carrying out computations on the way. These simulations are quite expensive. Any approach to data management that penalizes the running time of such an application is impractical. This means that one critical operation must be as fast as possible: the operation of obtaining data from an object, given a pointer or reference to the object. This operation might be called 'fetching an object'; more precisely, it is dereferencing a pointer. ObjectStore is designed to make the speed of dereferencing of pointers to persistent objects be as close as possible to that of transient objects, namely the speed of a single load instruction.

ObjectStore also has some of the same performance goals as ordinary relational DBMSs, and it generally accomplishes these using familiar techniques such as indexes, query optimization, log-based recovery, and so on. The implementation section explains how we approached all of these performance goals, focusing on the aspects of ObjectStore that differ from conventional techniques.

Another goal of ObjectStore is to provide several features that are missing from C++ and from most DBMSs: a collection facility (sets, lists, and so on), a way to express bidirectional relationships, and support for groupware based on versioned data.

Application Interface

In addition to the data definition and manipulation facilities provided by the host languages, C and C++, ObjectStore provides support for accessing persistent data inside transactions, a library of collection types, bidirectional relationships, an optimizing query facility, and a version facility to support collaborative work. Tools supporting database schema design, database browsing, database administration, and application compilation and debugging, are also provided.

There are three programming interfaces supported, a C library interface, a C++ library interface, and an extended C++ which provides a tighter language integration to the query and relationship facilities. This interface is accessible only through ObjectStore's C++ compiler, while the two library interfaces are accessible through other third-party C or C++ compilers, thus providing maximum portability. All of the features and performance benefits of the ObjectStore architecture are realized in all of the interfaces.

Accessing Persistent Data

A simple C++ program which uses the extended C++ interface to the system is presented in Figure 1. This program opens an existing database, creates a new persistent object of class employee, adds the new employee to an existing department, and sets the salary of the employee to 1,000. The keyword **persistent** specifies a storage class, saying that this variable resides in the specified database. Persistent variables associate names with persistent objects, providing the starting point from which navigations or queries begin. The **db** argument to the **new** operator specifies that the employee object being created should be allocated in database **db**.

It should be noted that the manipulation of data looks just like an ordinary C++ program, even though the objects are persistent. They also compile into the same machine instructions: the update of the salary field just uses a simple store instruction. ObjectStore automatically sets read and write locks, and automatically keeps track of what has been modified, helping to protect the integrity of the database against the possibility of programmer error. Access to persistent data is guaranteed to be transaction-consistent (i.e., all-or-none update semantics), and recoverable in the event of system failure.

It should be noted that in Figure 1 the variable **engineering_department** is not explicitly initialized. This is because it is a persistent variable, which refers to an object stored in the "**/company/records**" database. The object is looked up by name, 'engineering_department', in the database, and the program variable is initialized to refer to the named object in the database. (It would have been an error if there had been no such object in the database.) The persistent keyword in the ObjectStore extended C++ inferface simply provides a shorthand for looking up an object in the database by name, and binding a

```
#include <objectstore/objectstore.H>
#include <records.H>

main ()
{

  // Declare a database, and an "entrypoint" into the
  // database of type "pointer to department."

  database *db;
  persistent<db> department* engineering_department;

  // Open the database.
  db = database::open ("/company/records");

  // Start a transaction so that the database
  // can be accessed.
  transaction::begin ();

  // The next three statements create and manipulate a
  // persistent object representing a person named Fred.
  employee *emp = new (db) employee ("Fred");
  engineering_department->add_employee (emp);
  emp->salary = 1000;

  // Commit all changes to the database.
  transaction::commit ();
}
```

[illegible]
Manipulating persistent data

```
/* file records.H */

class employee
{
public:
  char* name;
  int salary;
};
class department
{
public:
  os_Set<employee*> employees;

  void add_employee (employee *e)
    { employees->insert (e); }

  int works_here (employee *e)
    { return employees->contains (e); }
};
```

FIGURE 2.
Using collections

local program variable to the persistent database object.

Collections

ObjectStore provides a collection facility in the form of an object class library. Collections are abstract structures which resemble arrays in traditional programming languages, or tables in relational DBMSs. Unlike arrays or tables, however, ObjectStore collections provide a variety of behaviors, including ordered collections (lists), and collections with or without duplicates (bags or sets).

Performance tuning often involves replacing simple data structures, such as lists, with more efficient but more complex structures such as b-trees or hash tables. This aspect of application development is also handled by the collection library. Users may optionally describe intended usage by estimating frequencies of various operations, (e.g., iteration, insertion, and removal), and the collection library will transparently select an appropriate representation. Furthermore, a *policy* can be associated with the collection, dictating how the representation should change in response to changes in the collection's cardinality. These performance-tuning facilities reduce the developer's involvement from coding data structures to describing access patterns.

Figure 2 shows the user-written include file **records.H**, used in this example. Note that the class department declares a data member of type **os_Set(employee*)** .**os_Set** is a (parameterized) collection class, found in the ObjectStore collection class library. If d is a department, then **d—>add_employee(e)** simply adds **e** into d's set of employees. **d—>works_here(e)** returns *true* if **e** is contained in d's set of employees, *false* otherwise.

ObjectStore includes a looping construct to iterate over sets. For example, the code in Figure 3 gives a 10% raise to each employee in department d. In the loop, **e** is bound to each element of **d—>employees** in turn.

```
department* d;
 . . .
foreach (employee* e, d—>employees)
  e—>salary *= 1.1;
```

Iteration over a collection

```
/* file records.H */

class employee
{
public:
  string name;
  int salary;
  department* dept
      inverse_member department::employees;
};

class department
{
public:
  os_Set(employee*) employees
      inverse_member employee::dept;

  void add_employee (employee *e)
    { employees—>insert (e); }

  void works_here (employee *e)
    { employees—>contains (e) ; }
};
```

Using relationships

The Relationship Facility

Complex objects such as parts hierarchies, designs, documents, and multimedia information can be modeled using ObjectStore's relationship facility. Relationships can be thought of as a pair of inverse pointers, so that if one object points to another, the second object has an inverse pointer back to the first. Relationships maintain the integrity of these pointers. For example, if one participant in a relationship is deleted, then the pointer to that object, from the other participant, is set to null. One-to-one, one-to-many, and many-to-many relationships are supported.

To continue the example in Figure 3, we could create a relationship between employees and departments, as in Figure 4. The **dept** data member of **employee** and the **employees** data member of **department** are declared to be inverses of one another. Because one data member is a single pointer and the other is a set, the relationship is one-to-many. Whenever an employee is inserted into a department's set of employees, the employee is automatically updated to refer to the department (and viceversa). Similarly, when an employee is deleted from a department's set of employees, the pointer from the employee to the department is set to null, guaranteeing referential integrity.

Syntactically, relationships are accessed just like data members in C++, but updating the value of a relationship causes the inverse relationship to be updated as well, so that the two sides are always consistent with one another. This means that after `d->add_employee(e)` in the code example given in Figure 1, e's `dept` would be `engineering_department`, even though this field was not explicitly set by the application. This update of e would occur as a result of inserting e into `d->employees`, because of the `inverse_member` declarations. Similarly, if `e->dept` is set to another department, `d2`, then e is removed from `d->employees`, and inserted to `d2->employees`. In general, maintenance actions can involve simply unsetting the inverse, or actually deleting the object on the inverse, at the schema-definer's discretion. The latter behavior is useful for deleting hierarchies of objects, so that, for example, deleting an assembly would cause all of its subassemblies to be deleted, along with their subassemblies, recursively.

Associative Queries

In relational DBMSs, queries are expressed in a special language, usually SQL. SQL has its own variables and expressions, which differ in syntax and semantics from the variables and expressions in the host language. Bindings between variables in the two languages must be established explicitly. ObjectStore queries are more closely integrated with the host language. A query is simply an expression that operates on one or more collections and produces a collection or a reference to an object.

Selection predicates, which appear within query expressions, are also expressions, either C++ expressions or queries. Continuing the previous example, suppose that `all_employees` is a set of employee objects:

```
os_Set(employee*) all_employees;
```

The following statement uses a query against `all_employees` to find employees earning over $100,000, and assign the result to `overpaid_employees`:

```
os_Set(employee*)&
overpaid_employees =
all_employees
[: salary >= 100,000 :];
```

[: :] is ObjectStore syntax for queries. The contained expression is a selection predicate, that is (conceptually) applied to each element of `all_employees` in turn. (In fact, the query will be optimized if an index on salary is present. This is discussed later.)

Any collection, even one resulting from an expression, can be queried. For example, this query finds overpaid employees of department d:

```
d->employees
[: salary >= 100000 :]
```

Query expressions can also be nested, to form more complex queries. The following query locates employees who work in the same department as Fred:

```
all_employees
[: dept->employees
[: name == 'Fred' :] :];
```

Each member of `all_employees` has a department, `dept`, which has an embedded set of employees. The nested query is true for departments having at least one employee whose name is Fred.

All of these examples make use of the language extensions available only through the ObjectStore C++ compiler; the [: :] syntax, for example, is a language extension. The same queries can be expressed via the library interface. The previous query would be restated in the C++ library interface as:

```
os_Set(employee*)>
& work_with_fred =
all_employees->query(
'employee*',
"dept->employees
[: name == \'Fred'\ :]");
```

The first argument to query, `employee*`, indicates the type of the collection elements. The second argument is simply the string representing the query expression. It is also possible to use the library interface to store precompiled and optimized queries in the database for later execution.

In its current form, the ObjectStore query language can express 'semijoins' but not full joins; i.e., the result of a query is a subset of the collection being queried.

Versions

ObjectStore provides facilities for multiple users to share data in a cooperative fashion (sometimes referred to as groupware). With these facilities, a user can check out a version of an object or group of objects, make changes (perhaps entailing a long series of individual update transactions), and then check changes back in to the main development project so that they are visible to other members of the cooperating team. In the interim, other users can continue to use the previous versions, and therefore are not impeded by concurrency conflicts on their shared data, regardless of the duration of the editing sessions involved. These extended editing sessions on private, checked-out versions are often referred to as long transactions. The design was influenced by [3, 6, 9, 10].

If other users want to make concurrent parallel changes, they can check out alternative versions of the same object or groups of objects, and work on their versions in private. Again, the result is that there are no concurrency conflicts, even though the users are operating on (different versions of) the same objects. Alternative versions can later be merged back together to reconcile differences resulting from this parallel development. This merging operation is a difficult problem and is left to the user to implement on an application-specific basis [8]. In support of this, ObjectStore allows simultaneous access to both versions of an object during the merge.

Users can control exactly which versions to use, for each object or group of objects of interest, by setting up private workspaces that specify the desired version. This might be the most recent version, or a particular previous version (such as the previous release), or even a version on an alternative branch. Users can also use workspaces to selectively share their work in progress. Workspaces can inherit from other workspaces, so that one designer could specify that his or her workspace should by default inherit "whatever is in the team's shared workspace"; he or she could then add individual new versions as changes are made, overriding this default.

For example, a team of designers working on a CPU design might set up a workspace in which all of their new versions are created. Only when their CPU design is completed would the finished version(s) be checked in to the corporate workspace, making them available to, say, the manufacturing group. Within the design team's workspace, there might be multiple subworkspaces, which are used by subgroups of the design team or individual team members. Just as the entire group makes its work available to manufacturing by checking in a completed version to the corporate workspace, individual designers or teams of designers can make their work-in-progress available to one another by checking their intermediate versions in to their shared workspaces. This is illustrated in Figure 5.

Just as the persistence of an object is independent of type, the versioning of an object is independent of type. This means that instances of any type may be versioned, and that versioned and nonversioned instances can be operated on by the same user code. This makes it easy to take an existing piece of code, which has no notion of versioning—for example, a circuit-design simulator—and use it on versioned data. The simulator does not have to be rewritten, because operating on a particular version of a circuit design is identical to operating on a nonversioned design.

Programs using versioned data need not distinguish among versioned, persistent, and transient data in accordance with ObjectStore's design principles.

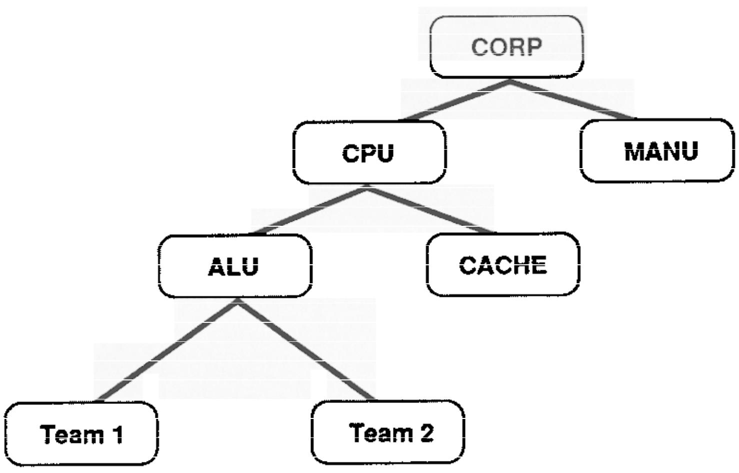
Architecture and Implementation

Storage System and Memory-Mapped Architecture

One fundamental operation of a database programming language is dereferencing: finding and using a target object that is referred to by a source object. ObjectStore's interface goals state that this must work just as in ordinary C++, to provide transparent integration with the language and to make dereferencing as fast as possible. This means that ordinary pointers from the host language must be able to serve as references from one persistent object to another.

ObjectStore's performance goals demand that once the target object has been retrieved from the database, subsequent references should be just as fast as dereferencing an ordinary pointer in the language. This means that dereferencing a pointer to a persistent target must compile exactly the same as dereferencing a pointer to a transient target, (i.e., as a single 'load' instruction), without any extra instructions to check whether the target object has been retrieved from the database yet. This creates a dilemma, since it is possible that the target object really has not yet been retrieved from the database.

Fortunately, these design goals are analogous to those of virtual memory systems, which support uniform memory references to data, whether that data is located in primary or secondary memory. ObjectStore takes advantage of the CPU's virtual memory hardware, and the operating system's interfaces that allow ordinary software to utilize that hardware. The virtual



Nesting of workspaces

memory system allows ObjectStore to set the protection for any page of virtual memory to no access, read only, or read/write. When an ObjectStore application dereferences a pointer whose target has not been retrieved into the client (i.e., a page set to no access), the hardware detects an access violation, and the operating system reflects this back to ObjectStore, as a memory fault. ObjectStore retrieves the page from the server and places it in the client's cache. It then calls the operating system to set the protection of the page to allow accesses to succeed (i.e., read only). Finally, it returns from the memory fault, which causes the dereference to restart. This time, it succeeds. Subsequent reads to the same target object, or to other addresses on the same page, will run in a single instruction, without causing a fault. Writes to the target page will result in faults that cause the page access mode and lock to be upgraded to read-write. All virtual memory mapping and address space manipulation in the application is handled by the operating system under the direction of ObjectStore, using normal system calls.

The ObjectStore server provides the long-term repository for persistent data. Databases can be stored either of two ways: within files provided by the operating system's file system, or within partitions of disks, using ObjectStore's own file system. The latter provides higher performance, by keeping databases as contiguous as possible even as they gradually grow, and by avoiding various operating system overheads. The server and the client communicate via local area network when they are running on different hosts, and by faster facilities such as shared memory, and local sockets when they are running on the same host.

The server stores and retrieves pages of data in response to requests from clients. The server has no knowledge of the contents of a page. It simply passes pages to and from the client, and stores them on disk. The server is also responsible for concurrency control and recovery, using techniques similar to those used in conventional DBMSs. It provides two-phase locking with a read/write lock for each page. Recovery is based on a log, using the write-ahead log protocol. Transactions involving more than one server are coordinated using the two-phase commit protocol. The server also provides backup to long-term storage media such as tapes, allowing full dumps as well as continuous archive logging.

Since the server has no knowledge of the contents of the page, much of the query and DBMS processing is done on the client side of the network. This contrasts with traditional relational DBMS systems in which the server is largely responsible for handling all query processing, optimization, and formatting. Although such offloading of work from the server is not ideal for all applications, this architecture does not preclude having the server handle more of the work.

ObjectStore maintains a client cache, a pool of database pages that have recently been used, in the virtual memory of the client host. When the application signals a memory fault, ObjectStore determines whether the page being accessed is in the client cache. If not, it asks the ObjectStore server to transmit the page to the client, and puts the page into the client cache. Then, the page of the client cache is mapped into virtual address space, so that the application can access it. Finally, the faulting instruction is restarted, and the application continues.

Many applications tend to reference large numbers of small objects, but networks are, in general, more efficient for bulk data. To compensate for this, whole pages of data are brought from the server to the client and placed in the cache and mapped into virtual memory. Objects are stored on the server in the same format in which they are seen by the language in virtual memory. This avoids potential per-object overhead such as calling a dynamic memory allocator, creating entries in object tables, or reformatting the nonpointer elements of the object.

When a transaction finishes, all pages are removed from the address space and modified pages are written back to the server (the client waits for an acknowledgment from the server that the pages have been safely written to disk). However, the pages remain in the client cache, so that if the next transaction uses those pages, it will not have to communicate with the server to retrieve them; they will already be present in the cache. This improves performance when several successive transactions use many of the same pages. Typical ObjectStore applications interleave computation very tightly with database access, doing some computation, then dereferencing a pointer and reading or changing a few values, then doing some more computation, etc. If it were necessary to communicate with a remote server for each of these simple database operations, the cost of the network and scheduler overhead would be enormous. By making the data directly available to the application and allowing ordinary instructions to manipulate the data, such applications perform faster.

Since a page can reside in the client cache without being locked, some other client might modify the page, invalidating the cached copy. The mechanism for making sure that transactions always see valid copies of pages is called 'cache coherence'. A copy of a page in a client cache is marked either as *shared* or *exclusive* mode. The server keeps track of which pages are in the caches of which clients, and with which modes. When a client requests a page from the server and the server notices that the page is in the cache of some other client (the *holding* client), the server will check to see if the modes conflict. If they

Applications can improve performance by exercising control over the placement of objects within a database.

do, the server sends a message to the holding client, asking it to remove the page from its cache. This is called a callback message, since it goes in the opposite direction from the usual request: the server is making a request of the client.

When the holding client receives the callback, it checks to see if the page is locked, and if not, agrees to immediately relinquish the page, and removes the copy of the page from its cache. If the page is locked, the client replies negatively to the server, and the server forces the requesting client to wait until the holder is finished with the transaction. When the holding client commits or aborts, it then removes the copy of the page from its cache, and the server can allow the original client to proceed. The use of callback messages was inspired by the Andrew File System [11]. Related cache coherency algorithms are discussed in [4].

In an ideal computer architecture with unlimited virtual address space, every object in every database could have a unique address, and virtual addresses could serve as unchanging object identifiers. Modern computers have virtual address spaces that are very large, but not unlimited. Single databases can exceed the size of the virtual address space. Also, two independent databases might each use the same address for their own objects. This is the fundamental problem that must be solved by any virtual memory-mapping approach to a DBMS.

ObjectStore solves this problem by dynamically assigning portions of address space to correspond to portions of the databases used by the application. It maintains a virtual address map that shows which database and which object within the database is represented by any address. As the application references more databases and more objects, additional address space is assigned, and the new objects are mapped into these new addresses. At the end of each transaction the virtual address map is reset, and when the next transaction starts, new assignments are made.

This solution does not place any limits on the size of a database. Naturally, each transaction is limited to accessing no more data than can fit into the virtual address space. In practice, this limit is rarely reached, since modern computers have very large virtual address spaces, and transactions are generally short enough that they do not access nearly as much data as can fit. An operation large enough to approach this limit would be divided into several transactions, and checked out into a workspace to provide isolation from other users.

When a page is mapped into virtual memory, the correspondence of objects and virtual addresses may have changed. The value of each pointer stored in the page must be updated, to follow the new virtual address of the object. This is called *relocation* of the pointers. When possible, ObjectStore arranges to assign the address space so that pointers as stored on the server happen to be the same as the values they ought to have in virtual memory. In this case, relocation is not needed, which improves performance. But sometimes relocation cannot be avoided. For example, when the database size exceeds the size of the available address space, relocation is required.

ObjectStore maintains an auxiliary data structure called the tag table that keeps track of the location and type of every object in the database. When a page is mapped into virtual address space and pointer relocation is needed, ObjectStore consults the tag table to find out what objects reside on the page, and then uses the database schema to learn which locations within each object contain pointers. It then adjusts the value of the pointer to account for the new assignments of data to the virtual address space. To minimize space overhead while keeping access fast, the tag table is heavily compressed, and is indexed. Each tag table entry contains a 16-bit type code, which indexes into a type table stored in the database's schema. The type table entry indicates which words of the type contain pointers. Tag table pages are brought into the client cache as needed, and managed in the cache like ordinary database pages.

Applications can improve performance by exercising control over the placement of objects within a database. By clustering together objects that are frequently referenced together, locality is increased, the client cache is used more efficiently, and fewer pages need to be transferred in order to access the objects. ObjectStore divides a database into areas called segments, and whenever an application creates a new persistent object, it can specify the segment in which that object should be created. Applications can create as many segments as are needed. Segments

Since locking granularity is on a per-page basis, the advantages of clustering are realized in decreased locking overhead.

may be transferred from server to client either en masse, or one page at a time, depending on the setting of an application-controlled per-segment flag.

Objects can cross page boundaries, and can be much larger than a page. Image data, for example, can be stored in very large arrays that span many pages. If an application needs to access only a small portion of such a huge object, it can use page-granularity transfer, to transfer only the pages of the object that are actually used. Conversely, many small objects can reside on a single page. Since locking granularity is on a per-page basis, the advantages of clustering are also realized in decreased locking overhead.

ObjectStore depends on the operating system to control the mapping and protection of pages, and to allow access violations to be handled by software. The most standard versions of Unix, such as SVR4, OSF/1, Berkeley bsd 4.3, and SunOS all provide these facilities. For other versions of Unix, ObjectStore includes a device driver that must be linked with the kernel when ObjectStore is installed. ObjectStore never modifies the Unix kernel itself. Future versions of these operating systems are expected to provide these memory manipulation facilities. ObjectStore currently runs on Sun 3 and SPARC, under SunOS, IBM RS/6000, under AIX, DEC DS3100, under Ultrix, HP series 300, 400, and 700, under HP/UX. By the end of 1991, ObjectStore should also be running on DEC under VMS, and SGI. Most other popular kernel-based operating systems, including VMS and OS/2, provide the facilities that ObjectStore needs. ObjectStore is also available on Microsoft Windows 3.0. Windows does not have a protected kernel like Unix, so ObjectStore controls virtual memory directly.

Collections

In designing the collection facility, an important design goal was that performance must be comparable to that of hand-coded data structures, across a wide range of applications and cardinality. Often, objects have embedded collections. For example, a Person object might contain a set of children. In these cases, cardinalities are usually small, often 0 or 1, and only occasionally above 5–10. Collections are also used to store all objects of some type, e.g., all employees, and such collections can be arbitrarily large. Furthermore, access patterns differ greatly among applications, and even over time within a single application. Clearly, a single representation type will be inadequate when performance is a concern, so multiple representations of collections must be supported. However, it is not desirable for the user to have to deal with these representations directly. The user should be able to work through an interface that reflects behavior, not representation.

The ObjectStore collection facilities are arranged into two class hierarchies: one for collections, and another for cursors. The base of the collection hierarchy is os_collection, which is actually the base for two hierarchies. One of these contains os_set, os_bag, and os_list. These provide familiar combinations of behavior. Other combinations can be obtained by specifying combinations of behavior for an os_collection, (e.g., a list without duplicates, or a set that raises an exception upon insertion of a duplicate, instead of silently ignoring it).

The other hierarchy under os_collection provides for various representations of collections. Each representation supports the entire os_collection interface, but with different performance characteristics. These classes are available for direct use, but it should never be necessary to work with representations directly. Instead, a representation is normally selected automatically, based on user-supplied estimates of access patterns (i.e., how frequently various operations will be carried out).

Operations on collections appear as *methods*, (or member functions, to use the C++ terminology). As is typical of object-oriented languages, there is a run-time function dispatch, to locate the appropriate implementation of each function, based on the collection's behavior and representation. When a collection modifies itself to employ a different representation, it actually modifies its own (representation) type description, so function dispatches will continue to work correctly.

Queries

Syntactically, queries are treated as ordinary expressions in an extended C++. However, query expressions are handled quite differently from other kinds of expressions. The obvious implementation strategy—iterate and check the predicate—would provide very poor performance for large collections. In relational DBMSs, indexes can be supplied to permit more efficient implementations. A query optimizer examines a variety of strategies and chooses the least expensive. ObjectStore also uses indexes and a query optimizer. The indexes are more complex than indexes in a relational DBMS, since they may index paths through objects and collections, not just fields directly contained in objects. The query optimization and index maintenance ideas presented here were inspired by [14]. Similar ideas on indexing and paths appear in [12, 15, 16].

Optimization techniques developed for relational DBMSs do not seem well-suited for ObjectStore. In a relational DBMS, relations are always identified by name. As a result, information about the relation, e.g., the available indexes, is available when the query is optimized, and a single strategy can be generated. In ObjectStore, collections are often not known by name. They may be pointed at (e.g., by a pointer variable or call-by-reference parameter), or result from the evaluation of an expression. This means that multiple strategies must be generated, with the final selection left until the moment the collection being queried is known, and the query is to be run.

Relational database schemas are heavily normalized—there are no such things as embedded sets or pointers. As a result, queries involve multiple tables whose contents are related to one another by 'join terms', i.e., expressions involving rows from a pair of tables (e.g., the department identifier column in the Employee table and the identifier column in the Department table). Consequently, optimizers spend most of their time figuring out the best way to evaluate queries with multiple join terms. In ObjectStore, queries tend to be over a small number of top-level (i.e., nonembedded) collections, usually one. Selection predicates involve paths through objects and embedded collections. These paths express the same sort of connections that join terms expressed in relational queries. Since the path is materialized in the database, with inter-object references and embedded collections, join optimization is less of a problem.

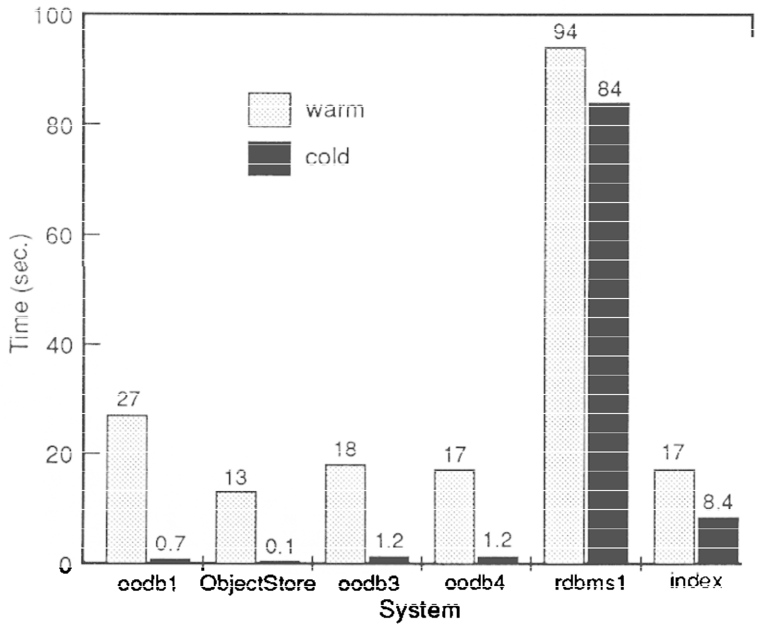
In ObjectStore, a parse tree representing the query is constructed at compile-time. Information concerning paths that appear in the query is propagated up the tree to the nodes representing queries. During code generation, a pair of functions is generated for each node in the query's parse tree. One is used to implement a scan-based strategy (visit each element and check the predicate), and the other implements an index-based strategy. Functions corresponding to query nodes also contain code to examine the collection being queried, (e.g., what indexes are present? what is the cardinality?) and make final choices about strategy. This approach allows for flexibility at run time, yet still carries out much expensive work (analysis of the query) at compile time.

If ObjectStore's C++ compiler is used, then query parsing and optimization occurs during compile time. Queries expressed using the library interface are actually parsed and optimized at run time. The same run-time library supporting query execution is used in both cases.

As noted earlier, paths can be viewed as precomputed joins. In ObjectStore, indexes can be created on paths. As a result, the join optimization problem faced by relational DBMS optimizers is replaced by a much simpler index selection problem. Analysis of the query indicates which indexes could be relevant. For example, this query finds employees who earn over $100,000 and work in the same department as Fred.

```
employees[: salary > 100000 &&
dept->employees[:
name == 'Fred' :] :]
```

There are two paths here—one on salary, and another starting at



Warm and cold cache traversal results

an employee—passing through the department of the employee, the set of employees of that department, and the name of each such employee. An index for each path either exists or it does not—the choice can be made quickly at run time. There is no need to reason about strategies based on the presence or absence of an index for each step of each path, as in a relational optimizer.

This is not to say that queries over paths avoid all query processing problems due to the presence of joins. In general, a comparison of a path to a constant (e.g., dept−> name == 'Research'), involves index selection only. Join optimization problems occur when two paths are compared, as in this query (not based on any object classes described previously):

```
projects[:
engineers[:
proj_id == works_on &&
name == 'Fred' :] :]
```

This query find projects involving Fred. There is no stored connection between projects and engineers. They are matched up by comparing the proj_id of a Project and the works_on field of an Engineer. ObjectStore would evaluate this join using iteration over projects, and an index lookup on engineers, (assuming the index is available). An index on engineers' names could also be used.

While this query is a valid ObjectStore query, it is an unusual one, and it reflects an unusual ObjectStore schema. Normally, the connection between projects and engineers would be represented by inter-object references, i.e., the join would be precomputed and stored in the participating objects. This is justified by analysis of programs in our application domains. True joins, as in the earlier query, are quite rare. For this reason we have not yet implemented join optimization. It is unusual to have queries involving multiple 'top-level' collections, (e.g., class extents) whose elements are related by comparing attributes. It is more common to have queries over a single top-level collection, with nested queries on embedded collections (i.e., queries over paths that may go through collections). The ObjectStore query optimizer reflects this.

While join optimization is less of a problem, compared to a relational DBMS, index maintenance is much more difficult. In a relational DBMS, updates affecting indexes are expressed in SQL. In ObjectStore, where the integration between the DBMS and the host language is much tighter, updates are ordinary expressions that have certain side effects. For example:

```
Person* p;
...

p−>age = p−>age + 1;
```

The assignment statement updates the age of person p. If p−>age happens to be the key to some index, then that index must be updated. It is not practical to check if index maintenance is required for every statement that modifies an object. The performance consequences would be disastrous. Instead, ObjectStore requires the declaration of data members that could *potentially* be used as index keys. Index maintenance checks are performed for these data members only. Example:

```
class Person
{
   ...
   int age indexable;
   int height;
   ...
};
```

The declaration of age as indexable indicates that updates of age need to be checked for index maintenance. Updates of height do not have to be checked. The indexable declaration does not affect type. As a result, most changes in indexability (adding or removing a declaration of indexable to an existing data member do not affect the schema of the database. (But recompilation would always be required.)

Index maintenance is further complicated by the presence of indexes on paths. For example, consider an index on children's names for a set of people. Such an index is useful for queries such as "Find people who have a child named Fred." Index updates are required when a person is added to the collection, a person in the collection has a child, or when one of this person's children changes his or her name.

Indexes on paths could be single-step, with an access method (e.g., hash table) used to represent each step of the path, or there could be one structure recording the association for the entire path. These alternatives have been discussed in [14]. ObjectStore uses a series of single-step indexes. When an indexable data member is updated, all affected access methods are updated. Then, all access methods downstream in affected index paths are updated too. Similarly, an update to a collection triggers updates that may affect all access methods of all indexes of the collection.

Applications

The performance and productivity benefits of ObjectStore have been demonstrated in a number of ObjectStore applications.

Performance Benefits

The Cattell Benchmark [5] was designed to reflect the access patterns of engineering (e.g., CASE and CAD) applications. The benchmark consists of several tests, but only the traverse test results are shown here since it best illustrates the performance benefits of ObjectStore's architecture. The test traverses a graph of objects similar to one that might be found in a typical engineering application (e.g., a schema). The graph in Figure 6 shows that the warm and cold cache traversal results when the client and server are on different machines (i.e., the remote case).

A cold cache is an empty cache, as would exist when a client starts accessing part of the database for the first time in recent history. A warm cache is the same cache after a number of iterations have been run. If the next iteration accesses the same part of the database, the cache is said to be warm. The difference between cold and warm cache times demonstrates that both the client cache and the virtual memory-mapping architecture have a significant performance benefit.

Cold cache times are dominated by the time required to get data from the disk of the server into the client's address space. Warm times reflect processing speed of data that is already present at the client and mapped into memory. We believe this to be the most important performance concern for our target application areas.

Productivity Benefits

The productivity benefits are demonstrated by the experiences of Lucid, Inc., which is developing an extensible C++ programming environment named Cadillac [7]. The environment has been under development since 1989 and will be released as a product. The system is being implemented in C++.

Before ObjectStore was available, the developers of Cadillac used a C++ object class which, when inherited, provided persistence. Classes that might have persistent instances had to inherit from this class. For each such class, methods (i.e., functions) for storing and retrieving the object from the database had to be defined. A reference to an object resulted in a retrieval from the database, if the object had not already been retrieved. While reads were transparent in that no special functions had to be called by the class user, writes had to be explicitly specified as function calls—a process that was prone to error. This mechanism was supported by a conventional Index Sequential Access Method (ISAM)-based file system.

Porting Cadillac to ObjectStore took one developer one week. The modifications were limited to three source files out of several dozen and involved, for the most part, disabling the persistence mechanism that had been in use. The simplicity of the port was due in large part to the architecture of ObjectStore, which treats persistence as a storage class rather than as an aspect of type. The conversion would have been much more difficult if functions that manipulated objects had to be modified to distinguish between persistent and transient objects.

In order to speed the porting process, the developers chose to allocate all objects in the database, even those that did not need to be persistent. Once fine-grained tuning commenced, however, objects and values that could be allocated transiently were allocated on the transient heap. Transaction boundaries were also added to shorten transactions, minimizing commit time and reducing concurrent conflicts.

The performance of Cadillac improved considerably following the installation of ObjectStore. Compilation from within the Cadillac environment ran three to five times faster with ObjectStore than with the original ISAM-based persistence mechanism. Compilation is a write-intensive operation, split into two transactions, one for each pass of the compiler. Read-intensive operations showed even more improvement, running 10 times faster using ObjectStore.

Work in Progress

Object Design, Inc. was founded in August 1988, and version 1.0 of ObjectStore was released in October 1990. Version 1.1, described here, was released in March 1991 and was the result of approximately 30 person-years of effort.

We are extending this work in a number of ways. New features under development include:

- **Schema evolution:** When a type definition changes, instances of the type, stored in the database, need to be modified to reflect the change.
- **Support for heterogenous architectures:** Some applications require access to a database from multiple architectures with varying memory layouts (e.g., different byte orderings and floating-point representations).
- **Communication with existing databases:** Many applications require the ability to access existing, nonobject-oriented databases (e.g., SQL and IMS databases). To retain the productivity benefits of ObjectStore, it is necessary to provide transparent access to these databases, i.e., through the existing ObjectStore interface.

Conclusions

ObjectStore was designed for use in applications that perform complex manipulations on large databases of objects with intricate structure. Developers of these applications require high productivity through ease of use, expressive power, a reusable code base, and tight integration with the host environment. However, even more important is the need for high performance. Speed cannot be sacrificed to obtain these benefits.

The key to meeting these requirements is the virtual memory-mapping architecture. Because of this architecture, ObjectStore users deal with a single type system. This permits tight integration with the host environment, ease of use, and the reuse of existing libraries. Other approaches to persistence taken by other object-oriented DBMSs require transient and persistent objects to be typed differently. As a result, conversion between transient and persistent representations are required, or software that had been developed to deal with transient objects must be modified or duplicated to accommodate persistent objects. In a relational DBMS, all persistent data is accessed within the scope of the SQL language with its own independent type system.

The virtual memory-mapping architecture also leads to high performance. References to transient and persistent objects are handled by the same machine code sequences. Other architectures require references to potentially persistent objects to be handled in software, and this is necessarily slower.

ObjectStore's collection, relationship, and query facilities provide support for conceptual modeling constructs such as multivalued attributes, and many-to-many relationships can be translated directly into declarative ObjectStore constructs. **C**

References

1. Agrawal, R., Gehani, N.H. ODE (Object database and environment): The language and the data model. *ACM-SIGMOD 1989 International Conference on Management of Data* (May–June 1989).
2. Bancilhon, F., Maier, D. Multilanguage object oriented systems: New answers to old database problems. *Future Generation Computers II*, K. Fuchi and L. Kotti, Eds., North-Holland, 1988.
3. Biliris, E. Configuration management and versioning in a CAD/CAM data management environment (An example). Prime/Computervision internal memo, June 1989.
4. Carey, M.J., Franklin, M.J., Livny, M., Shekita, E.J. Data caching trade-offs in client-server DBMS architectures. In *Proceedings ACM SIGMOD International Conference on the Management of Data* (1991).
5. Cattell, R.G.G. and Skeen, J. Object operations benchmark. *ACM Trans. Database Syst.* To be published.
6. Chou, H., Kim, W. Versions and change notification in an object-oriented database system. In *Proceedings of 25th ACM/IEEE Design Automation Conference* (1988).
7. Gabriel, R.P., Bourbaki, N., Devin, M., Dussud, P., Gray, D., Sexton, H. Foundation for a C++ programming environment. In *Conference Proceedings of C++ At Work.*
8. Glew, A. Boxes, links and parallel trees. In *Proceedings of the April '89 Usenix Software Management Workshop.*
9. Goldstein, I.P. and Bobrow, D. A layered approach to software design. Xerox PARC CSL-80-5, Dec. 1980.
10. Goldstein, I.P., Bobrow, D. An experimental description-based programming environment: Four reports. Xerox PARC CSL 81-3, Mar. 1981.
11. Kazar, M.L. Synchronization and caching issues in the Andrew file system. In *Usenix Conference Proceedings*, (Dallas, Winter 1988), pp. 27–36.
12. Kemper, A., Moerkotte, G. Access support in object bases. In *Proceedings ACM SIGMOD International Conference on Management of Data* (1990).
13. Maier, D. Making database systems fast enough for CAD applications in object-oriented concepts, database and applications. W. Kim and F. Lochovsky, Eds., Addison-Wesley, Reading, Mass., 1989, pgs. 573–581.
14. Maier, D., Stein, J. Development and implementation of an object-oriented DBMS. In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Eds., MIT Press 1987. Also in *Readings in Object-Oriented Database Systems*, S.B. Zdonik and D. Maier, Morgan Kaufmann, Eds., 1990.
15. Shekita, E. High-performance implementation techniques for next-generation database systems. Computer Sciences Tech. Rep. #1026, University of Wisconsin-Madison, 1991.
16. Shekita, E., Carey, M. Performance enhancement through replication in an object-oriented DBMS. In *Proceedings ACM SIGMOD International Conference on Management of Data* (1990).

CR Categories and Subject Descriptors: C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed systems;* D.3.2 [**Software**]: Programming Languages—*Language classifications;* D.4.2 [**Software**]: Operating Systems—*Storage management;* H.2.1 [**Information Systems**]: Database Management—*Logical design;* H.2.3 [**Database Management**]: Languages—*Query languages;* H.2.4 [**Database Management**]: Systems—*Query processing, transaction processing;* H.2.8 [**Information Systems**]: Database Management—*Database applications*

General Terms: Design, Management

Additional Key Words and Phrases: C++, database (persistent) programming languages, ObjectStore, object-oriented programming

About the Authors:

CHARLES W. LAMB is a member of the engineering staff and co-founder of Object Design. He was previously an employee of Symbolics, where he worked on the design and implementation of the Statice object-oriented database system.

GORDON LANDIS is a member of the engineering staff of Object Design. He was previously a co-founder of Ontologic, where he led the design and implementation of the Vbase object-oriented database system.

JACK A. ORENSTEIN is a member of the engineering staff and co-founder of Object Design. He was previously a computer scientist at Computer Corporation of America, where he conducted research on spatial data modeling and spatial query processing on the PROBE project.

DANIEL L. WEINREB is a Database Architect and co-founder of Object Design. He was previously a co-founder of Symbolics, where he led the design and implementation of the Statice object-oriented database system.

Authors' Present Address: Object Design, Inc., One New England Executive Park, Burlington, MA 01803; email: cacm@odi.com