Using the Co-existence Approach to Achieve Combined Functionality of Object-Oriented and Relational Systems

R. Ananthanarayanan, V. Gottemukkala, Georgia Institute of Technology
W. Kaefer, T. J. Lehman, H. Pirahesh, IBM Almaden Research Center

Abstract: Once considered a novelty, object oriented systems have now entered the mainstream. Their impressive performance and rich type systems have created a demand for object oriented features in other areas, such as relational database systems. We believe the current efforts to combine object oriented and relational features into a single hybrid system will fall short of the mark, whereas our approach, the *co-existence* approach, has the distinction of requiring far less work, but at the same time promising both the desired functionality and performance. We describe the attributes of our co-existing systems, an object oriented system (C++) and a relational system (Starburst), and show how this combination supports the desired features of both object-oriented and relational systems.

1 Introduction

During the last few years *Object Oriented Systems* (OOSs) have proven successful in managing complex data processing tasks inherent in applications such as computer aided design/manufacturing (CAD/CAM), geographic information systems (GIS), and computer aided software engineering (CASE). The main advantages of OOSs are their application-specific modeling power and their ability to perform data operations at memory speeds. The modeling power of OOSs stems from a rich type system that incorporates type inheritance to control both type structure and behavior. Structure inheritance allows users to create complex data structures through an evolutionary process of refining existing, less complex structures, rather than always creating completely new structures. Behavior inheritance allows users to refine the behavior of types through the use of dynamic functions, either choosing methods defined on a parent type (super-type) or defining a new (possibly refined) method to fit the needs of a subtype.

In addition to their data modeling power, OOSs offer fast pointer-based traversal of objects at or near memory speed because they not only bring the data into the *application address space*, but moreover they use *memory-based object pointers*. In contrast, relational database management systems keep the data in their own address space and allow applications to navigate on the data only through their application program interface. Furthermore, they maintain run-time interpreted *disk-oriented references* to objects (tuples), whereas most OOSs *swizzle* object pointers from a disk to memory representation once and then use the memory version of the pointer, a pointer that needs little or no run-time interpretation.

On the other hand, *Relational Data Base Management Systems* (RDBMSs) offer a different set of features, such as management of large amounts of data (terabytes), fine-grained concurrency control, and query optimization, as well as utilities for loading and reorganizing data, and generating statistics. Furthermore, robustness, failure handling, and authorization are essential issues in such data-intensive environments. However, the very primitive type systems of RDBMSs limit their data modeling capabilities to a degree which is not acceptable in the above-mentioned application areas.

In the past, an application had to choose between the offered set of OOS or RBDMS features, as no single system incorporated both. Much work is being done to produce such a hybrid system, taking the form of one of two approaches: adding relational features to an object oriented system [22], or adding object oriented features to a relational system [4]. The first approach is a potentially formidable task, as a complete RDBMS is much larger than an OOS. There is a large amount of accumulated knowledge in relational systems for managing large amounts of data, query optimization, *etc.* and to add relational features to an object oriented system would involve traversing the same "technology curve" that relational systems covered over a decade ago. It would be advantageous to leverage, rather than reimplement

[1]This work was funded by an IBM Almaden Innovation Grant.

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0109...$1.50

or even reinvent, relational technology. Similarly, the second approach, adding object oriented features to a relational system, might result in incorporating a rich type system in the RBDMS, but would require a complete rewrite of the relational storage management layer to achieve the performance of an OOS. Even then, it seems unlikely that such a hybrid system could support *object pointer traversal at memory speed.*

Neither of the two approaches brings together the desired system features without paying a high cost in the form of either effort or performance. This dilemma brings us to a new approach which may be best described by the term "co-existence." In the co-existence approach, we minimize the cost of building the overall system by taking two systems (an OOS and an RDBMS) and let them co-exist and simultaneously manage the same data. For the systems to co-exist we have to enhance both the OOS and the RDBMS with mechanisms that support some of each other's features. For example, to support the OOS's pointer based browsing, the RDBMS has to be enhanced to access the same memory as the OOS. Similarly, to model the RDBMS's relations, the OOS has to support object sets. Nevertheless, the work required to accommodate the new functionality in both systems is far less than the effort needed to combine both systems into a single hybrid system.

In this paper we describe the features needed to enhance both the OO and relational systems, as well as discuss how this system works in a client/server environment. The remainder of this paper is organized as follows: In Section 2 we discuss the architecture of our system and compare our approach to other systems. In Section 3 we present the extensions we have to make to an existing relational system to achieve our goals. Sections 4 and 5 explain the details of our implementation. Finally, in Section 6 we present our conclusions.

2 System Overview

2.1 Application Characteristics, Architecture

Our system is intended to support advanced, technical applications such as CAD/CAM, that exhibit a "load-operate-merge" behavior. This means a *working set* of data (typically in the size of megabytes) is extracted from a much larger database (typically in the range of gigabytes or terabytes) and loaded into the virtual-memory of a work-station. CAD/CAM applications usually employ complex algorithms encoded in large vendor-provided libraries written in object-oriented languages like *C++* [19]. Thus, our system is designed to manage the working set of data in such a way that C++ library routines are applicable. Besides pointer-based traversal and manipulation of data, which is the typical way object-oriented systems work on the data, applications often need to gather more abstract or statistical information about the data. Thus, our system is designed to allow for associative access to the working set such as that provided by SQL. As a consequence, we must support both an object-oriented and an associative query interface to the data. After processing is done the (modified) working set must be integrated into the original database.

This "load-operate-merge" behavior of applications seems ideally suited for a client-server architecture. The user work-stations behave as clients and download the working sets from a server that is responsible for the entire database. Once the application is done modifying the working set, the downloaded data can be shipped back to the server where it is merged into the database.

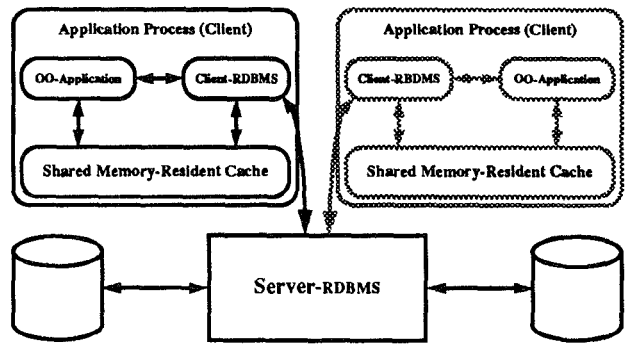

Figure 1: Proposed Architecture for the System

Figure 1 shows the basic architecture of our system. The working set resides in the *Shared Memory-Resident Cache* (SMRC). The data in SMRC can be accessed both through an OO interface and an associative query interface, *i.e.* SQL API (Application Program Interface). Thus, the DBMS is running on the work-stations as well as on the server. When an application generates a new data object it is allocated through the storage management component (SMRC) that is accessible to both the OOS and the RDBMS. This component allocates space for the object, as well as space for additional infra-structure that allows the RDBMS to access the object. Note, the object may either be allocated through C++ (the "usual" way) or through the SQL API.

2.2 Related Work

As mentioned earlier, others have taken different approaches to bring OO and Relational technology together. In this section we compare and contrast our approach and system architecture with those of systems with similar objectives.

The *ObjectStore* approach[12] takes an OO language (C++) and provides a tightly integrated language interface to traditional DBMS features such as persistence, associative queries, sets, and transaction management. ObjectStore provides type-independent persistence and set-oriented behavior through collections and queries. To implement these features in ObjectStore the memory-management scheme had to be modified (persistence), a compiler had to be developed (associative queries), and libraries had to be written (sets, transaction management, *etc*). ObjectStore introduces a new database system. Our approach is an integrated one, where the functionality of an existing database is

extended to handle OO applications. Furthermore, we *reuse* the extensive functionality of an existing RDBMS to provide traditional DBMS features, which otherwise have to be reinvented. Typically, a drawback with using an RDBMS to store persistent objects is the need to call a translation function when copying data between the fields of a tuple in the RDBMS and data members of objects in the language. However, we can avoid the difference in RDBMS and application-language data formats (termed as "impedance mismatch" in the literature) by optionally storing objects in their native format and *enhancing the RDBMS to access the native object-oriented data through user-defined member functions.*

O_2 [5] provides persistent storage for objects based on the notion of *persistent roots* which are shared between O_2 and the application. Persistent objects are stored in O_2 using a language-independent format, *e.g.* complex objects are decomposed and stored in multiple records. Thus, applications can access and store objects only after data conversion and through accessor functions provided by O_2. Associative access to persistent objects is supported *via* an extended, SQL-like query language. Thus, O_2 merges the ideas of OOS and RDBMS into one single hybrid system thereby creating its own notion of object-orientation. Furthermore, persistent objects are only accessible through accessor functions provided by O_2 which results in a performance disadvantage compared to ObjectStore, or our approach, where objects are directly accessible by the application.

Polyglot [4] takes yet another approach. In Polyglot, the objective is to add all the OO features to an existing RDBMS, which defines a new type system of its own. As is typical of type systems which support inheritance, this type system is incompatible with type systems of programming languages, such as C++. Therefore, one cannot buy vendor provided (object only) libraries written in, say, C++ and use it within the system. Instead, source code of such libraries must be adapted to the RDBMS type system. In contrast, SMRC allows existing (vendor provided) libraries to be used without change. In fact, there is a major incentive to be compatible with object-only libraries, since that is typically all that vendors provide.

A further difference between Polyglot and our system exists in the way the object-oriented functionality is implemented. Our implementation realizes most of the object-oriented features, such as inheritance and function dispatch, outside of the RDBMS. Not only does this allow for fewer changes to the RDBMS, it also potentially offers better performance since efficient dispatching capabilities of languages, such as C++, can be used. Furthermore, our method makes the system more easily extensible since it allows the system to be adapted to many application languages.

Our approach is an evolutionary approach similar to the one recommended in [17]. Our objective is to reuse as much existing technology as possible. To this end, we employ an RDBMS to provide support in areas of strength such as persistence, transaction management, and associative queries. We achieve this without forcing the OOS's type system into the RDBMS. We have identified a set of changes that need to be made to an RDBMS that, when added incrementally, give varying degrees of OO support.

2.3 Important Design Considerations

Due to the numerous aspects involved, it is beyond the scope of this paper to describe the complete design of our architecture. However, in order to understand the various design decisions made, we briefly discuss some important aspects of the overall system architecture in the following sections. In general, we concentrate on data processing aspects within the client part of our system. We illustrate our considerations using our prototype implementation based on C++ as the OOS and *Starburst* as the RDBMS. As of this writing the client part is operational, whereas the server part and client/server interaction are still under development.

2.3.1 Security, Integrity, Client/Server

As mentioned earlier, OOSs offer fast pointer-based traversal of objects at or near memory speed. This allows fast access to objects and navigation through them. In contrast, RDBMS objects (tuples) are kept in the *RDBMS address space.* As a result any manipulation of objects, or navigation through them, requires crossing address spaces. In our experience, the *address-space crossing* and the cost of navigation have been the major reasons behind the difference between performance of OOSs and RDBMSs. Thus, SMRC is designed as a part of the application's address space (we discuss the navigation issue later).

However, including the data into the application's address space may compromise its security and integrity. For example, some OOSs (*e.g.*, ObjectStore [12]) ship physical pages to the client rather than a logical unit, such as an object. So, when an application requests an object, the page containing that object is returned to the application along with other objects in that page. This compromises security of the data, because the application can read objects for which it is not authorized, or it can even modify these other objects (may be accidentally through the use of an incorrect pointer value). Other OOSs, such as Versant [15], ship whole objects instead of physical pages. This method solves the problem of object level side-effects mentioned above. However, it exposes all attributes of an object. For example, there is no way to hide the salary of an employee and expose only the name and address. Traditionally, RDBMSs avoid these security problems by maintaining different address spaces. In SMRC, we solve this problem by ensuring that only those data is copied to the client's address space which the client is authorized to access. Furthermore, we perform only logical updates on the server which are derived from the client's data. That is, we do not integrate "data containers" into the server which might be infected by the client.

2.3.2 Data Transfer between Client and Server

Shipping pages, objects, or parts of objects (attributes) from the server to the client not only affects security and integrity of the data, but also affects system performance. Object shipping typically is slower than page shipping, since objects are usually much smaller than pages. Switching to the finer granularity requires extracting objects from pages, and can multiply the number of messages between client and server. This may increase the cost by an order of magnitude. RDBMS go to the extreme of only shipping objects and within that only the requested attributes, although many such objects may be blocked into a single message. In some cases this can actually be a performance boost, since only the needed data is sent to the application. This method also provides full security and integrity, as authorization is checked by trusted server software.

In an RDBMS, all the manipulation and navigation specified by the DBMS requests must be done within the DBMS address space. Some systems [9] use a mechanism of application code, sometimes called stored procedures, run inside the RDBMS address space to avoid crossing the application to server boundary. If, however, an RDBMS application does not exploit such a mechanism, it will suffer performance degradation due to RDBMS address space crossings.

The decision to ship pages, objects, or parts of objects depends on how the object-oriented data is represented on the server, *i.e.* within the RDBMS. In the following section, we investigate three different mapping methods and their implications to data transfer, data security, data integrity, and performance.

2.3.3 Relational Representation of OO Data

There are several methods of mapping object-oriented data (*i.e. objects*) to relations (*i.e. tuples*) which offer different advantages and disadvantages. Thus, we allow the *application to choose the appropriate mapping method for each object type or each object.* Additional information in the class definition of the objects is used to determine which mapping method is used for an object type. Furthermore, in some cases an additional parameter given to the `new` operator specifies if the object is persistent (*i.e.* stored in the database) or not. We offer the following three mapping methods:

1 *Basic mapping*: An object type is mapped to a single-attribute relational table, *i.e. an object is stored within a relational tuple as one attribute value* [1]. The object remains in its native OOS format, consequently allowing the application to directly apply the object's member functions on an object pointer. Thus, the OOS can directly access the data without any additional costs. Furthermore, storing the data in this way does not compromise the efficiency of the OOS data allocation, since only some additional amount of storage is allocated and initialized. On the other hand, the RDBMS can access the objects as tuples, but it has to apply the OOS accessor and mutator[2] functions to capture the behavior of the data-members of the objects (see [7] for related optimization issues). Thus, all the set-oriented functionality of the RDBMS (*e.g.* concurrency control, recovery, authorization, *etc.*) is applicable to the data. Only for the attribute-oriented functionality (*e.g.* associative queries, access path[3] support) does the RDBMS have to apply the OOS member functions to access the data. Note that such queries can be run both on the client or the server side.

We preserve full authorization and data integrity of RDBMS, because we update the server using the client's changes as new values. *Attribute value security* (*e.g.* hiding of `salary`) can be done by using the `salary` accessor function in the select list, instead of shipping the whole object. Obviously, if whole objects are shipped, the security coverage is identical to the earlier mentioned object shipping techniques of OOS, such as Versant [15].

2 *Structured mapping*: An object type is mapped to a multiple-attribute relational table (similar to O_2 [5]), *i.e.* for each attribute definition within the object type there exists a corresponding attribute within the relation. Thus, the full functionality of the RDBMS is available on the data. However, in this case the access to the data *via* the OOS becomes more costly, because it's unlikely that the RDBMS and the OOS share a common storage format for the tuples/objects. Therefore, the OOS has to use the access functions supplied by the RDBMS to access attributes of the object. Again, since on the client side the RDBMS runs within the address space of the application, these access functions are similar to member functions, *i.e.* procedures. This technique allows *attribute level security* (*e.g.*, hiding of `salary`), since only the authorized attributes are sent to the client address space. Another advantage of this approach is that only referenced attributes within a query need to be moved during query processing.

3 *Clustered mapping*: A set of objects is mapped to one attribute of type *binary large object* (BLOB) on the RDBMS side. This significantly limits the available RDBMS functionality, but provides the best performance on the OOS side. Shipping the BLOB data between client and server corresponds to shipping of *page sets.* However, authorization and integrity is still checked for the BLOB data by the RDBMS which has many advantages compared to page shipping done by OOS, such as ObjectStore [12].

We have already discussed the implications of the mapping methods in terms of functionality on the OOS side and the RDBMS side as well the implications concerning the data transfer between client and server.

[1] An obvious extension of basic mapping is using several attributes of a tuple each storing one object.

[2] We refer to functions which read or write the values of private attributes of an object as *member functions.*

[3] The mutator functions for these attributes must ensure access path maintenance.

But, we have another important aspect to discuss. Usually, the objects in CAD/CAM applications exhibit a high degree of interconnection. Thus, supporting these connections is mandatory. One of the major advantages of OOS is the ability to do *pointer swizzling.*

2.3.4 Pointer Swizzling

Often, linking of objects is erroneously cited as the major advantage of OOSs, but links alone are not responsible for the superior performance displayed by OOSs. The technique of linking objects has been exploited in RDBMS. For example, links were first introduced in System R [2], in IBM DB2 [8], and in *Starburst*'s IMS attachment [3]. However, in RDBMSs, the link representation itself does not change, regardless of whether the data is on disk or buffered in main memory. An important advance of OOSs is the introduction of *pointer swizzling* [21, 12], which converts disk pointers to virtual-memory pointers upon loading of the data. Swizzling pointers during loading/unloading of objects requires some kind of *persistent pointers.* Some OOSs use the virtual memory address of the object as the basis of a persistent pointer, because moving of objects within the address space is often not necessary. In contrast, RDBMS must use some kind of indirection to address a tuple, as tuples may move as a result of dynamic schema modifications, data reorganization, and length changing updates. We discuss the dependencies between the type of persistent pointers and the swizzling scheme in section 5.3. But, it should be noted, that the mapping of objects to a tuple, to a column, or within BLOB data affects the type of the persistent pointer as well as the swizzling scheme.

3 Relational Access to OO Data

As mentioned earlier, the object-oriented data may be stored in its native format or in RDBMS format. In this section, we deal with the former case, *i.e.* an object is stored within the RDBMS as *one column of uninterpreted bytes.* Since the RDBMS cannot understand the internal structure of an object, it has to apply member functions defined by the OOS in order to access the attributes of the object. Thus, the RDBMS has to provide for user-definable functions. We have identified two additional features, namely *distinct types* and *memory control,* which have to be supported by the RDBMS in order for it to co-exist with an OOS. Surprisingly, *inheritance* or *substitutability* can be handled outside the RDBMS, as part of the adaptation code. In the following, we describe each of the mentioned features in more detail.

1 *User-definable Functions*: The following example shows a simple class definition with two member functions and its relational counterpart.

C++ definition:

```
class Rectangle { float len; float wid;
public:
    float length() { return (len) };
    float width() { return (wid) };
    ... };
```

RDBMS definition:

```
create table RectangleData
    (this bytes (sizeof(Rectangle) ) );
    /* register length and width */
create view Rectangle (length, width) as
    (select length(this), width(this)
    from RectangleData);
```

The class `Rectangle` contains two private attributes, called `len` and `wid`, describing the length of the sides of the rectangle. The values of these two attributes are visible only through calls of the member functions, called `length` and `width`, respectively. In order to address objects of this type *via* the RDBMS, *i.e. via* SQL, we have to define a relational representation for the object type. We define the relational table `RectangleData` as a single-attribute table containing an attribute of type `bytes`[4] to store objects of type `Rectangle` (basic mapping method). In order to address the length or width of a rectangle *via* SQL the RDBMS has to call the member functions of class `Rectangle` which must be registered as user-defined functions within the RDBMS. The view definition shown above (a) illustrates how to use user-defined functions in an SQL statement and (b) facilitates querying the object-oriented data *via* SQL.

2 *Distinct Types*: Relational access to object-oriented data is achieved through introduction of the C++ member functions into the RDBMS. However, C++ allows for the definition of functions which have the same name but different numbers or types of parameters. This becomes especially apparent in the context of inheritance and dynamic functions (we discuss this in section 4.2). For now, we use a simpler example — we define two functions which return the area of a rectangle. One function extracts this information given a rectangle whereas the second function needs the lengths of its sides as input (here we use the user-defined type `sidelen`):

```
float area (Rectangle r) { ... };
float area (sidelen x, sidelen y) { ... };
```

The RDBMS has to dispatch the functions using their *signatures.* The signature of a function serves as its unique identifier and therefore combines the function name with its parameter and return types. Thus, the RDBMS must know about the types in the C++ program with which the member functions are defined. In our example we have to register the type `sidelen`, defined in the C++ program, as a type within the RDBMS. Basically, `sidelen` is just another name for the data type `float`, *i.e.* `sidelen` is a so-called *distinct type* [10]. Distinct types allow the RDBMS to differentiate between user-defined types based on the same basetype, *e.g.* defining a second distinct type `size` as `float` allows the RDBMS to differentiate between the *size* of an rectangle and its *side lengths.*

[4] In C++ objects have a fixed length; in general we may define the attribute's type as a bytestring of variable length.

3 *Substitutability*: The notion of *substitutability* is closely related to the concept of *inheritance*. Substitutability allows for the usage of an instance of a subtype whenever an instance of a type is required. This applies for *parameter substitution* as well as for *references*, *i.e.* a reference may point to an instance of the required type or of an instance of a subtype of the required type. Type substitutability complicates function resolution, because along with subtypes often the methods defined for the supertype are also refined. Thus, if we pass a subtype to a function, then the refined implementation of the function should be executed. Functions, which react in this way to the type of their actual parameters, are called *dynamic functions* (see section 4.2.2 for details). Support of dynamic functions has two consequences. First, every object must be *tagged with its exact type*, and, second, function resolution of dynamic functions must be done at *runtime*. We use the technique of gateway functions (see section 4.1.1) to simulate the C++ function dispatch. Thus, this specific kind of function dispatch is realized *outside* the RDBMS.

4 *Memory Allocation*: *Features 1 through 3* are concerned with enhancing the RDBMS to work on object-oriented data. However, in order to address the data *simultaneously* through the OOS and the RDBMS we have to modify the memory allocation of both systems, which is done using the SMRC storage component. We discuss this in section 5.

It should be noted here, that *features 1 through 3* allow object-oriented access to the OOS data *via* the RDBMS. This holds for the RDBMS running on the client side as well as for the RDBMS running on the server side. *Feature 4* adds the ability to *simultaneously* access the data *via* the client RDBMS and the OOS. In the following section we discuss the implementation of *features 1 through 3*, and in section 5 we discuss the implementation of *feature 4*.

4 Realizing Relational Access to OO Data

For our implementation we choose *C++* as the OOS and *Starburst* as the RDBMS. *Starburst* is an extensible relational database prototype developed at the IBM Almaden Research Center [6]. We choose *Starburst* because it supports experimentation at all levels of the system. In this section we describe how we realized *features 1 through 3* with respect to *C++* and *Starburst*.

4.1 Adding User-defined Functions

Consider a user-defined function such as *length* or *width*. The implementation of such a function is specified by the user as part of the application program. Thus within the compiled application program the function has a *load-address*. In order to make this function implementation callable from SQL, *i.e.* from *Starburst*, we have to register the name of the function in the function catalog of *Starburst* and we have to provide a mechanism that allows *Starburst* to call a user-defined function during query execution.

4.1.1 Execution of User-defined Functions

The implementation of a user-defined function is specified using a host language such as C++. There are two characteristics of the host language that affect the implementation of user-defined functions in an RDBMS.

1 *Data representation*: Every host language chooses a representation for its basic data types (such as integers) and aggregate data types (such as arrays). The representation may depend on the target machine on which the language is implemented. To divorce itself from such machine-dependent characteristics, an RDBMS such as *Starburst* maintains its own representation.

2 *Procedure call linkage*: The host language also follows a procedure call/return convention that depends on the target machine. This convention is in terms of register and stack format when program control transfers from one procedure to another. Again, to avoid such machine-dependent handling, *Starburst* uses a different generic calling convention, called *XPR linkage*.

A user-defined function expects data representation and procedure call linkage of the host language implementation. To satisfy this requirement, *Starburst* uses a *gateway* function to convert data between C++ and *Starburst* representations, and also between XPR linkage and C++ procedure call linkage. Gateway functions are implemented in C++ and are automatically synthesized by a preprocessor. The preprocessor consists of a parser for function signatures and a generator which produces the source code for the gateway functions.

For example, consider the gateway function for the user-defined function **area**, implemented in C++. Let the signature of **area** be:

float area (**float**, **float**);

The corresponding gateway function for **area** is as follows.

```
Result area_gateway(<xpr-linkage>)
  float temp1, temp2, result;
  temp1 = convert_from(<input[0]>, float);
  temp2 = convert_from(<input[1]>, float);
  result = area(temp1, temp2);
  <output> = convert_to(result, float);
```

`convert_from(value, type)` converts `value` in *Starburst* data representation into the representation specified by host language basic type `type`. `convert_to()` performs the reverse operation. Simple atomic data types such as `int`, `float` and `char` in C++ are represented by the corresponding Starburst data type. We use bytestrings in *Starburst* to represent user definable data types such as records (`struct` in C++) and arrays. The format of the bytestring is the same as the data representation of C++ (strictly speaking, the representation as dictated by the particular implementation of C++) with respect to padding and alignment of components of the type. As mentioned earlier objects may also

be represented as bytestrings in C++ format. With the above representation, function calls on objects need not be altered since the data representation is in the format of the language. Hence, data conversion routines are needed only for converting *Starburst* representation of simple data types to that of C++.

4.2 Realizing Inheritance

4.2.1 Compile-time Function Dispatch

Functions can be classified according to inheritance characteristics of the classes. A *base function* of a class is one that is defined in the class itself. An *inherited function*, on the other hand, is one that is not defined in the class, but by some ancestor of the class, and is applicable to the class due to inheritance rules of the language (Figure 2). Given a set of classes arranged in a hierarchy (a lattice in case of *multiple inheritance*) and the corresponding set of base function signatures of each class, the *closure of functions on inheritance* is obtained by applying the rules of inheritance to the hierarchy and pushing the functions applicable to a class down to its descendants. Then all applicable functions for a class are registered as user-defined functions within the RDBMS. Thus, inheritance is realized through a scheme in which functions appear to be flat to the RDBMS (*i.e.* without any associated inheritance hierarchy). As mentioned before, this scheme allows adaptation of the RDBMS to different type inheritance models, such as those of C++ and SQL [10].

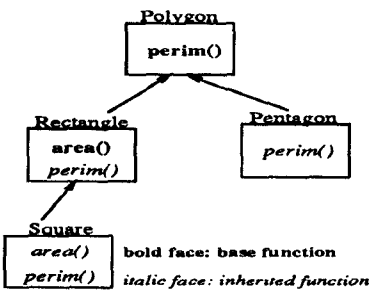

Figure 2: Closure of functions on inheritance

4.2.2 Dynamically Dispatched Functions

The address of a *static function* can be resolved *at compile time*, based on the types of the parameters of the actual function call. In the case of *dynamically dispatched functions*, the actual function address for a corresponding call is not known until *run-time*. For example, a pointer to an object of a type can hold the address of an object of that type or any of its sub-types. If a parameter to a function is such a pointer, the address of the function to be called is determined at run-time by examining the type of the corresponding object. In the following, we describe a scheme to implement *virtual functions*, the C++ term for dynamically dispatched functions.

In C++, virtual functions are implemented using object-type specific arrays of load-addresses of virtual functions, called *V-tables*. Each object stores a pointer to the V-table associated with its type. Furthermore, an index is associated with each virtual function. Thus, when a pointer to a class is used to point to an object of its sub-class, the appropriate function will be called. This scheme is problematic for a database system, as an object with an associated V-table may outlive the process that created it and hence the V-table would no longer be valid. Thus, the V-table pointer of an object retrieved from the database stable storage needs to be changed so that it points to the V-table of the class to which the object belongs in the context of the current process.

4.2.3 Type-Tags and Vtables

For each object in the RDBMS, we maintain a *type-tag* that uniquely identifies a C++ class. For each class in the application program, the preprocessor declares *dummy variables*. During program initialization we construct a look-up table called the *V-table look-up table* (VTLT). The VTLT stores pointers to the dummy variables, and is indexed by class type-tags. The key idea is that the dummy variables that are compiled with the application program contain valid V-table pointers with respect to the address space of the process. If a function is called on an object which contains a V-table pointer, the corresponding gateway function looks up the correct V-table pointer using the type-tag as the key into VTLT, and copies the address into the object.

So far, we are able to manage and query object-oriented data within the RDBMS while preserving encapsulation and object behavior. However, we are not yet able to address the data simultaneously through the OOS and the RDBMS. This requires changes to the memory management components of both systems. We will require both systems to use a shared storage component, the *Shared Memory Resident Cache* (SMRC). SMRC offers both an OOS and an RDBMS interface. Thus, we replace the `new` operator of the OOS with the one provided by SMRC. SMRC allocates the storage and adds further information so that the object is accessible as a tuple to the RDBMS[5]. The following section describes the design of the SMRC.

5 Design of the SMRC

The purpose of the SMRC is to allow access to its data through the OOS as well as through the RDBMS and to speed up the relational access by taking advantage of the main memory nature of the data. Furthermore, SMRC is responsible for the exchange of data between the client's address space and the server. Thus relocation of data into SMRC requires *swizzling* of data addresses. In this section, we discuss the design of SMRC. First, we show how SMRC satisfies the relational interface to its data. Second, we discuss support for set-

[5] `New` operators also attach type-tags to objects.

oriented OOS access. Most OOS use the notion of collections (*e.g.* [12]) to refer to a set of objects. Thus, at a first glance collections are similar to relations. However, collections offer some additional variations. For example, collections may maintain an order (lists) and often contain only pointers to objects, not the objects themselves. Thus, SMRC offers special support for collections. At the end of this section, we discuss some issues of pointer swizzling, object identifiers and references.

5.1 Shared Data Cache (SDC)

Starburst's Data Management Extension Architecture (DMEA) [14] formalizes the interface for creating new *storage methods* and *attachments*. Storage methods provide the means for storing the tuples of a relation, whereas attachments provide the means for managing data structures about relations (*e.g.*, indexes). Consequently, SMRC is designed as a storage method, the *Shared Data Cache* (SDC) storage method, in *Starburst*. As a result, the data stored in SDC is accessible *via* the usual relational operators of *Starburst* during query execution. In fact, the design of SDC is very similar to that of *Starburst*'s MMM storage component, described in [13], which supports memory-resident relations. But, in addition to the MMM functionality, SDC provides OOS with a procedure call interface thus avoiding the long access path to an attribute passed through all the involved RDBMS layers. Moreover, SDC provides direct access to its tuples and attributes without crossing address space boundaries.

Applications directly allocate memory from SDC through overloading of the `new` operator. However, the object is allocated in a way that it is accessible as a tuple through the RDBMS. To this end, memory in SDC is composed of segments which in turn are composed of partitions. Memory for objects is allocated from different segments based on the type of the objects. SDC also supports clustering at object allocation time. *Clustering allows applications* to group (heterogeneous) objects that will be accessed together. Thus, clustering provides the application the benefits of locality of reference both in server I/O and client-server interaction. We represent a cluster as a set of partitions that belong to one or more segments.

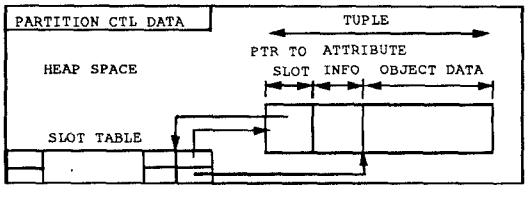

Figure 3: Storage Format for Objects in SMRC

Each object in SDC is stored (through the overloaded `new` operator) in *Starburst*'s tuple format (Figure 3) so that it can be treated as a tuple when relational access to that object is required. Since SDC is a memory-resident storage component the objects in SDC can be referenced by virtual memory (VM) pointers. However, the VM pointers are transient and are valid only while the partition is mapped at a particular VM address. In contrast, the segment, partition, and offset ($< S, P, O >$) triplet, often referred to as the *tuple identifier* (TID), provides a stable and persistent identifier of the object (cf. section 5.3). Thereby, the offset refers to a slot in the slot table of a partition (Figure 3). Each slot contains two pointers. One references the object and can therefore be used as any object pointer in C++. The other references the tuple (*i.e.* some information stored in front of the object) which allows *Starburst* to treat the data as any other tuple within the database (the object's data is treated as an attribute value). Similarly, along with the tuple/object, a pointer to its slot in the partition is stored. This is also different from the earlier mentioned MMM storage method which supports only access of the tuple given the TID. Maintaining a pointer from the tuple to its TID makes it easy to convert between transient (VM-pointer to the tuple) and persistent (TID) object handles.

5.2 Collection Storage Method (CSM)

OOSs use a *collection*, which can be thought of as a set of pointers, to refer to a set of objects of one type or subtypes of that type. *Enumerators* or, in terms of RDBMSs, *cursors* allow to visit all objects of a given collection. The purpose of the collection storage method (CSM) is to provide an efficient cursor for OOS access as well as relational access to collections. Collections can be categorized by the way they are created and maintained as follows:

1 *Base Collection*: A base collection represents a *class extent*, which is the set of all instances of a class. Thus, if we store all objects of one type, *i.e.* of one class within one SDC table, the relation itself constitutes the base collection. However, the application may choose to scatter objects of one type over several relations. In this case, we have to maintain the class extent explicitly. Whether a base collection should be created is specified in the class template.[6] When any instance of a class with a base collection is created, its reference is automatically inserted into that base collection.

2 *Restricted Collection*: Restricted collections are similar to views in relational systems. A restricted collection is created by specifying a source collection or a relation and a predicate to restrict the selection. References to all the qualifying objects are inserted into the restricted collection. For example, when the predicate `area() < 100` is applied on `Rectangle` we obtain a restricted collection of small rectangles.

3 *User-Managed Collection*: The user/application explicitly adds and deletes references of objects to a user-

[6] By default, no class extent is maintained.

managed collection. For example, suppose an application wants to create a collection of "well formed" rectangles and assume there is no predicate which, when applied to a rectangle, determines if it is "well formed." In this case the application would manually create the `Nice-Rectangle` collection and insert the "well formed" rectangles into the collection.

Collections are implemented by the CSM in the form of single-column relations that contain references to the objects in that collection. The references used are the persistent object handles described in Section 5.1. Elements of collections also have persistent TID, *i.e.* $< S, P, O >$ handles, that allow us to treat collections as first-class entities, so we can create new collections from existing collections.

Scanning a collection through the API interface would be very inefficient compared to an OOS, because some computations are needed to find the next element in the SDC table. Thus CSM maintains a doubly linked list connecting all tuples stored in the CSM, *i.e.* besides the single column that holds the references to the objects, additional columns are added that hold *next/previous* pointers which point to the next/previous tuple in the collection table. These pointers are maintained by the storage method and are updated whenever an object is added to or deleted from the collection. Since main memory pointers are valid only in the address scope of the actual process they have to be adjusted if the data is shipped between address spaces, *i.e.* between client and server. The process of adjustment is called *pointer swizzling.*

5.3 Swizzling, References, and Object Ids

Swizzling: Pointer swizzling [16, 11, 20], *i.e.* the transformation of disk-based pointers into virtual-memory pointers (and vice versa), offers two advantages. First, it offers performance gains since the pointers need not be interpreted for each traversal. Second, which might be more important, some swizzle techniques (e.g. [21]) allow applications to work with persistent pointers exactly the same way as with common C++ pointers. Thus, it is possible to be object code compatible to C++ libraries; libraries which were not created to be used in a database environment. With respect to the second argument, we intend to use such a swizzling scheme for our clustered mapping method, where we store a set of objects as a BLOB.

Every time an object is allocated in a BLOB we record the object's address in an *object address table* that is stored within the BLOB along with the current load address of the BLOB within the application's address space. Moving the BLOB to the server only requires checking for pointers referring to transient data; this pointer must be invalidated on the server side. Moving a BLOB from the server into application's address space may need additional work. If the BLOB is located on the same VM address as before than no further work is necessary. Otherwise, we compute the offset between the current load address and the previous load address and adjust all pointers within the BLOB.[7] We can implement this algorithm with a similar mechanism as DB2's *field procedure* [8]. DB2 invokes a field procedure when data is moved from disk to memory or vice versa. One usage of field procedures is data compression/decompression. Analogously, in the SMRC environment field procedures should be invoked whenever data goes across a process boundary. Thus, our swizzling scheme can be easily implemented using existing techniques. However, this swizzling mechanism is sufficient only if the BLOB is self-contained, *i.e.* if there are no pointers referring to objects outside the BLOB. We call such pointers *references.*

References: References are persistent pointers to objects that can be used on the server to link objects. Thus, we provide a *dereference* function which locates an object given a reference to it.[8] In this sense, a TID is a simple kind of reference whereas a foreign-key is not, because a foreign key does not indicate which table is referenced. However, in our environment we need more information for references. In order to locate an object within a BLOB we have to know the tuple which the BLOB belongs to, further we need the column within the tuple representing the BLOB, and within the BLOB the address of the object. Thus, a reference is a quintet (*Count* will be explained below): $(< TID, Column, Offset, Count, VM - address >)$. This kind of address allows us to locate an object regardless of the mapping method used, *i.e.* regardless of mapping the object to a tuple (*Column* and *Offset* are omitted), to a column (*Offset* is omitted), or to a BLOB. *Offset* refers to the offset of the object within the *object table.* This allows compaction of the BLOB without invalidating references to the objects within the BLOB. On the other hand, we have to invalidate all references to an object if it's deleted. Searching all references and invalidating them poses a number of problems in large DBs. The database could be partitioned across many nodes. Often, not all the nodes are accessible at the time deletion happens. Furthermore, referencing objects could be locked by other transactions. In addition, finding all such references could be very expensive. Thus, we have to make sure that the reference is invalid when it's dereferenced. We use the *Count* for this purpose, *i.e.* when an object is deleted and a new object is created at the same address we increment the associated *Count.*[9] The *dereference* function checks both fields and returns an error in case of a discrepancy. Otherwise, the VM-address of the object is returned and the *VM-address* within the reference is updated. Further calls

[7] This swizzling method allows us to be binary compatible with existing C++ libraries.

[8] This requires overloading of the C++ *dereference* function; in this case our systems provides only source code compatibility. However, explicitly dereferencing is needed only for pointers going outside a BLOB.

[9] The Memory manager allocates objects in such a way that the location of the *count* field of a deleted object is not used for any other purpose.

of *dereference* would then return this address.

Object Identifiers: In our environment it is not necessary to expose the object identifier, *i.e.* the quintet, to the application. Instead, only swizzled references (VM-pointers) are used within the application to modify relationships. Thus, we may change the quintet for reasons of reorganization. Furthermore, in many applications primary keys are used instead of object identifiers.

6 Conclusion and Project Status

In addition to CAD/CAM, we see other application areas, such as those from the medical and scientific communities, that were once users of relational systems but are now branching out into complicated (non-relational) types that represent multi-media objects, such as documents, images, and visualizations. Relational database systems will either evolve to support these new types, or they will become legacy systems as applications evolve in using the newer technologies (such as object oriented systems) that can handle the new types. Similarly, many applications currently using object oriented systems for their rich type systems and high performance navigation capabilities, such as those involved in 3-dimensional visualization, are finding that set oriented queries provide a much needed balance to a predominantly navigation-oriented system. Over time, an increasing number of applications will have database system requirements of high performance, a rich type system, and a set oriented query capability, just as many CAD/CAM applications require today.

In this paper, we have described the *co-existence* approach to creating a system that supports both object oriented and relational features without the cost of building a new single hybrid system. The key to co-existence is the SMRC component that provides a dual object-oriented/relational interface to the same data, *simultaneously*. It not only provides the high performance object oriented access that one would expect, but since it is a memory-resident database component that runs in the address space of the client, it also significantly improves the performance of relational access to client data as well. The fact that SMRC supports a standard relational storage component interface implies that a server (assuming our "basic mapping") can use the exact same user defined functions to query server data that the client uses to query client data. This is a capability that many other systems lack, as most object oriented client/server systems (ObjectStore, Versant, *etc.*) do not allow queries to be performed on the server.

We predict that, in the near future, successful mainstream database products will have to support both object oriented and relational features. The question now is, how are they going to do it? The *co-existence* approach reduces development time significantly by leveraging existing object oriented and relational technology rather than re-implementing it. In fact, in only 7 person-months we were able to build a working client component that displays the desired object oriented and relational features thereby leveraging from *Starburst*'s extensibility features. Furthermore, we are able to run relational queries on both client data and server data. Now that our client implementation is mostly complete, our efforts are focussed on general-purpose (multi-node) client/server interaction, and we are going to implement the various mappings and swizzling protocols.

Acknowledgments We would like to thank Rakesh Agrawal, C Mohan, Inderpal Narang, Peter Schwarz, and Joel Richardson for many useful discussions. Many of our colleagues in the Starburst project deserve a special thanks for providing a very stimulating environment. Thanks are also due to Pat Selinger, Jim Stamos, and Bill Cody for their comments on an earlier version of this paper.

[1] contains a longer version of this paper.

References

[1] R. Ananthanarayanan, V. Gottemukkala, W. Kaefer, T.J. Lehman, and H. Pirahesh. SMRC: Incorporating Object-Orientation into Starburst. Research Report, IBM Almaden Research Center, 1993. under preparation.

[2] M. Astrahan, M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones, J. Mehl, G. Putzolu, I. Traiger, B. Wade, and V. Watson. System R: Relational Approach to Database Management. ACM Transactions on Database Systems, 1(2):97–137, June 1976.

[3] Mike Carey, Eugene Shekita, George Lapis, Bruce Lindsay, and John McPherson An Incremental Join Attachment for Starburst. In Proc. 16th International Conference on Very Large Data Bases, Brisbane, August 1990

[4] L. DeMichiel, D. Chamberlin, B. Lindsay, R. Agrawal, and M. Arya. Polyglot. Extensions to Relational Databases for Sharable Types and Functions in a Multi-language Environment In Proc. DE 93 [18].

[5] O. Deux and et al The O_2 System. Communications of the ACM, October 1991.

[6] L.M Haas, W. Chang, G.M Lohman, J McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst Mid-Flight: As the Dust Clears. IEEE Transactions on Knowledge and Data Engineering, pages 143–160, March 1990.

[7] J.M. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates In Proc. ACM-SIGMOD International Conference on Management of Data, Washington, May 1993.

[8] IBM IBM Database 2 Administration Guide, March 1992.

[9] Informix. Guide to SQL Reference, December 1991.

[10] ISO_ANSI. ISO-ANSI Working Draft: Database Language SQL2 and SQL3; X3H2, ISO/IEC JTC1/SC21/WG3, 1992.

[11] A. Kemper and D. Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases. In Proc. DE 93 [18].

[12] Chares Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The Objectstore Database System Communications of the ACM, October 1991.

[13] T Lehman, E Shekita, and L.F. Cabrera. An Evaluation of the Starburst Memory-Resident Storage Component. IEEE Transactions on Knowledge and Data Engineering, December 1992.

[14] Bruce Lindsay, John McPherson, and Hamid Pirahesh. Data Management Extension Architecture In Proc. ACM-SIGMOD International Conference on Management of Data, pages 220–226, San Francisco, May 1987.

[15] Mary Loomis Client-Server Architecture. Object-Oriented Programming, February 1992.

[16] J.E.B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle IEEE Transactions on Software Engineering, 18(3), August 1992.

[17] Hamid Pirahesh and C. Mohan. Evolution of Relational DBMSs Toward Object Support A Practical Viewpoint. In Proc. Datenbanksysteme in Buro, Technik und Wissenschaft, Kaiserslautern, March 1991. Springer-Verlag. Also available as IBM Research Report RJ 8324, IBM Almaden Research Center, September 1991

[18] Proc. 9th IEEE International Conference on Data Engineering, Vienna, April 1993

[19] Bjarne Stroustrup The C++ Programming Language. Addison-Wesley Publishing Co , 1987

[20] S.J. White and D.J. DeWitt. A Performance Study of Alternative Object Faulting and Pointer Swizzli ng Strategies In Proc. 18th International Conference on Very Large Data Bases, Vancouver, September 1992.

[21] Paul Wilson. Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware. Electrical Engineering and Computer Science Technical Report UIC-EECS-90-6, University of Illinois at chicago, June 1990.

[22] Stanley Zdonik and David Maier. Fundamentals of Object-Oriented Databases In Stan Zdonik and David Maier, editors, Readings In Object-Oriented Database Systems. Morgan-Kaufmann Publishers, Inc , 1990.