# Storing and using objects in a relational database

by B. Reinwald
T. J. Lehman
H. Pirahesh
V. Gottemukkala

***In today's heterogeneous development environments, application programmers have the responsibility to segment their application data and to store those data in different types of stores. That means relational data will be stored in RDBMSs (relational database management systems), C++ objects in OODBMSs (object-oriented database management systems), SOM (System Object Model) objects in OMG (Object Management Group) persistent stores, and OpenDoc™ or OLE™ (Object Linking and Embedding) compound documents in document files. In addition, application programmers must deal with multiple server systems with different query languages as well as large amounts of heterogeneous data. This paper describes SMRC (shared memory-resident cache), an RDBMS extender that provides the ability to store objects created in external type systems like C++ or SOM in a relational database, coresident with existing relational or other heterogeneous data. Using SMRC, applications can store and retrieve objects via SQL (structured query language), and invoke methods on the objects, without requiring any modifications to the original object definitions. Furthermore, the stored objects fully participate in all the characteristic features of the underlying relational database, e.g., transactions, backup, and authorization. SMRC is implemented on top of IBM's DB2® Common Server for AIX® relational database system and heavily exploits the DB2 user-defined types (UDTs), user-defined functions (UDFs), and large objects (LOBs) technology. In this paper, the C++ type system is used as a sample external type system to exemplify the SMRC approach, i.e., storing C++ objects in relational databases. Similar efforts are required for SOM or OLE objects.***

In recent years, object-oriented (OO) technology has achieved wide acceptance, maturity, and market presence. An OO application development project often starts with established OO tools, class libraries, and object frameworks,[1] followed by a customization step, and then is enhanced and refined by using features such as inheritance and encapsulation. This new programming paradigm has significantly improved both the programmer's productivity and the timeliness and cost of application development. It is the growing interest in OO applications, coupled with the attractive features of relational database management systems (RDBMSs), that led to the advent of extended RDBMSs, e.g., systems like Postgres and Starburst, as well as object-oriented database management systems (OODBMSs), e.g., systems like ObjectStore**, O2**, GemStone**, and Versant**.[2–4] Since these systems were established, OODBMSs have matured significantly, creating a market presence and increased market share. At the same time, RDBMS vendors saw some of the same OO trends and subsequently developed object-relational database management systems (ORDBMS), e.g., systems like UniSQL**, Illustra**, and DB2*.[5–7] RDBMSs continue to dominate the database market, and market analysts expect that this trend will continue.

Many users of RDBMSs are expanding toward applications that require more effective handling of non-traditional data, such as text, voice, image, and financial data. It is no surprise then, that most users




0018-8670/96/$5.00 © 1996 IBM

also desire their OO data to be stored in their databases without compromising the essential industrial-strength features of RDBMSs that they already rely upon. Such features include robustness, high performance, standards compliance, and support for open systems, security, bulk I/O capabilities, and different levels of concurrency and isolation. As a result, there is constant pressure on RDBMS vendors to provide additional functionality for storing objects that were created in the external type system of an OO programming language. This functionality goes beyond user-defined types (UDTs), user-defined functions (UDFs), and large objects (LOBs) in SQL3.[8] UDTs extend the relational type system with new data types, based on the relational built-in data types. The UDF mechanism provides a way to add functions to the existing base of relational built-in functions. LOBs give the RDBMS a way to manipulate large data objects, typically for multimedia applications. Although the addition of UDTs, UDFs, and LOBs to an RDBMS increases its functionality, these new features do not match the functionality of classes, methods, and objects in an OO programming language like C++.

This paper describes the shared memory-resident cache (SMRC) prototype implementation, at the IBM Almaden Research Center, that stores C++ objects in an RDBMS (e.g., DB2 Common Server for AIX*) by exploiting the UDT, UDF, and LOB technology.[9,10] The design and implementation of SMRC[11] was especially driven by the following requirements and goals:

- The approach must be compatible with existing class libraries; thus there is no opportunity to inherit persistence properties from a common root object and modify class definitions to include additional constructors or add methods to support persistence properties.
- The objects must be accessible in SQL (structured query language) queries as the existing relational data.
- The methods of acquired class libraries must be usable within SQL queries.
- The performance of queries involving objects must be reasonable. This is particularly an area of concern where methods, used within query predicates, are applied to millions of database records. If the predicate evaluator is inefficient in invoking methods of objects, then when invoked millions of times on objects, the response time will be unacceptable.

SMRC (mostly)[12] achieves the above goals by exploiting advanced features of RDBMSs and by providing an efficient binding to bridge the gap between objects of external type systems and RDBMSs in an attractive and inexpensive way. By external type system, we refer to types defined in C++, which are different from the tables and fields defined in SQL. Using SMRC, C++ objects are stored in the database in the same binary format as they were created in the C++ client application language. Thus, no translation of C++ class definitions to relational schemata and no data conversion needs to be performed. Standard SQL is used to store and retrieve the C++ objects in the relational database. When retrieving an object from the database to client memory, SMRC performs pointer *swizzling* (due to the relocation of the object in the client memory). Swizzling is the conversion of persistent database pointers into main memory address pointers. Whereas schema mapper products are useful to provide an object-oriented view of existing relational data, SMRC provides persistence for new OO data that need to be stored in relational databases. In this sense, SMRC is complementary to schema mapper products like Persistence** (see the section on traditional approach and related work) that require substantial data transformation between the relational representation and C++ objects.

An alternative to using SMRC for making C++ objects persistent might be to use one of the abovementioned OODBMSs. OODBMSs provide many features that are not available in most relational databases, such as a rich object-oriented C++ data model, less impedance mismatch, fast navigational access, etc. However, OODBMSs offer these features at the cost of introducing their own server environment in addition to an existing RDBMS environment, and thus burden the user with managing multiple database servers. In fact, SMRC does not compete directly with OODBMSs. OODBMSs target different market segments and work best for those users who have mostly OO applications and need only persistence and simple query facilities for their OO data. OODBMSs offer smaller, faster servers for OO data, and can handle varying granularities of data with ease. In contrast, SMRC supports a tight C++ language binding as well as clustering and pointer swizzling for fast pointer browsing, seemingly as part of the existing RDBMS that users already depend on. SMRC is designed to allow users of an RDBMS to incorporate OO data into their existing relational tables and applications. Using SMRC is similar to using an OODBMS, but SMRC uses a two-level store model rather than the traditional single-level store of most OODBMSs.

In this paper, we describe the design and implementation of SMRC. We first elaborate on the problems of storing C++ objects in relational databases, point out the shortcomings of related approaches, and briefly introduce the SMRC approach. Next we describe the SMRC concepts and the application programming interfaces. Then we discuss various implementation issues and present some performance numbers. Finally, we provide a summary and give a brief outlook on future work.

## Class definitions and relational schemata

Many different approaches are proposed to map class definitions into a relational schema. In this section, we first show why these approaches are inadequate, and then we present the approach pursued by SMRC.

**Traditional approach and related work.** Data to be stored in a relational database system must first be normalized, following the well-known relational normalization rules.[13] Normalization typically results in a corresponding table per object type, with a corresponding column per data member.[14] Most existing database applications are designed in this way. However, this approach poses some problems when applied to class definitions that involve additional language concepts like encapsulation, inheritance, and substitutability. Nevertheless, some schema-mapper products available in the marketplace support a (semi-)automatic mapping of class definitions to relational schemata, e.g., Persistence[15] and Subtleware**.[16] In these products class definitions are mapped to tables, exposing data members (even the private ones), and nested data structures are spread across tables. Class hierarchies are mapped either in a collection of tables or a "super" table. In the first case, a root table contains all basic data members and, additionally, a discriminant column to decide on the subtype of the objects. The tables for leaf classes carry only the additional attributes. In the super table case, the class hierarchy is completely flattened into one super table. The records of this table contain null values in the columns of data members that are not applicable. In both approaches, the C++ main memory pointers are replaced by primary key and foreign key relationships, and system-specific constructors, destructors, and access methods inherited from a persistent root class are included in the C++ class definitions. The access methods usually contain the hidden SQL code to communicate with the underlying database system and to destruct and store, and retrieve and construct the objects. To be fair, it is important to point out that products that map classes to tables are typically designed to promote OO views of legacy relational data. The data originate in the relational database, and these products offer an OO view of the data. They are not concerned with destroying the structure of an object by mapping it into a table because they are instead creating objects from tables. In contrast, SMRC concentrates on new OO data that were created in an OO application and then are stored in an RDBMS.

Other kinds of products provide portable C++ class library interfaces to relational databases. For instance, the class library from Rogue Wave Software[17] contains classes like column, row, cursor, table, etc., to communicate with an RDBMS. These class libraries are mostly useful for an object-oriented access to existing relational data in databases (again, where the data originate in the RDBMS), but are not at all able to deal with the previously raised issues. They offer method application programming interfaces (APIs) for their own generic storage libraries and, for portability reasons, link in appropriate SQL run-time libraries provided by the RDBMS vendors. Market acceptance as well as performance are critical issues for these approaches.

Persistence frameworks like PSOM (SOM [System Object Model] persistent framework),[18,19] OMG (Object Management Group) Persistence Service,[20] and Taligent frameworks provide an object-oriented infrastructure to make objects persistent. Framework classes can be subclassed by the user in order to customize how and where objects should be stored. The application program must use the infrastructure and API of the framework to achieve persistence.

For the following varying reasons, all the above-described approaches conflict with the SMRC goals stated earlier in the introduction to this paper:

- Object nature is destroyed. The proposed mapping approaches destroy the object nature, as they flatten the data members of objects into columns of records. Each method application requires translating and even reassembling the object into the original representation before methods can be applied on it. This approach degrades the performance of search queries in decision support systems, which apply predicates to a potentially large number (millions) of records and, thus, multiply the cost of object reassembling.
- Class libraries are useless. Database records are not objects. Since the class methods are only applicable on objects, the acquired class libraries are

useless for the database system without object recreation.

- Encapsulation is broken. Problems also arise with the loss of C++ semantics. Private data members in class definitions should be accessible only via methods or "friends," and therefore should not be exposed in columns of relational tables. Although one could hide private data members by restricting access to base tables and allowing access only through views that omit the private data members, this would be different from the original semantics of private data members.
- Proprietary query languages and access methods are used. The above (briefly introduced) approaches use their own query languages and infrastructures, making it difficult to develop portable database applications.

Some of the above problems may remain even if OO features are added to existing RDBMSs. For instance, systems such as Polyglot[21] and others[22] introduce their own type systems with their own notions of encapsulation, inheritance, and substitutability. By introducing their own type systems they remain incompatible with the external type system of an OO programming language and thus do not address the problems that SMRC solves.

**The SMRC approach.** The previous section outlined certain language concepts and discussed shortcomings of existing approaches. The shortcomings exist mostly because the described approaches introduce their own query language and try to map the type system of the programming language to the RDBMS type system. In this section we give an overview of the SMRC approach and list the major concepts required to implement the approach.

*Object preserving.* In SMRC, objects are stored in the database as they are created in the C++ type system; therefore, the nature of the object is preserved. No type transformation of the object representation is required upon object retrieval, and class library methods can be applied almost immediately on the objects without a significant loss of performance, since it is not necessary to recreate the original object representation. SMRC takes care of the C++ language peculiarities in implementing encapsulation, inheritance, and substitutability. Using SMRC, the database system does not have to adopt the specific C++ semantics and can retain its language independence. Objects are stored via SQL in UDT columns of binary built-in database types; thus, the approach does not introduce yet another query language. The object methods are applicable on the client side in the application (as regular methods) as well as on the server side (as UDFs[23]).

*SQL.* The structured query language, SQL, is a universal basis for data storage and it appears to be more attractive for independent software development than start-up query languages. SQL is already used by existing database applications.

*Object containers.* The fields of a relational table are used as containers to store objects. SMRC employs two mapping schemes to store C++ objects in containers: the abstract data type (ADT) mapping and the binary large object (BLOB) mapping,[24,25] depending on how the containers are populated with objects. The ADT mapping stores a single object of a class or class hierarchy in a container, whereas the BLOB mapping clusters many objects of different class definitions in a container.

*Pointer swizzling.* As the objects are stored in native main memory format, pointers in the objects need to be swizzled (converted to main memory address pointers) by SMRC when the objects are retrieved from the database and relocated in main memory. SMRC type-tags the objects (associates an object with its data type) before storage, which allows it to locate the pointers within the objects upon retrieval. SMRC supports two types of pointers depending on the location of the target object of the pointer in the database:

- Internal pointer—The referenced object is stored within the same container as the current object. This model is used mostly in the BLOB mapping.
- External pointer—The referenced object is stored in a different container from the current object.

Internal pointers are implemented as normal C++ pointers, declared in the class definitions. External pointers, which have additional semantics with regard to object faulting, require more structure than just C++ pointers and are treated separately. We implemented two different approaches for external pointers, one of which is compatible with pre-existing class libraries.

The use of BLOB fields as general-purpose object containers is very powerful from an application developer's point of view, as no data structure mappings are required; it is powerful as well from a database system point of view, as it does not have to deal with the inner details of an external type system. UDFs can

be employed to interpret the contents of objects and retrieve certain data members of objects only. The synergy between SMRC, BLOBs, and UDFs provides the additional functionality for relational databases to store C++ objects, coresident with existing relational or other kinds of nontraditional data. However, the BLOB container approach also has as a consequence, that certain database operations cannot be performed directly. Indexes, join operations, or objects as part of primary keys are not possible, as the BLOB type cannot be assigned or compared to any other type. These are well-known problems that also exist in other areas of data management, e.g., the storage of OLE** objects in any kind of container, or text processing documents or spread sheets in files. In any case, only the original application is capable of looking into the contents of these containers or files. However, certain parts of the container that are accessed frequently or need to be indexed, can always be stored separately in addition to the container. Technology like Notes/FX** (Field eXchange) is available to automatically synchronize the values in the container and the separately stored values.[26] Notes/FX uses OLE embedded objects to provide bidirectional data exchange between fields in a Notes document and objects created by FX-enabled OLE server programs.

## SMRC concept and APIs

In this section, we first sketch various SMRC sample applications and give a first impression on how to use SMRC in combination with an RDBMS. Then, we describe the prerequisites to make objects persistent, elaborate on the application programming interfaces (APIs) for the ADT and BLOB mapping, and show some examples of using external pointers.

**Developing applications in a SMRC/RDBMS environment.** One important feature of a relational database system is that users can extend the database by adding columns to existing (and populated) tables. In the case of SMRC, the application exploits this feature by using the additional columns as data containers to store C++ objects. Using the ADT mapping, one C++ object of a class or class hierarchy exists only in one data container, whereas in the BLOB mapping, many objects of different class definitions map into the same data container. The selection of one of the proposed mapping approaches depends on the specifics of an application. Figure 1 shows two samples for the ADT and BLOB mapping, explained in more detail in the next few paragraphs. The top of the figure describes the C++ classes, the bottom of the figure describes the relational tables and the middle part shows C++ objects to be stored in the tables.

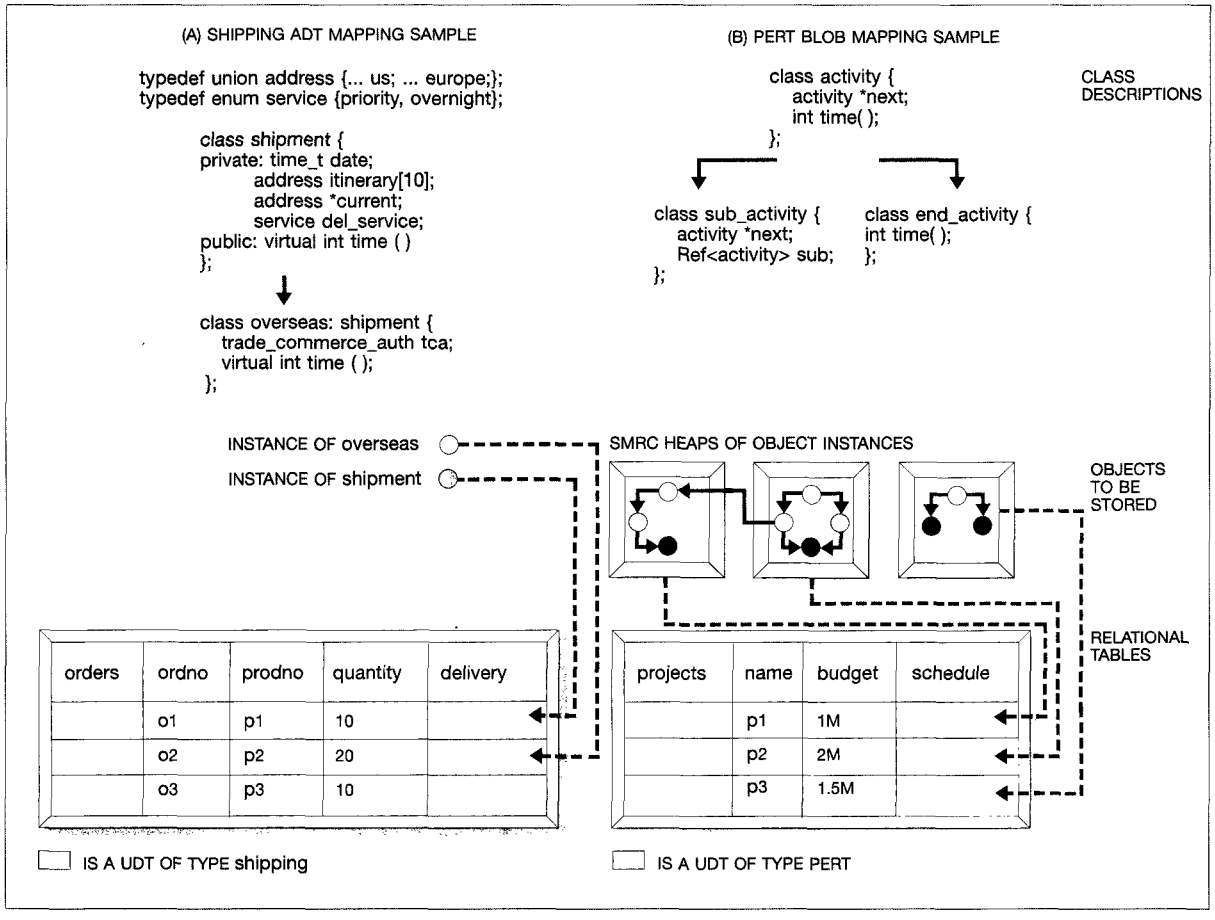
The ADT mapping applies to applications where single C++ objects may act as additional descriptive attributes to database entities. Figure 1A shows a table orders with some typical columns like ordno (order number), prodno (product number), and quantity, and an additional column delivery that contains the C++ objects describing the delivery of an order. The data type of the delivery column is a UDT called shipping, which we will explain later. The C++ objects belong to the shipping class hierarchy consisting of a super class shipment for usual deliveries and a specialization class overseas in the case of customs being involved in the delivery. The class library provides the required method implementations, e.g., a method time() evaluates the itinerary of a delivery and estimates the delivery time. The dashed lines in Figure 1A sketch the mapping of single objects of the class hierarchy in the delivery column of the orders table.

One goal of storing the C++ objects along with the relational data is to perform queries that make use of both the relational and the object-oriented data in the database. For this purpose, the time() method of the C++ class library is registered as a UDF in the database system and, thereafter, it can be used in SQL queries like the example that follows. Note that the correct virtual function must be invoked for each select-item of column delivery according to the C++ type of the delivery argument, which may change from record to record, due to subclassing. (Although it is not shown in the sample application, SMRC supports multiple inheritance.)

```
select ordno, prodno
from orders
where time(delivery) > 5 and quantity = 10;
```

The BLOB mapping applies in applications where a heterogeneous set of interconnected C++ objects constitutes an additional attribute of a database entity. Typical BLOB mapping applications come from the areas of project management, network management, workflow management, and complex geographical information system (GIS) applications. Figure 1B sketches a project management sample application. It shows a table projects with columns name and budget, and a BLOB column schedule to store heaps (collections of memory) of C++ objects representing PERT charts of the projects. (PERT, or

**Figure 1 SMRC mapping samples**




Project Evaluation and Review Technique, charts illustrate critical paths for completion of project tasks.) A class hierarchy includes a super class activity and two subclasses for sub and end activities. The figure shows C++ objects of three PERT charts allocated in SMRC heaps, which are mapped into the schedule column of the projects table. The sample also shows the usefulness of external pointers, as one of the objects in the PERT chart for project p2 refers to project p1 as a subproject. The implementation of a "real" BLOB mapping application (and the related experiences) using SMRC is described in a paper referenced earlier.[10]

SMRC supports additional functionality for *external pointers* (as opposed to internal pointers). An external pointer contains all the information required to retrieve the referenced object from the database. SMRC is able to *fault in* the referenced object from the database automatically when the external pointer is *dereferenced.* (When an object is referenced, but is not in main memory, a *fault* condition occurs resulting in retrieving the object from the database. The terminology used for this event is *fault in. Dereferencing* a pointer results in the value at the location that the pointer points to.) In the case of the ADT mapping, only the one referenced object is *faulted in*, whereas in the case of the BLOB mapping, the whole heap containing the referenced object is installed in main memory.

The application program determines how C++ objects should be mapped to database containers. The application program creates objects either in the pro-

gram default heap space (for the ADT mapping) or in SMRC heaps (to cluster objects for the BLOB mapping). Object creation happens either via the SMRC overloaded new operator or the standard C++ new operator. When the standard C++ new operator is used with the BLOB mapping, an additional SMRC API call is required to type-tag the created objects and eventually copy the objects into a SMRC heap.

In the ADT mapping, internal pointers are hidden pointers, introduced by the C++ compiler to implement inheritance and substitutability, as well as pointers referring to a data member within the same object. In the BLOB mapping, additionally, internal pointers can refer to objects allocated within the same SMRC heap. External pointers are supported for the ADT and BLOB mapping, and they have to be assigned by a special SMRC API call.

The decision whether to use ADT or BLOB mapping depends on the access patterns to the objects used by the application. The BLOB mapping offers two major advantages over the ADT mapping. First, the application programmer has the ability to cluster many objects of different class definitions in the same container, in the event that they are logically related to each other and are often requested at the same time. Second, many related objects can be retrieved by one database operation, as opposed to the ADT mapping that retrieves one object at a time. On the other hand, retrieving one object at a time might be more useful for applications that require a fine-grained access to data.

With SMRC, C++ applications use standard SQL (query language) to store objects (or SMRC heaps of objects) in object containers of the database (i.e., table fields). Objects are stored in binary format of the C++ type system without any data conversions. The table columns for the object containers are defined as UDT types of some built-in binary datatype of the database system.

**The SMRC persistence schema.** A SMRC persistence schema is a collection of application type descriptions created by the SMRC schema compiler. The schema essentially describes the layout of all of the persistent C++ objects for the application, which is needed for memory management and pointer swizzling. The schema includes structural information (in particular, size and pointer offset information) that contains the type information of embedded structures, unions, and dynamic arrays. In addition, the type information contains the offsets of the hidden pointers, i.e., offsets of virtual function table (vtable) and virtual base (vbase) pointers. To create a persistence schema, the SMRC schema compiler takes as input an application schema source file that includes the header files containing the C++ class definitions and SMRC flags that mark which classes should be made persistent. The schema compiler produces a named persistence schema that is stored in the schema database. A persistence schema is compiler-specific due to the compiler-specific allocation of the hidden pointers within the objects, but not machine-dependent, as the persistence schema uses only symbolic information.[27,28] The current SMRC implementation uses IBM's C Set++* compiler.[29]

The application schema source file that follows shows the flagging of the shipping application in Figure 1A. The purpose of the file essentially is to include the header files with the C++ class definitions and selectively flag those classes (within a dummy function just for compilation purposes) that might have persistent objects. Similar approaches to capture C++ class information are pursued by OODBMSs. The use of additional flagging macros in an application schema source file provides a way for users to plug in user-provided functions for unions, or repeating functions for dynamic arrays. The overall schema compilation process is described in Reference 10.

```
#include "smrc_macros.h"
#include "shipping.h"
void dummy () {
  SMRC_TYPE (shipment);
  SMRC_TYPE (overseas);
};
```

**Application programming interface for ADT mapping.** In this section we describe the SMRC ADT mapping API, and employ a more comprehensive version of the previously introduced shipping application to demonstrate the use of the API.

SMRC tracks type and relocation information for pointer swizzling purposes. The type information provides the pointer offsets to achieve addressability of the pointer data members in the objects. The relocation information provides the basics to calculate the load differences of the objects required for pointer swizzling. Since the database system does not know about C++ class definitions (and C++ does not support run-time type information), SMRC attaches type tags to the objects before they are stored in the database system. After an object is retrieved

from the database, all the internal pointers (the hidden pointers in the ADT mapping) within the objects are swizzled before object usage. The external pointers are swizzled transparently at dereference time.

The following SMRC API calls provide the required functionality for type information and pointer swizzling:

- *Type tag C++ object*: smrc_tag (objptr, type_name, schema_name, hv). Before an object is stored, smrc_tag() is called to copy the object referenced by objptr into the SQL host variable hv, deswizzle the (hidden) pointers, and type-tag the object copy. Type_name and persistence schema_name are used by SMRC to create a unique type tag. The tag call is required for newly created objects (created by the standard C++ new operator) as well as updated retrieved objects, as SMRC performs deswizzling of pointers in this call. After tagging an object, the object in the SQL host variable is stored in the database via an SQL insert or update statement.

- *Swizzle pointers*: swizzle (hvptr). Objects are retrieved from the database into an SQL host variable hvptr via select statements. The swizzle() call takes as an input a pointer to the retrieved (unswizzled) object in the host variable, swizzles the object, and returns a pointer to the swizzled C++ object.

- *Get the type of an object*: smrc_object_type (hvptr). When object instances of a class hierarchy are stored in a column, it is useful to be able to dynamically identify the type of a particular object in the column. The smrc_object_type() call returns a character string identifying the type of the object currently retrieved into the SQL host variable hvptr.

The following steps show the use of the ADT mapping API for the shipping sample application in Figure 1A. We start with the database description and then insert and retrieve objects to and from the table.

*Create table/add additional column.* The objects of the C++ class hierarchy in Figure 1A are stored in a table column delivery based on a distinct type. A distinct type essentially is a renamed built-in database type.[8] The size of the distinct type is the size of the largest class in the class hierarchy (plus 4 bytes for the type tag). The following statements can be performed in dynamic SQL in order to determine the 830 varchar size (size of class overseas + 4) and define the table:

```
create distinct type shipping as varchar (830)
   for bit data with comparisons;
create table orders (ordno int, prodno int, . . . ,
  delivery shipping);
```

If orders is an already existing (populated) table, column delivery is simply added to it by modifying the table.

*Register class methods as UDFs.* Class methods to be used within SQL must be registered with the RDBMS. As the class methods cannot directly be registered as UDFs—they do not follow the SQL UDF calling conventions—SMRC generates external UDF gateway functions for each class method to be used within SQL statements. The signature of a UDF gateway function has the appropriate UDT as an input type and the result type of the class method as an output type. The implementation of the UDF gateway function obeys the SQL UDF calling conventions. It first swizzles the input, and then calls the original class method.

```
create function time (shipping)
  returns integer
  language c
  external name '/u/reinwald/udf_lib!time';
```

*Insert objects into the database.* Objects are created via the standard C++ new operator and inserted into the database via the standard SQL insert statement.[30] In the sample smrc_tag call, "overseas" is the type name within the "shipping" application schema. An object is created, tagged in an SQL host variable hv of an appropriate size and inserted into a table.

```
struct {unsigned short len; char data[830];} hv;
overseas *delptr = new overseas();
  . . .
smrc_tag (delptr,'overseas','shipping',&hv);
exec sql insert into
  orders (ordno, prodno, quantity, delivery)
  values (10, 20, 10, :hv);
```

*Retrieve objects from the database.* Objects are retrieved using the standard SQL select statement. We do not impose any additional restrictions on such statements. These statements can be dynamic, or static for better performance, and can flow across any supported API, such as DRDA,[31] ODBC,[32] etc. They can also be interactive or embedded in applications.

The SMRC swizzle call may be used from within the client application after retrieving the C++ object into an SQL host variable, or from within the UDF gateway implementation. The following two examples demonstrate both cases: in the first example, a delivery object is retrieved into an SQL host variable and swizzled on the client side. As the object can be either of C++ type shipment or overseas, proper type casting needs to be done. The second example shows the use of the time UDF gateway function; thus, the swizzle call is hidden in this UDF. The UDF runs on the server side.

Case 1: Client side swizzling.

```
select ordno, delivery
into :ordno, :del_obj
from orders
where quantity > 10;
if (!strcmp(smrc_object_type(&del_obj),
            "shipment")) {
  dp = (shipment *) swizzle(&del_obj);
} else {
  dp = (overseas *) swizzle(&del_obj);
};
```

Case 2: Server side swizzling.

```
select ordno, prodno, time(delivery)
into :ordno, :prodno, :time_delivery
from orders
where quantity > 10;
```

**Application programming interface for BLOB mapping.** The API for the BLOB mapping essentially consists of the methods of the SMRC heap class. The SMRC heap class provides the necessary methods to both manage objects in memory heaps and swizzle the pointers in the objects after retrieval from disk. Given that the objects within a SMRC heap are stored and retrieved in one database operation, it is reasonable to consider a SMRC heap as the unit of persistence as well as the swizzle unit.

A SMRC heap is associated with a persistence schema at heap creation time. Many different heaps with different schemata can exist in an application simultaneously. An application can allocate objects directly in the SMRC heap, via the SMRC overloaded new operator, or it can alternatively create C++ objects in its own heap and then later call the SMRC "deep object copy" routine, which copies a complex network of referenced objects into a SMRC heap (see Reference 10 for details). Each SMRC heap has a root object (or, potentially multiple root objects) that gives the application an entry point to the network of objects within a heap. The entire heap of objects is stored in binary format in a relational table.

Upon retrieval of a heap in main memory, all the internal pointers within a heap are swizzled at one time ("heap-at-a-time" swizzle approach), after which, a user can navigate through the objects at main memory speed by dereferencing the C++ pointers. External pointers are swizzled lazily at dereference time, when a heap containing a referenced object gets *faulted in* by SMRC. Retrieving only a subset of the objects in a heap is not supported, although the user might have a UDF operating on the heap, which returns only a value, or a table function which returns a set of tuples. Currently, table functions are currently not supported by DB2.

The following API calls are listed in the order of typical usage in an application:

- *Create heaps*—A SMRC heap is created with an associated persistence schema. Objects of class definitions within this schema can be allocated in the created heap. The size of a heap grows dynamically.

  ```
  smrc_heap *hp = new smrc_heap('PERT');
  ```

- *Create and delete objects*—Objects are allocated in a heap via a SMRC overloaded new operator and removed from the heap via a cancel method.

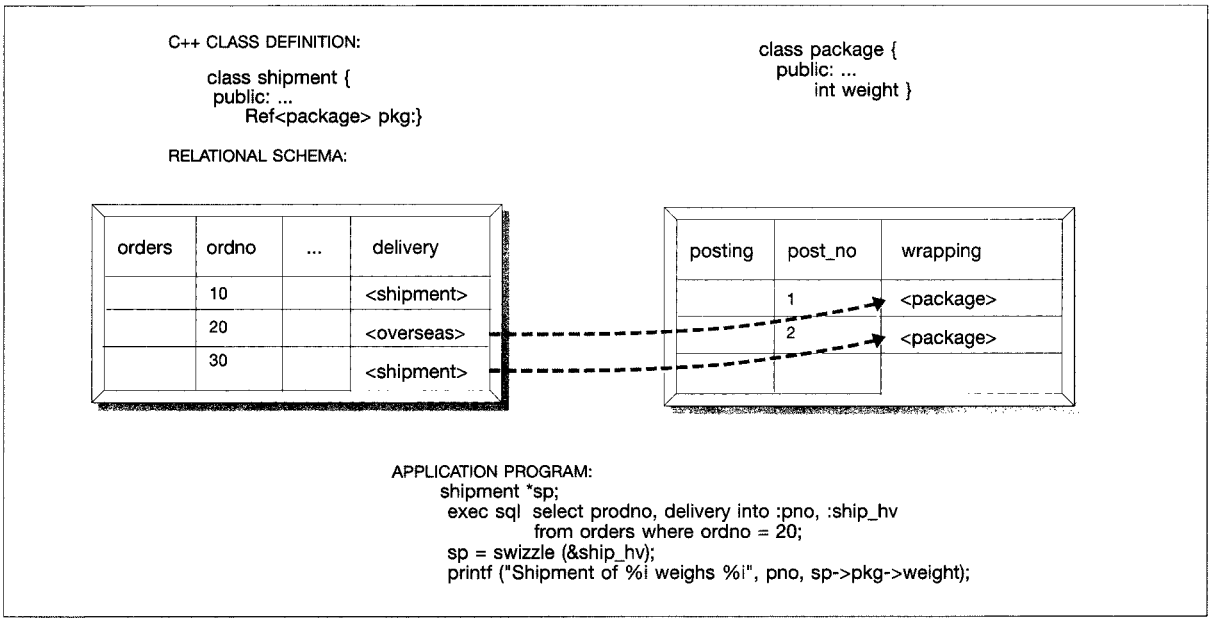
  ```
  // create new object in heap
  obj = new (hp, 'activity') activity;
  // remove existing object
  hp→cancel (obj);
  ```

- *Root objects*—Root objects provide entry points to a heap. They can be set (set_root) and retrieved (get_root) via heap methods.

  ```
  hp→set_root (objptr);
  objptr = (activity *) hp→get_root();
  ```

- *Store heaps in database*—SMRC heaps comprise multiple memory segments to allow dynamic growth. Thus, before a SMRC heap can be stored as a value in the database, it must first be "packed" into a contiguous memory segment. However, the SMRC heap management avoids this copy step if the heap is not segmented. As shown in the sam-

**Figure 2 External pointer sample application**




ple below, SMRC sets up the SQL host variable (hv) to store the heap into the database (the 500k size in the declaration of the host variable is required by the RDBMS for range checking).

```
sql type is blob(500k) *hv;
hv = hp→pack();  // pack heap hp and setup hv
insert into projects (schedule) values (:*hv);
```

- *Retrieve heaps from database*—SMRC heaps are retrieved from the database into an SQL host variable.

```
sql type is blob(500k) hv;
select schedule into :hv
from projects;
```

- *Swizzle heaps*—A retrieved heap in a host variable (hv) is swizzled and assigned to a SMRC heap variable. After this, all the SMRC heap methods can be applied (e.g., get the entry point of the heap with hp→get_root()).

```
smrc_heap *hp;
hp = swizzle (&hv);
objptr = (activity *) hp→get_root();
// Now the application can access objects in the heap
// via (pure) C++ pointer browsing.
```

**Working with external pointers and object caching.** SMRC supports external pointers,[33,34] which extend the scope of pointers and refer to objects stored in other fields of the same column, other columns in the same table, and even columns in other tables. Figure 2 shows an extension of the shipping ADT sample application. Class shipment contains an external pointer pkg to class package. The shipment objects are stored in column delivery of table orders and the package objects in column wrapping of table posting. The sample application code first shows retrieving and swizzling of a shipment object from the orders table. From an application programmer's point of view, the external pointer pkg behaves exactly like an internal pointer. But in a normal C++ application, the dereferencing of the pkg pointer in the shipment object would cause a segmentation violation, as the appropriate package object might not be resident in memory. However, as the pkg pointer is declared as a SMRC external pointer, SMRC is able to catch this violation, automatically query the database for the referenced package object, swizzle the retrieved object, and install it in main memory so that the object can be referenced by C++.

The assignment of external pointers is different from internal pointers, as additional information is required, such as the table and column in which an application stored the referenced object. This information is provided in an assignment method. At assignment time, SMRC creates object identifications (OIDs) and stores them as external pointers, which are used to *fault in* the referenced objects.

SMRC supports two different approaches to "declare" external pointers, and the application programmer can opt between the two choices as appropriate.

- *Template-based approach*—An external pointer is declared within the C++ class definition via a SMRC-provided Ref template. This approach is like the ODMG-93 C++ language binding.[35]

```
class shipment {
  ...
  Ref⟨package⟩ pkg;
};
```

- *Flagging-based approach*—An external pointer is flagged via a SMRC macro in the application schema source file.

```
void dummy () {
  ...
  SMRC_Ref(shipment, pkg);
};
```

In the template-based approach, SMRC uses an overloaded dereference operator that checks object residency at dereference time and queries the database in case of an object fault. The flagging-based approach uses the ability of the paging hardware to trap access violations in order to catch object faults at dereference time. For the application programmer, the choice between the template-based or the flagging-based approach depends on whether the class definitions can be modified to use the SMRC Ref template and to have a portable application, or to not modify the class definitions but depend on page protection in the hardware.

Whether an object is *faulted in* via the overloaded dereference operator or via page protection, SMRC allocates the *faulted in* objects in an object cache. From an application programmer's point of view, there is no distinction whether an object exists in the application address space or the object cache, which is part of the application address space.

Nevertheless, the application programmer must be aware of the object cache to exploit its additional functionality. The object cache offers the following API (we introduce it without the syntax):

- *Flush cache*—All of the objects in the object cache are written back into the database and removed from the cache. This operation is useful at the end of application execution.
- *Save cache*—All of the objects in the object cache are written back to the database, but still exist in the cache. This operation is useful for saving object changes in the database while continuing the application.
- *Register objects in cache*—In the event that an object is retrieved separately (manually) by the application via SQL, it can later on be registered in the object cache.
- *Remove objects from cache*—Objects can be explicitly removed from the object cache. One might use this to avoid having modifications stored back to the database during flushing or saving cache.

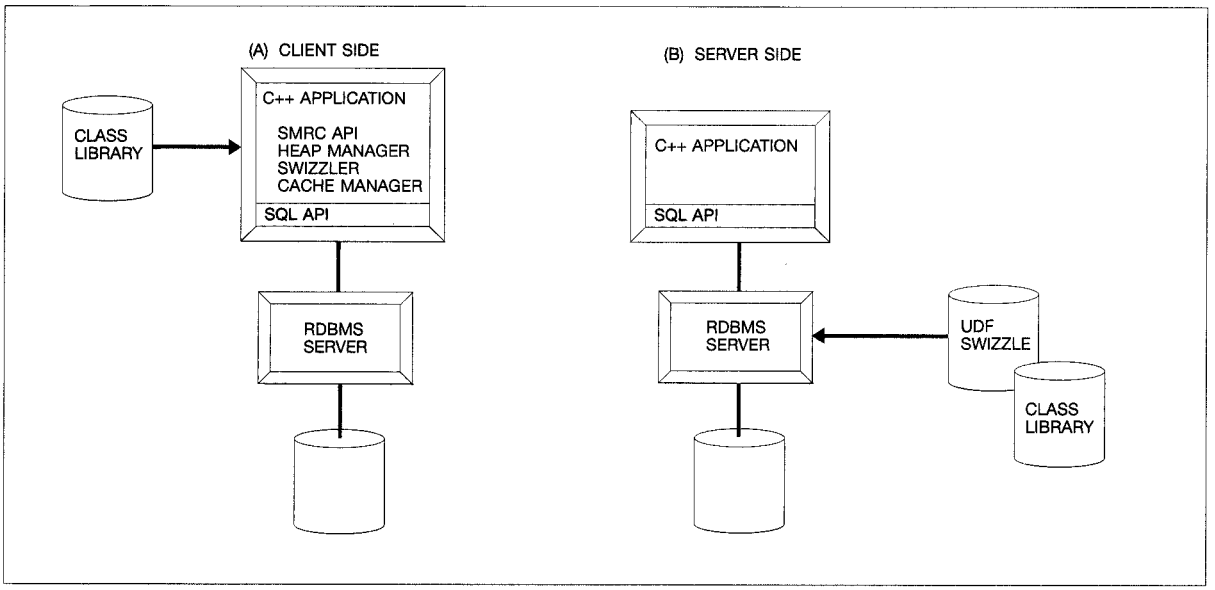
## Implementation

In this section, some of the specifics of implementing SMRC are addressed. We start with an architectural overview and briefly introduce the SMRC heap manager. The main part of the section is concerned with pointer swizzling in ADT and BLOB mapping as well as implementing external pointers.

**Implementation overview.** SMRC runs under the control of an RDBMS server and uses the SQL query language. This makes it relatively easy to extend existing relational database applications to use SMRC for additional storage of C++ objects and to have the stored objects be part of an integrated client/server database solution. Additionally, SMRC benefits from all the industrial-strength RDBMS features with regard to concurrency control, recovery, etc., of the underlying database system.

Figure 3 describes the environment one would use to develop an application with SMRC and an RDBMS, and shows the road map for this implementation section. SMRC can run on the client side as well as on the server side. Figure 3A shows the SQL API used by the C++ application as well as the SMRC API. The SMRC schema compiler (not shown in the figure) provides the required type information for type-tagging and pointer swizzling. A SMRC heap manager provides the object clustering functionality for the BLOB mapping. The cache manager *faults in* and allocates

**Figure 3 SMRC overview**




objects referenced via external pointers. At the server side (Figure 3B), C++ objects can participate in SQL queries by registering the class methods as UDFs. Since the UDFs are executed on the server side, SMRC performs pointer swizzling before the methods are applied.

**SMRC heap manager.** The SMRC heap manager is the key component for the BLOB mapping. It supports the functionality of a full-fledged heap manager on the client side, including main memory management of all the objects that should be stored within the same field of a relational table. A SMRC heap is segment-oriented and grows dynamically in size.

SMRC maintains two auxiliary data structures for the management of the objects within a heap: a type table and an object table for each type. The type table refers to the complete type description in the schema database and thus provides the heap manager with the required object layout information. The type table is built at heap creation time and is related to the persistence schema specified at heap creation time. The object table for each type is updated during each object allocation or deletion in a heap. The object tables grow dynamically. The entries in the object table refer to the objects within the heap. Type table and object tables are persistent, along with the objects in a heap. They provide addressability of each object and pointer within the objects in a heap, which is required for pointer swizzling. Thus, a heap is completely self-contained; it can be shipped in client/server environments and interpreted at each destination.

**Pointer swizzling.** When objects are retrieved from disk and reloaded into main memory, all main memory pointers within the objects must be swizzled due to object relocation. SMRC supports three different approaches for pointer swizzling—all three approaches are implemented to support either the ADT mapping, the BLOB mapping, or external pointers.

- *Deswizzle pointers*—All the pointers within an object are deswizzled, i.e., the current object address is subtracted from all the pointer addresses before an object is saved on disk, thereby making them offsets to the beginning of the object. After object retrieval, the pointers are swizzled by adding the new object address to all the pointer addresses.
- *Save previous object load address*—The previous object load address is saved on disk along with the object. After object retrieval from disk, the pointers are swizzled by the difference between the previous and the new object load address.

- *Object identifications (OIDs)*—Main memory pointers are replaced by persistent OIDs, which are independent of the current location of a referenced object in main memory. A referenced object can always be identified with an OID, either in main memory or on disk.

For the *ADT mapping*, we have implemented the deswizzle approach, as it is the more efficient way in terms of memory space (the old object load address does not have to be saved along with the objects on disk). Deswizzling happens when an object is type-tagged with the SMRC type-tag call introduced in the ADT mapping API. The type tag is used after object retrieval to swizzle the pointers according to the new object location, i.e., we add the address of the new object location to the offsets in the deswizzled object. For the ADT mapping, swizzling is performed one object at a time.

For the *BLOB mapping*, we basically apply the second approach. However, it is not necessary to save the previous load address for all the objects in a heap, as it is sufficient to save just the load address of the entire heap itself. The relative address of an object to the load address of a heap remains the same, as a heap is relocated as a whole. Saving the previous load address of a heap is the more efficient swizzle approach for the BLOB mapping than traversing and deswizzling all the objects in a heap. This approach is similar to the memory-mapped segments in ObjectStore,[36] of course without doing memory-mapping. In ObjectStore, the pointers in the pages of a segment are swizzled on the basis of the relocation of the segment. In the case of SMRC, a whole heap of objects is loaded by the application into main memory, and SMRC swizzles the internal pointers of all the objects in a heap with the heap load address as a reference point. The type table and object tables in a heap are scanned to gain addressability of the objects, and the associated type information provides the offset information of the pointers within the objects. Thus, the SMRC swizzler is able to directly address and swizzle all the pointers in a heap without any search or navigational overhead.

For *external pointers*, SMRC performs pointer swizzling based on OIDs. If an object is referenced via an external pointer and the referenced object is not yet in main memory, the referenced object is *faulted in* and the location of the object is used as a main memory pointer (swizzled pointer). Details on swizzling external pointers are described later.

The previously described pointer swizzling approaches are used for user-defined pointers. The swizzling of virtual function table (vtable) pointers (the same approach is applied for function pointers) is described in the next section. Reference 10 elaborates on incorporation of user-provided functions to swizzle unions and dynamic arrays.

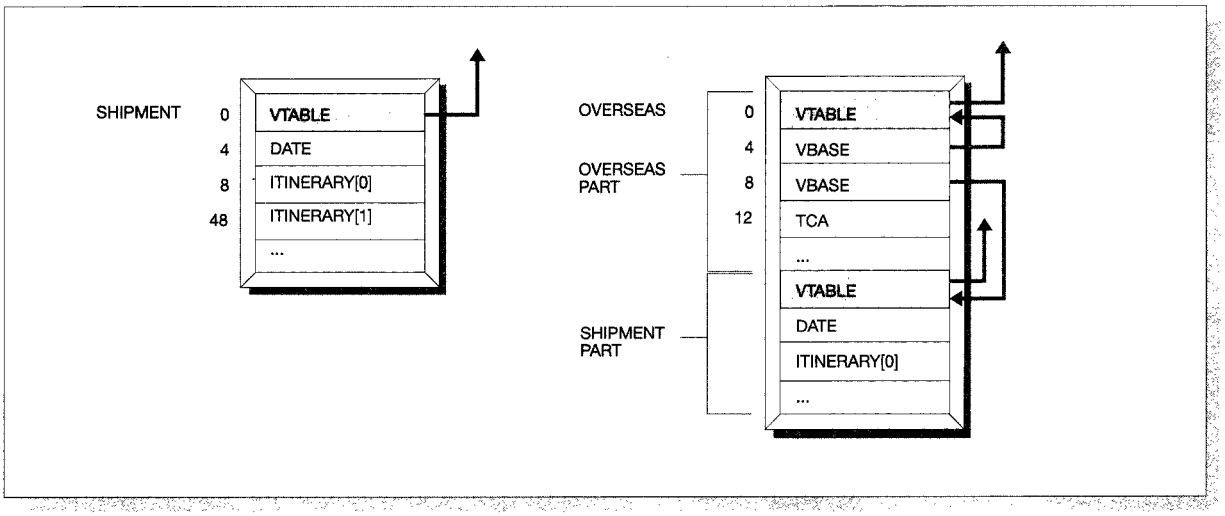
**Hidden pointers.** C++ compilers implement dynamic dispatching and substitutability[34] via two types of "hidden" pointers: vtable pointers and virtual base offset pointers. Figure 4 shows the object layout of the shipping class hierarchy introduced in Figure 1A and highlights the compiler-introduced vtable and virtual base offset pointers. The hidden pointers are introduced by the C++ compiler for class definitions that contain virtual functions or virtual base classes. Just like any normal pointer, these hidden pointers need to be swizzled when the object is relocated in main memory. The location of the hidden pointers within an object depends on the specific C++ compiler. At this point, the SMRC schema compiler is compiler-dependent (IBM's C Set++), as it relies on the C++ compiler to specify the offsets of the hidden pointers.

Virtual base offset pointers refer within an object and can be swizzled using conventional methods. However, the vtable pointers (pointers to the table that implements dynamic dispatching of virtual functions) depend on the allocation of the current instance of the vtable in an application and cannot be swizzled on the basis of object relocation. The best solution would be to let the C++ run-time system swizzle the hidden pointers, since it knows exactly how to set these pointers. Unfortunately, C++ sets the hidden pointers only when an object is created with the new operator as part of the constructor execution and does not export a callable "swizzle" function. A thorough discussion of the whole hidden pointers issue can be found in Reference 37.

SMRC swizzles the vtable pointers by allocating dummy objects with a correct vtable pointer (one that was created in the current instance of the application) and "steals" the correct value of the vtable pointer from this object. The SMRC schema compiler provides the location of the vtable pointer in an object. This approach is similar to the ObjectStore approach[38] that maintains a hash table, mapping type names into vtable addresses.

The table is created during the ObjectStore internal schema generation time at application startup time.

**Figure 4 Main memory layout of C++ objects**




Ontos' approach (Vbase), making the vtables persistent as well, does not seem to be appropriate.[39] A constructor approach is exploited by O++ (Ode),[40] that introduces a "faked" new operator (does not allocate memory). The new operator triggers the execution of a constructor that fixes all the hidden pointers. As no data members should be initialized with the constructor, all the default class constructors of an application have to be rewritten in order to distinguish whether they are used for pointer fixing or usual object initialization. This approach is not useful for SMRC, as it would require a recompilation of parts of a class library, although the constructor rewriting can be triggered automatically by a C++ precompiler.

**Object cache and OIDs.** The object cache, similar to the SMRC heap (a superset, really), is part of the application address space and can grow dynamically in size. SMRC uses the object cache to manage automatically *faulted in* objects via external pointers. SMRC maintains an in-memory object table (hash table) with the object identifications (OIDs) of all the loaded objects in an object cache. An OID uniquely identifies an object in the database and thus can be used for the following two purposes:

1. *Object residency checks*—Before an object is *faulted in* automatically, SMRC must check whether or not the referenced object is already loaded in the object cache.
2. *Object faulting*—When an object must be *faulted in*, SMRC needs to be able to retrieve it from the database.

   SMRC launches an "under the cover" SQL statement to retrieve objects from the database:

   ```
   select ⟨object_column⟩
   from ⟨table⟩
   where ⟨predicate⟩
   ```

The SQL statement takes the OID as an input, which is kept along with the external pointer causing the object fault. To provide all the input for the select-statement, an OID contains the following information (20-byte structure):

```
OID = {table_id, column_id, row_id}
```

Table_id and column_id are created out of the database catalogs for the tables and columns in a database. Database catalog information is cached to quickly translate the table_ids and column_ids in OIDs to the corresponding table and column names for the setup of the SQL statements. A row_id is similar to a system-generated primary key, but is not reusable. It uniquely identifies a record within a table and contains physical information to speed up database access.[41] By having a row_id as part of the OID in an external pointer, the *faulting in* of referenced objects can be very fast.

**Swizzling external pointers.** SMRC swizzles external pointers in a "lazy" fashion, depending on whether the template-based approach or the flagging-based approach has been chosen to declare an external pointer. In the *template-based approach*, SMRC pursues "swizzling on discovery."[36,42] Unswizzled pointers in loaded objects are swizzled as soon as they are discovered, i.e., during assignment or pointer dereferencing. For this purpose, the Ref template implements overloaded assignment and dereference operators. The approach avoids having unswizzled pointers in local variables and unnecessary object loading. In the *flagging-based approach with page protection*, SMRC supports "swizzling at dereference time," as only pointer dereferencing can be trapped and not the assignment of an unswizzled pointer to a local variable. At object load time, SMRC swizzles an external pointer to a protected page and installs a signal handler to catch the segmentation violation at dereference time (page protection traps).[43,44]

Similar page protection approaches are implemented in ObjectStore,[38] Texas Persistent Store,[45] and QuickStore.[46] However the SMRC implementation differs in two important aspects from the above approaches:

- *OIDs of different objects can share protected pages for trapping purposes*—When SMRC loads an object with external pointers, it stores the object identifier of the target objects for the external pointer on a protected page (with other object identifiers). On a protection trap, the SMRC handler knows the object identifier on the protected page (based on its location on the page) and is therefore able to query the database (as previously explained). The retrieved object is not allocated on the protected page, but in the object cache, which is not page protected (see the next bullet on how to assign the address of the faulted-in object to the fault-causing pointer). By putting many different OID targets on a single protected page we avoid "the fan-out problem," where whole page frames would be allocated for each external pointer in memory.
- *Reverse reference lists* (RRLs)—SMRC uses RRLs to track all references to an object. An RRL is a list of back pointers to objects (actually to pointers within objects) that reference the same object, i.e., the same object identifier allocated in a protected page. Using RRLs, SMRC is able to (1) redirect the fault-causing pointer to the address of the *faulted in* object during protection trap handling, and (2) avoid additional page faults caused by other objects that refer to the same object. Consequently, irrelevant residency checks are avoided and performance is improved. Additionally the RRLs can be used for garbage collection (reallocating unused memory) in the object cache. Given their usefulness, we feel that the time and space overhead for maintaining RRLs is justified.

Comparing the two external pointer approaches, the page protection approach makes object faulting entirely transparent to the compiled code, as opposed to the overloaded dereference operator that requires source code modification to define external pointers. On the other hand, fielding a page protection trap from the operating system is an expensive operation. Studies by Hosking and Moss[47,48] show that software solutions can be more efficient. Detailed performance comparisons and a discussion of the trade-offs between software dereferencing and memory-mapped storage systems with page protection traps (E versus QuickStore) can be found in Reference 46. The "unduly large granularity of virtual memory pages"—as stated by Hosking and Moss[47]—is not a problem in SMRC, as the virtual memory primitives are only used for page protection traps and the protected pages can serve many different external pointers.

## Performance

We evaluated the performance of SMRC through experiments that were implemented on an IBM RISC System/6000* with 128 megabytes of main memory running AIX 3.2.5 and DB2 Version 2.1. Client applications and the database server run on the same machine. Here we present some of our experimental results for the ADT and BLOB mapping.

**ADT mapping performance.** With regard to space efficiency, SMRC requires only 4-byte storage overhead for the type tag of each object—a type tag is stored as part of an object. With regard to the performance of storing the objects, the type tag operation requires a memory copy (the *memcpy* routine) of the object to get the data into the SQL host variable, plus an address assignment operation for each deswizzled vbase pointer. If an application uses the SMRC overloaded new operator, copying of the object is not necessary, as SMRC directly allocates the object along with the required type tag. For the SQL insert operation of the host variable, SMRC relies on the performance of the applied database operation.

For object retrieval, there is the performance of the select statement and the swizzle operation itself. The

swizzle operation costs an address assignment for each vbase pointer as well as for each vtable pointer. The SMRC internal persistence schema with the class layout description is built at startup time. It is global information that is used by all swizzle operations in an application.

We have not yet run any commercial relational or *de facto* OO benchmarks, as none are specifically geared to measure the unique set of features in SMRC. Relational benchmarks do not exploit the SMRC technology, and OO benchmarks do not incorporate the unique SMRC functionality of having coexistence between relational and object-oriented data. However, to gain an understanding of our relative performance to OODBMSs, we are preparing to run the OO7 benchmark.[49]

In the meantime, we developed the following experiment. We compared the SMRC ADT mapping approach (which maps an object to a single column) to an approach that completely "flattens" the C++ class definitions and stores all the data members in additional table columns.[50] In both cases, however, we required that the language object be available so that it can be passed to the UDF (time) to compute the query predicate. In the SMRC case, the object can simply be retrieved and passed to the time method. In the flattened case, however, the object must be reassembled before it can be passed to the time method (this work of reassembling the object is done in the UDF before the method call).[51]

We populated the orders table from Figure 1A with 2000 C++ objects and executed a query that did a table scan and invoked the time method on the C++ objects. We made the query result empty, to factor out the client/server communication costs and thus focus on the overhead of running SMRC in the server.

Table 1 shows the performance of the two queries. In both cases—the SMRC approach and the class flattening approach—the same original C++ time method was executed as part of the UDF invocation. The experiment shows that SMRC is able to preserve the C++ object nature; C++ methods can be applied after object retrieval from the database and object relocation in main memory. SMRC also performs slightly better (approximately 7 percent in the experiment) in comparison to a class flattening approach. Before conducting this experiment, we thought that the SMRC approach might be faster than a normalized approach, mostly because of the overhead in restoring the objects from the normalized tables. However, this experiment compares the SMRC approach against an unnormalized table, and SMRC was still faster.

**Table 1 ADT mapping performance results**

| Approach | Query (scans 2000 tuples) | Elapsed Time |
|---|---|---|
| SMRC ADT mapping | select prodno, quantity from orders where time (delivery) = 0 | 2.66 sec. |
| Class "flattening" | select prodno, quantity from orders where time (. . . 32 parameters . . . ) = 0 | 2.89 sec. |

**Table 2 BLOB mapping performance results**

| | | |
|---|---|---|
| Type information | type table entries | 230 |
| | pointer definitions | 610 |
| | user-provided functions | 22 |
| Sample query | heap size | 85 kilobytes |
| | number of objects | 950 |
| | swizzled pointers | 4300 |
| | elapsed time | 83 milliseconds |

**BLOB mapping performance.** The BLOB mapping approach was applied in a nontrivial sample application.[10] The persistence schema contained more than 160 SMRC flagged class definitions with approximately 260 persistent pointer definitions and several definitions for unions, dynamic arrays, and function pointers (see Table 2). The type table in a heap had more than 230 type entries (it included the embedded types) with more than 610 pointer definitions (including the transient pointers). A typical heap size contained approximately 950 objects that allocated 85 kilobytes of object memory and contained 4300 swizzled pointers. Given the size of the application, swizzling of the entire heap was completed in a respectable 83 milliseconds of elapsed time. If we were to flatten this data rather than use the BLOB mapping, the equivalent relational operation would involve multiple joins across many tables to reassemble the C++ objects. Actually, the worst case of normalizing all the data types—which could result in 160 tables (and could require a 160-way join)—would probably cause the database system to run out of memory.

In contrast to the ADT mapping, which maps an object to a single container, the BLOB mapping maps a heap of (possibly) heterogeneous objects to a sin-

gle container. It is this primary difference (with more detail provided below) that gives the BLOB mapping better performance:

- *No deswizzling*—As the old heap load address is kept in the heap header information, no deswizzling of the pointers within objects of a heap is required. Therefore, the storage operation of a heap is confined to just loading the entire heap of objects into a column of a table.
- *Direct addressability for pointer swizzling*—All of the pointers can be swizzled without search or navigational overhead. The swizzle operation itself consists of a single addition and assignment operation. Obviously, if the heap is loaded in its original location, then no pointers need to be modified (except the vtable pointers).
- *Cluster of objects*—Since a heap represents a self-contained set of objects and references (external pointers are treated differently), the entire heap can be adjusted in one swizzle call. No further I/O or memory allocation operations would be required to additionally load or swizzle internally referenced objects.

## Summary and outlook

In this paper, we have described an approach to making C++ persistent using an RDBMS. Although many bridge technologies between object-oriented and relational systems have recently appeared in research publications and product lists, the SMRC approach is still unique, as it pursues a tight language binding by storing objects in the same binary format in which they were created in the host language. As this binary format is a "black box," the database system can only provide container functionality, i.e., storage management, and not use the data contents directly in most relational operations, although we have discussed alternatives that use UDFs to expose parts of the object. SMRC does not require a new object model or database language for persistence, but instead simply employs C++ and the industry standard SQL. This approach preserves the object-oriented language features of C++, such as inheritance and substitutability, while adding persistence and object relocation. Objects are stored via the ADT or BLOB mappings, as appropriate for the application, and can be cross-referenced via external pointers. Our approach is compatible with class libraries, as it does not require a modification of the class definitions to inherit persistence properties from a common root class. Thus, third party C++ class library software can be used on both client and server sides.

Although our general design is complete, there are still some implementation details missing. A primary issue is heterogeneous portability. Currently, the SMRC schema compiler works only with the IBM AIX C Set++ compiler. However, as the schema compiler generates schema information as C++ source code, the produced schema files then can be used on any platform. Furthermore, SMRC requires a homogeneous client/server platform for the object format. Unfortunately, the problem of building a general-purpose object translator (including the translation of the method code) across multiple platforms is extremely difficult. Interestingly, the solution may lie in a different language such as Java**,[28] a new object-oriented programming language offered by Sun Microsystems, Inc. Java, an interpreted language, is machine-independent and can be used to create stand-alone applications or program fragments. Java methods can easily be moved across platforms to any machine that has a Java interpreter. We are exploring this possibility.

A secondary issue is the implementation of external references. One of our goals was to work with existing class libraries. Unfortunately, our preferred solution (software swizzling using smart pointers) is not compatible with existing class libraries—the code must be recompiled to use the overloaded dereference (→) operator. Only the less preferred, platform-dependent solution (the page-fault method) is truly compatible with existing class libraries. We are still battling with this dilemma. Furthermore, we must explore other kinds of external references, such as uniform reference locators (URLs) and OLE references.

## Acknowledgments

Our thanks to Patrick Gainer, V. Srinivasan, and Stefan Dessloch for their help in preparing this paper. Mike Carey gave us valuable comments to present and relate our work to the state of the art.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of CI Labs, Microsoft Corporation, Object Design, Inc., 02 Technology, Servio Corporation, Versant Object Technology, UniSQL, Inc., Illustra Information Technologies, Inc., Persistence Software, Inc., Subtle Software, Inc., Lotus Development Corporation, or Sun Microsystems, Inc.

## Cited references and notes


1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA (1995).
2. R. Cattell, *Object Data Management: Object-Oriented and Extended Relational Database Systems*, Addison-Wesley Publishing Co., Reading, MA (1991).
3. *Communications of the ACM*, R. Cattel, editor of special section on next-generation database systems **34**, No. 10, 30–120 (October 1991).
4. A. Kemper and G. Moerkotte, *Object-Oriented Database Management: Applications in Engineering and Computer Science*, Prentice Hall, Englewood Cliffs, NJ (1994).
5. W. Kim, "UniSQL/X Unified Relational and Object-Oriented Database System," *Proceedings of the SIGMOD International Conference on Management of Data* (Minneapolis), ACM Press, New York (1994), p. 481.
6. *Illustra User's Guide, Illustra Server*, 2.1 edition, Illustra Information Technologies Inc., 111 Broadway, 20th floor, Oakland, CA 94607 (June 1994).
7. D. Chamberlin, *Using the New DB2: IBM's Object-Relational Database System*, Morgan-Kaufmann, San Francisco, CA (1996).
8. *Database Language SQL3*, J. Melton, Editor, American National Standards Institute (ANSI) Database Committee (X3H2), (August 1994).
9. R. Ananthanarayanan, V. Gottemukkala, W. Käfer, T. Lehman, and H. Pirahesh, "Using the Co-Existence Approach to Achieve Combined Functionality of Object-Oriented and Relational Systems," *Proceedings of the SIGMOD International Conference on Management of Data* (Washington, DC), ACM Press, New York (1993), pp. 109–118.
10. B. Reinwald, S. Dessloch, M. Carey, T. Lehman, H. Pirahesh, and V. Srinivasan, "Making Real Data Persistent: Initial Experiences with SMRC," *Proceedings of the International Workshop on Persistent Object Systems*, M. Atkinson, D. Maier, and V. Benzaken, Editors, Tarascon, France, 1994, Workshops in Computing, Springer-Verlag, Berlin (1995), pp. 202–216.
11. SMRC (shared memory-resident cache) is commonly pronounced "smarc," and should be on CompuServe™ under the GO DB2 area. Contact the authors for more information.
12. We must honestly say that we "mostly" achieve these goals because although our general design does achieve them, our prototype implementation only comes close. However, parts of the SMRC implementation are already used in the DB2 common server for storage and retrieval of C++ objects used in its sophisticated visual explain tool. We point out the differences between the SMRC design and implementation at the appropriate points in this paper.
13. C. Date, *An Introduction to Database Systems*, sixth edition, Addison-Wesley Publishing Co., Reading, MA (1995).
14. M. Loomis, "Object and Relational Technologies," *Object Magazine* 35–43 (Jan. 1993).
15. *Persistence User Manual*, 1.2 edition, Persistence Software Inc., 1650 S. Amphlett Blvd., Suite 100, San Mateo, CA 94402 (1993).
16. *Subtleware for C++/SQL: Product Concepts and Overview*, Subtle Software Inc., 1 Albion Road, Billerica, MA 01821 (1994).
17. D. Linthicum, "Rethinking C++," *DBMS Magazine* **8**, No. 5, 1–2 (1995).
18. *SOMobjects, User's Guide*, SC23-2680, IBM Corporation (June 1993); available through IBM branch offices.
19. C. Lau, *Object-Oriented Programming Using SOM and DSOM*, Van Nostrand Reinhold, Thomson Publishing Company, New York (1994).
20. R. Sessions, *Object Persistence—Beyond Object-Oriented Databases*, Prentice Hall, Englewood Cliffs, NJ (1996).
21. 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," *Proceedings of the International Conference on Data Engineering* (Vienna), IEEE Computer Society, New York (1993), pp. 651–661.
22. H. Pirahesh and C. Mohan, *Evolution of Relational DBMSs Toward Object Support: A Practical Viewpoint*, Research Report RJ8324, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120 (1991).
23. The class library with the method implementation has to be compiled for the same platform, on the client and server side. Otherwise, an object conversion (from the originating client representation to the server representation) is required to incorporate different architectures.
24. The name "BLOB mapping" has been chosen, as the fields of the relational table are of database type BLOB. The BLOB type is a special form of LOB. In DB2, it can contain up to two gigabytes of binary data.[25]
25. T. Lehman and P. Gainer, "DB2 Lobs: The Teenage Years," *Proceedings of the 12th International Conference on Data Engineering* (New Orleans), IEEE Computer Society, New York (1996), pp. 192–199.
26. *Application Program Interface (API) User Guide Release 4.0*, Lotus Development Corporation, 55 Cambridge Parkway, Cambridge, MA 02142 (1995).
27. Object-level incompatibility is created by competing compiler vendors, and could be solved by language-neutral development environments like SOM (System Object Model).[18,19] SOM supports the building and packaging of binary class libraries so that object classes produced by one C++ compiler can be used from C++ programs (or even other languages) built with another compiler. An interpreted language such as Java[28] might also be able to solve the heterogeneity problem.
28. A. Van Hoff, S. Shaio, and O. Starbuck, *Hooked on Java*, Addison-Wesley Publishing Co., Reading, MA (December 1995).
29. *C Set ++ for AIX/6000, User's Guide*, SC09-1605, IBM Corporation (1993); available through IBM branch offices.
30. Typically, users store many columns with one SQL insert statement, including the C++ object column. This is important for performance. SQL even allows multiple record inserts, resulting in better performance due to more set-oriented processing.
31. M. Zimowski, "DRDA, ISO RDA, X/Open: A Comparison," *Database Programming & Design*, 54–61 (June 1994).
32. R. Orfali, D. Harkey, and J. Edwards, *Essential Client/Server Survival Guide*, Van Nostrand Reinhold, Thomson Publishing Company, New York (1994).
33. In C++, a similar concept to external pointers is called *smart pointers*, because they behave more intelligently and perform additional work compared to normal C++ pointers.[34]
34. M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Co., Reading, MA (1990).
35. R. Cattell, *The Object Database Standard: ODMG-93*, Morgan-Kaufmann, San Francisco, CA (1994).
36. S. White and D. Dewitt, "A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies," *Pro-*

*ceedings of the Conference on Very Large Data Bases (VLDB)*, Vancouver, Canada (1992), pp. 419–431.
37. A. Biliris, S. Dar, and N. Gehani, "Making C++ Objects Persistent: The Hidden Pointers," *Software—Practice and Experience* **23**, No. 12, 1285–1303 (1993).
38. C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System," *Communications of the ACM* **34**, No. 10, 50–63 (1991).
39. T. Andrews, *The Vbase Object Database Environment*, A. Cardenas and D. McLeod, Editors, Research Foundations in Object-Oriented and Semantic Database Systems, Prentice-Hall, Inc., Englewood Cliffs, NJ (1990).
40. R. Agrawal, S. Dar, and N. Gehani, "The O++ Database Programming Language: Implementation and Experience," *Proceedings of the International Conference on Data Engineering* (Vienna), IEEE Computer Society, New York (1993), pp. 61–70.
41. The row_id consists of a temporary and a permanent part. The temporary part is used as a hint only. If the referenced object is part of a table that is reorganized, the record may get a new row_id. In this case, the permanent part of the row_id is used to find the record. The temporary and permanent part of a row_id might be implemented as rids (record ids) in relational databases.
42. J. Moss, "Working with Persistent Objects: To Swizzle or Not to Swizzle," *IEEE Transactions on Software Engineering* **18**, No. 8, 657–673 (1992).
43. Many operating systems provide primitives for memory mapping, manipulation of page protection, and setting up of signal handlers to be invoked in the event of an access violation (e.g., mmap, mprotect, and sigaction[44]). In this way, the ability of the paging hardware to trap access violations can be exploited by systems like SMRC.
44. *AIX General Programming Concepts*, SC23-2205, IBM Corporation (1993); available through IBM branch offices.
45. P. R. Wilson and S. V. Kakkad, "Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware," *Proceedings of the International Workshop on Object Orientation in Operating Systems* (Paris, France), IEEE Computer Society, New York (1992), pp. 364–377.
46. S. White and D. DeWitt, "QuickStore: A High Performance Mapped Object Store," *Proceedings of the SIGMOD International Conference on Management of Data* (Minneapolis), ACM Press, New York (1994), pp. 395–406.
47. A. Hosking and E. Moss, "Protection Traps and Alternatives for Memory Management of an Object-Oriented Language," *Proceedings of the 14th Symposium on Operating Systems Principles* (Asheville, NC), ACM Press, New York (1993), pp. 106–119.
48. A. Hosking and E. Moss, "Object Fault Handling for Persistent Programming Languages: A Performance Evaluation," *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages, and Applications* (Washington, DC), ACM Press, New York (1993), pp. 288–303.
49. M. Carey, D. DeWitt, and J. Naughton, "The OO7 Benchmark," *Proceedings of the SIGMOD International Conference on Management of Data* (Washington, DC), ACM Press, New York (1993), pp. 12–21.
50. It was awkward to program this case, as we had to expose private data members. We could not keep inheritance and C++ enumerated types, and we flattened out arrays and replaced pointers by indices. For performance, however, this seemed to be the fastest approach to store the data in relational tables.
51. For fairness, we also plan to run the test where the flattened object can use one or more attributes to answer the query directly, without requiring the C++ object to be reassembled first.

*Accepted for publication December 22, 1995.*

**Berthold Reinwald** *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: reinwald@almaden.ibm.com).* Dr. Reinwald has been a research associate at the IBM Almaden Research Center since December 1995, joining IBM in July 1993 as a postdoctoral fellow. He designed and implemented a persistent object system on top of DB2 Common Server, and was involved in the development of IBM's Visual Explain™ tool as part of DB2 Common Server. Dr. Reinwald has significantly contributed to the area of workflow management, and has been active in several areas of object-oriented languages and systems. His recent research activities cover several aspects of workflow and data management systems, including transactions, distributed architectures, communication infrastructures, object-oriented extensions, and persistence. Dr. Reinwald received a Ph.D. in computer science from the University of Erlangen-Nürnberg in 1993 (best Ph.D. thesis award from the university). He is the author and coauthor of several conference and journal papers, and has published a book about workflow management (in German).

**Tobin J. Lehman** *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: toby@almaden.ibm.com).* Dr. Lehman joined the IBM Almaden Research Center in 1986, shortly after finishing his Ph.D. degree from the University of Wisconsin-Madison. His thesis introduced a number of novel concepts for memory-resident database systems, such as the T-Tree index structure, dynamic lock granularity, and partition-based logging and recovery. At IBM Research, he participated in a number of projects, including the R* Distributed database project, the ARBRE (a teradata-like database machine) project, the Almaden Computing Environment project, the Starburst extensible database system project (designing and implementing both a large object system and a memory-resident storage system). Dr. Lehman codirected the successful SMRC (shared memory-resident cache) project and, with the SMRC project coming to an end, he is currently leading the effort to take the lessons learned there and apply them to general-purpose abstract data type (ADT) support in a relational DBMS.

**Hamid Pirahesh** *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: pirahesh@almaden.ibm.com).* Dr. Pirahesh has been a research staff member at the IBM Almaden Research Center since 1985, involved in research, design, and implementation of the Starburst extensible database system. Dr. Pirahesh has close cooperations with the IBM Database Technology Institute and IBM product divisions. He also has direct responsibilities in the development of IBM's DB2 Common Server product. He has been active in several areas of database management systems, computer networks, and object-oriented systems, and has served on many program committees of major computer conferences. His recent research activities cover various aspects of database management systems, including extensions for object-oriented systems, complex query optimization, deductive databases, concurrency control, and recovery. Dr. Pirahesh is an associate editor of *ACM Computing Surveys* journal. He received M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles and a B.S. in electrical engineering.

**Vibby Gottemukkala** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: vibby@watson.ibm.com).* Dr. Gottemukkala is a research staff member in the Data Intensive Systems department at the IBM Thomas J. Watson Research Center, where he is currently investigating issues in interfacing parallel applications and databases, integrating database systems with tertiary storage, and designing databases for 64-bit architectures. His research interests also include storage architectures for parallel and distributed databases, distributed shared memory and its application in database systems, and database concurrency and coherence control. He received a Ph.D in computer science from the Georgia Institute of Technology in 1995.

Reprint Order No. G321-5600.