Persistence Software: Bridging Object-Oriented Programming and Relational Databases[1]

Arthur M. Keller[2]
Stanford University
and
Persistence Software

Richard Jensen[3]
Persistence Software

Shailesh Agarwal[4]
Persistence Software

ABSTRACT

Building object oriented applications which access relational data introduces a number of technical issues for developers who are making the transition to C++. We describe these issues and discuss how we have addressed them in Persistence, an application development tool that uses an automatic code generator to merge C++ applications with relational data. We use client-side caching to provide the application program with efficient access to the data.

1. INTERFACING C++ CLASSES WITH RELATIONAL DATA

Object orientation promises dramatic benefits in software productivity, quality and reusability. Yet as with most technology innovations, it requires a significant break from the development practices of the past. Specifically, the difficulty of integrating objects with relational databases has emerged as a major barrier to adoption of object technology by developers who have a significant existing base of hierarchical or relational data.

Today, C++ developers have to hand code an interface between their objects and their existing relational databases. For many projects, this task alone accounts for 20 - 30% of the total programming effort. Using a code generator to automate this work can provide improvements in productivity and quality of C++ applications. This approach also provides a way for companies to transition to C++ applications while leveraging existing investments in relational data.

2. ALTERNATE APPROACHES

One approach for interfacing C++ classes to a relational database is through a C++ class library, containing classes that model relational entities such as tables, tuples and fields. To build an application, the developer customizes these building blocks by specifying a mapping between a generic tuple instance and a specific class instance. Inheritance, associations and runtime behaviors must be hand-coded.

In contrast, a code generator produces C++ classes directly from the application object model information. The developer can concentrate on the correctness of the model while the code generator automatically creates the database interface portion of the application. The code generator can also create a table schema based on the object model, or map the object model into an existing table schema.

The application object model supplies the information about inheritance, attributes and associations; the code generator translates these inputs into appropriate table structures and C++ class definitions. Classes may be related via a generalization-specialization hierarchy or binary associations. (Persistence does not currently support aggregation of classes or higher-order associations.)

3. BENEFITS OF DATABASE INTERFACE GENERATOR FOR C++/RDB ACCESS

Manually interfacing C++ classes to relational tables is feasible, but becomes tedious and prone to error when many classes exist. It is also hard to ensure that the semantics of the model are enforced. A Database Interface Generator ("generator") automates this repetitive task and provides a uniform interface for each class in the model. By providing consistency checks at the model level, the generator can increase the confidence in the quality of the database access portion of the application.

Description of Database Interface Generator Approach

Using the application object model to drive the code generation preserves the semantics of the model. The code generator takes care of:

• Encapsulating database access for classes and attributes.

---

[1] For further information on Persistence Software, please write to 1650 South Amphlett Blvd., Suite 100, San Mateo, CA 94402 or `info@persistence.com`

[2] Author's address: Stanford University, Computer Science Dept., Stanford, CA 94305-2140, `ark@db.stanford.edu`

[3] Author's address: Persistence Software (see above), `rjensen@persistence.com`

[4] Author's address: Persistence Software (see above), `sagarwal@persistence.com`

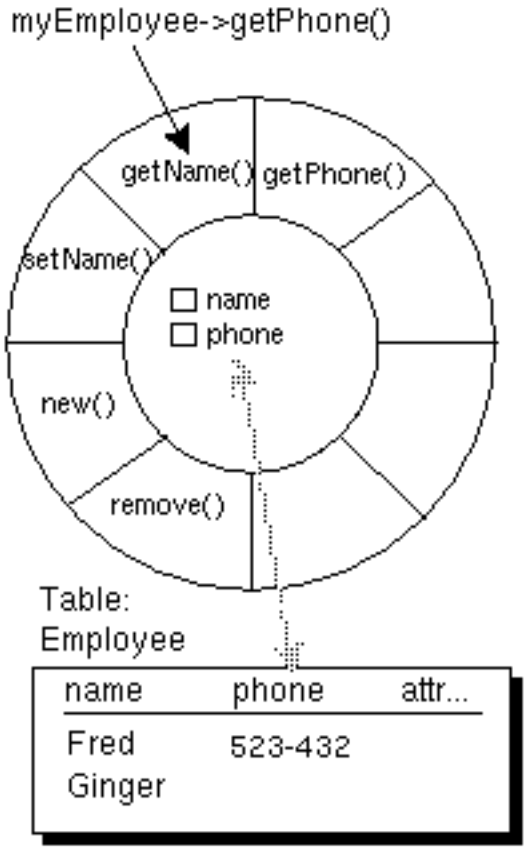

Figure 1. Class definition and mapping.

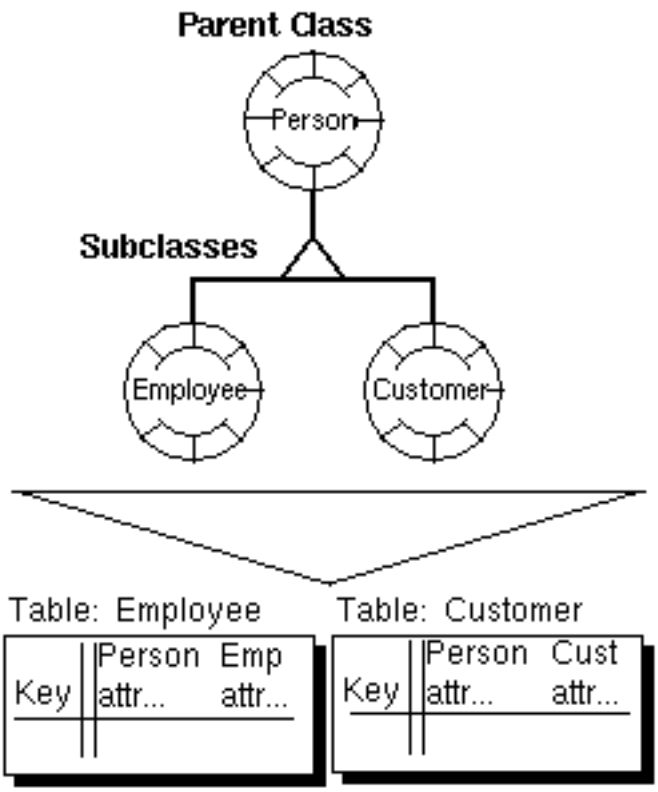

Figure 2. Inheritance.

- Inheriting attributes and methods from parent classes.
- Associating classes with one another.

The generator then emits code that enforces these semantics for each class.

Encapsulating Database Access

C++ developers expect to be able to work with persistent objects which encapsulate the details of data access. The generator performs this encapsulation by mapping classes and attributes in the application object model to tables or views in the database.

The developer has a choice of specifying the primary key attributes for each table or asking the generator to create and maintain a unique OID for each instance. Within the class, the generator provides a create and remove method for instances and a set and get method for each attribute, fully encapsulating the details of data access within the methods of each class (see Figure 1).

Inheriting Methods and Attributes

Inheritance in the application object model maps to single inheritance in the C++ classes. (Persistence currently does not support multiple inheritance). Only leaf classes are mapped to tables in the database (horizontal partitioning). Attributes and associations from parent classes are automatically propagated down to column definitions in the leaf class table (see Figure 2).

Horizontal partitioning of tables for inheritance minimizes the total number of tables and speeds access for single instances (e.g., fetch Employee where name = “Smith”). It has the disadvantage, however, of slowing access for queries across parent classes (e.g., fetch Person where name = “Smith”) because the query must be replicated across each subclass table.

An alternate technique for inheritance would be to map each class to a table (vertical partitioning). This approach speeds queries across parent classes but slows retrieval for single instances, which are forced to access several tables to “rebuild” themselves each time they are accessed (e.g., first read the Parent table, then the Employee table to retrieve the object where name = “Smith”).

Associating Classes

Associations in the object model are implemented by foreign keys in the database. For associations, the code generator creates get and set methods in each class to access instances of the other class through the association. For example, suppose we have two classes, Department and Employee. Each Department employs zero to many Employees, and each Employee works in one and only one Department (see Figure 3).

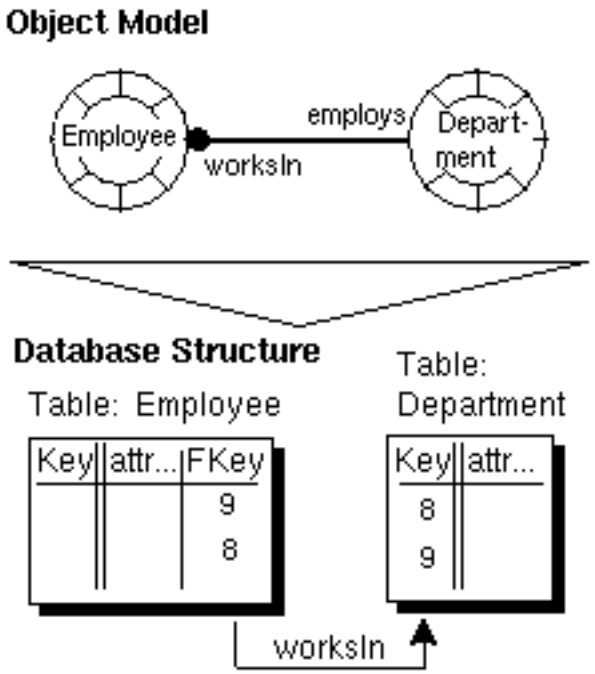

Figure 3. Association.

For this object model, the generator would create a getEmploys() method in the Department class to get all the Employees associated with a particular Department by performing a foreign key lookup in the Employee table. This allows direct support of navigational queries in the developer's C++ application. Similarly, addToEmploys() and rmvFromEmploys() would add and remove instances from the set of Employees related to a particular Department.

The cardinality of binary associations are enforced via column constraints in the database and code in the C++ classes. For example, an Employee must work in one and only one Department. Therefore the constructor for the Employee class created by the generator will take a Department as a required attribute.

The generator maintains the semantics of the association during deletion via code in the C++ classes. The developer specifies a delete constraint: block, propagate, or remove (and set foreign key to NULL) for each class in each relationship. For example, on deletion of a Department, the delete action "block" would block the deletion if there are any Employees associated with this Department, while the delete action "propagate" would propagate the deletion through to all associated Employees.

Violations of the delete constraints are reported to the user via an error mechanism-regardless of whether the error was detected by the database or by the generated C++ classes.

Benefits of Database Interface Generator Approach

A Database Interface Generator (“generator”) can encapsulate database access for C++ classes, providing productivity, quality and reuse benefits over other approaches.

Increasing Productivity

The most significant benefit from a generator is the productivity it provides for applications which access relational data. By automating the creation of database interface methods for each class, a code generator can reduce the total development time for such an application by 20 to 30%.

A generator also reduces the impact of changes in the model. This can be especially true during a prototyping phase for a project. Significant changes to the application object model can be implemented in minutes rather than weeks, enabling an iterative approach to building the application.

As the class hierarchy is changed, or new attributes or associations are needed, the database access code is quickly modified with the developers devoting their time to changing their use of the interfaces for the regenerated classes--work that would have been necessary anyway, but a much smaller fraction of the application needs to be changed.

Improving Code Quality

The generator produces code according to a set of rules. Over time, these rules will be more thoroughly tested than database interfaces written from scratch. As the rules are modified to produce more correct and efficient code, the benefits that have been incorporated into the generator can be applied retroactively to existing applications - thereby improving their quality and performance with little or no productivity impact.

If the generator is careful to preserve the interfaces to the generated classes, this will cost very little. If there are interface changes, the cost of making the necessary adaptations is still small relative to the cost of a team making similar improvements to an existing application.

Reusing Classes

A generator can also increase the reusability of classes. One barrier to reuse is learning the interface for classes created by different developers. With a generator, each class shares a core set of capabilities and a uniform interface. Once developers have learned the basic methods for manipulating one class, they can easily work with other classes.

A second barrier to reuse is that the original developer usually only implements the methods they need to complete a particular task. With a generator, each class contains a complete set of methods for working with the

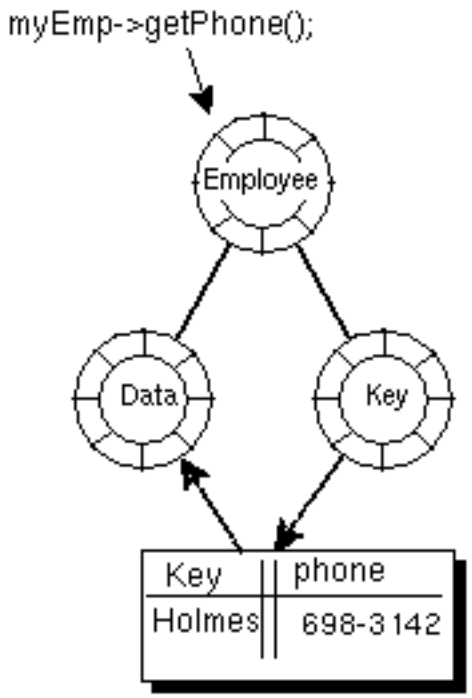

Figure 4. Smart pointer.

database. The developer is assured of having "complete" classes to work with.

4. CONSIDERATIONS FOR RUNTIME OBJECT MANAGEMENT

There are a number of important runtime issues in merging C++ classes with relational databases. Specifically:

• Navigating objects structures which have been read from the database

• Providing concurrent access to multiple users

• Ensuring consistency between cached objects and tuples in the database

Here again, a code generator can be used to produce specialized classes which resolve these issues. In addition, a Runtime Object Management System("runtime system") provides database access and object caching for applications generated by the code generator.

The runtime system enables rapid object access and navigation by swizzling primary and foreign key attributes into in-memory pointers. It also provides data consistency and concurrency by invoking the transaction and locking mechanisms of the underlying databases.

The Runtime Object Model

The methods created by the Database Interface Generator never return a direct pointer to the data. Instead, they return a smart pointer to the data. This smart pointer keeps a reference count and can be shared by several variables. For example, if two queries return the same tuple, both query results would point to the same smart pointer.

The smart pointer in turn contains a pointer to the data for an instance and the primary key value for an instance. To ensure consistency between the object cache and the database, it is necessary to flush the data in the cache each time a transaction is commited. When this happens, the primary key value is used to transparently re-read the data from the database (see Figure 4).

Navigating Object Structures

C++ instances can refer directly to other instances through pointers. Using pointers, C++ developers can build complex in-memory structures which can be quickly navigated by following the pointer links between objects. Relational tuples, however, can only refer indirectly to other tuples through foreign key "pointers." Navigating relational structures, such as a bill of materials, requires a separate query to traverse each link of the structure in each direction.

The runtime system performs the task of converting primary and foreign key values into in-memory pointers, a process we call semantic key swizzling. This has the effect of speeding performance for navigational queries once the object instances have been read in from the database.

The runtime system supports key- based queries over objects in the cache. For any class, given the primary key values, if the corresponding object has already been registered then the runtime system returns its smart pointer. This feature enables applications to selectively cache certain sets of objects which can then be rapidly accessed via the object cache.

In our previous example, each Department employs zero to many Employees. The method, getEmploys() performs a foreign key lookup in the Employee table to retrieve all the Employee instances which are associated with a particular Department. The runtime system creates pointers between the cached Department instance and its associated Employee instances. The next time this association is navigated - either from the Department or the Employee side - the information will be returned immediately from the object cache.

Enabling Concurrent Data Access

The developer uses transactions to control the level of locking performed in the database. The three basic types of transactions currently supported are: dirty read (no locks), consistent read, and read/write. As data is accessed or updated by the application, depending on the type of transaction, the runtime system places the appropriate locks on the corresponding tuples in the underlying database using the database's locking mechanism.

In addition to these transactions, the developer can specify either shared or exclusive locks on entire tables in the database. In cases where the application does not specify any transaction, an implicit read-write transaction is started.

When the application explicitly invokes a transaction, the implicit transaction is committed and the new transaction is started.

Enforcing Data Consistency

As objects are retrieved, tuples corresponding to these objects are locked in the underlying database for the duration of the transaction. When the transaction is committed these locks are released and the data in the cache must be flushed.

When the transaction commits, only the data for the objects is flushed - the smart pointers are retained in the cache. So, the next time data corresponding to any of these objects is requested, the appropriate locks are automatically re-acquired and data is read from the database and cached.

The cache maintains a single copy of the data for the entire application. This avoids duplication of data if different parts of the application have to access data associated with a given object. Maintaining a single copy of data ensures that the data remains consistent. Different parts of the application have access to the latest version of the data and changes in one part of the application are visible throughout the application.

Providing Database Independence

The runtime system manages the database connections, responds to database queries and implements the transaction mechanism. It provides a database independent interface which enables applications to transparently access databases from different vendors.

5. BUILDING APPLICATIONS WITH PERSISTENCE

Persistence is an application development tool which consists of a Database Interface Generator, to convert an application object model into C++ classes and relational tables; and a Runtime Object Management System, to speed data access and provide data concurrency and consistency.

Building an application with persistence follows a four-step process: (i) enter an object model for the application, (ii) generate the database interface classes, (iii) add custom code to the generated classes, (iv) compile and link to the appropriate runtime system.

i. Entering the Object Model

The application in this example will have two classes, Department and Employee. The two classes have a single association: a Department employs zero to many Employees, and an Employee works in one and only one Department.

The developer enters class and attribute information into the Persistence interface. For each class, the developer specifies inheritance and identifies the primary key attributes for the corresponding table.

Next the developer creates the associations between the classes and for each association identifies the foreign key attributes which store that association in the database.

ii. Generating Database Interface Classes

Once the object model is entered, the developer presses the generate button in Persistence to create the corresponding classes. For each object in the model, persistence generates a corresponding class. For example, the files, Department.H and Department.C will be generated for the Department object in the model. Persistence generates a strongly typed ordered collection class, Department_Cltn, to store query and association results.

Since the class files generated by Persistence will be overwritten each time the application object model changes, Persistence provides a set of include files, Department_stubs.H and Department_stubs.C to store custom methods for the class. The stubs files are not overwritten when the object model schema changes, allowing the developer to preserve their changes as the object model evolves.

To handle runtime caching, Persistence also generates a smart pointer, key, and data class for each object - in this example, Department_rep, Department_keyObj and Department_Data. These runtime classes are used to provide object caching and consistency; they are not used by the developer.

iii. Adding Custom Code

Once the code is generated, the developer can add custom methods to the classes and incorporate Persistence-generated classes into other classes and projects. Figure 5 shows a code sample for a simple application that logs in to the database, creates new Department and Employee instances, and sets the association between a Department and an Employee.

iv. Linking To Runtime System

All code generated by Persistence is database independent. At link step, the developer links to the runtime object management system and to the appropriate database library to create a complete application.

6. CONCLUSIONS

We have demonstrated an approach to bridging object-oriented programming to relational databases. Our approach allows new applications to be written in C++ using legacy relational databases while operating concurrently with legacy applications. Our approach has good performance by using queries to efficiently pre-fetch desired data from the database to take advantage of associative search and by caching the data to permit in-memory navigation. Thus, data is loaded into memory based on application needs instead of its physical organization in the database.

```
// Sample method with Persistence generated
// methods shown in bold

void demonstratePersistenceMethods()
{
  // Login to the database
  ROMS::connect("scott","tiger");

  // Create new persistent department
  Department * currDept = new
    Department("Sales","Building 1");

  // Create new employee, assign to department
  Employee * emp =
    new Employee("Jensen", currDept);

  // Read department with key = "Systems"
  Department * existingDept =
    Department::queryKey("Systems");

  // Read all employees who work in Systems dept
  Employee_Cltn * systemsEmpSet =
    existingDept->getEmployees();

  // Update employee relationship
  // (also maintains ref integrity)
  emp->setWorksIn(existingDept);

  // Delete Sales dept from database
  // (also checks delete constraints)
  currDept->remove();
}
```

Figure 5. Application code example.

We use the client-server services of the relational DBMS, but we provide a cache of objects on the client side. This approach of caching on the client side is typically used by object-oriented DBMS. However, we also take advantage of the relational DBMS' server side caching as well as its efficient associative search capabilities. In this way, our customers retain their investment in application software, relational DBMS, and their data while obtaining the benefits of writing new software using the object paradigm.

7. ACKNOWLEDGEMENTS

Chris Keene helped improve the presentation of this material. Derek Henninger assisted in the design of this system. Some of the ideas presented in this work were based on experience of the Penguin project at Stanford University.

8. BIBLIOGRAPHY

[Barsalou 90a] Thierry Barsalou, "View Objects for Relational Databases," Ph.D. dissertation, Stanford University, March 1990, available as technical report STAN-CS-90-1310.

[Barsalou 90b] Thierry Barsalou and Gio Wiederhold: "Complex Objects For Relational Databases," Computer Aided Design, Vol. 22 No.8, Buttersworth, Great Britain, October 1990.

[Cattell 91] Rick Cattell, *Object Data Management: Object Oriented and Extended Relational Systems*, Addison-Wesley, 1991.

[Barsalou 91] Thierry Barsalou, Niki Siambela, Arthur M. Keller, and Gio Wiederhold, "Updating Relational Databases through Object-Based Views," *ACM SIGMOD*, Denver, CO, May 1991.

[Keller 85] Arthur M. Keller, "Updating Relational Databases Through Views," Ph.D. dissertation, Stanford University, February 1985, available as technical report STAN-CS-85-1040.

[Keller 86a] Arthur M. Keller, "The Role of Semantics in Translating View Updates," *IEEE Computer*, **19**:1, January 1986.

[Keller 86b] Arthur M. Keller, "Choosing a View Update Translator by Dialog at View Definition Time," *12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986.

[Keller 86c] Arthur M. Keller, "Unifying Database and Programming Language Concepts Using the Object Model" (extended abstract), *Int.\ Workshop on Object-Oriented Database Systems*, IEEE Computer Society, Pacific Grove, CA, September 1986.

[Keller 87] Arthur M. Keller and Laurel Harvey, "A Prototype View Update Translation Facility," Report TR-87-45, Dept. of Computer Sciences, Univ. of Texas at Austin, December 1987,

[Law 90] Kincho H. Law, Gio Wiederhold, Thierry Barsalou, Niki Sambela, Walter Sujansky, and David Zingmond, "Managing Design Objects in a Sharable Relational Framework," CIFE, Stanford University, March 1990, ASME meeting, Boston, August 1990.

[Lee 90a] Byung Suk Lee and Gio Wiederhold, "Outer Joins and Filters for Instantiating Objects from Relational Databases through Views," Center for Integrated Facilities Engineering (CIFE), Stanford University, Technical Report 30, May 1990.

[Lee 90b] Byung Suk Lee, "Efficiency in Instantiating Objects from Relational Databases Through Views," Ph.D. dissertation, Stanford University, December 1990, available as technical report STAN-CS-90-1346.

[Wiederhold 86] Gio Wiederhold, "Views, Objects, and Databases," IEEE Computer, **19**:12, December 1986.

[Wiederhold 89] Gio Wiederhold, Thierry Barsalou, and Surajit Chaudhuri, "Managing Objects in a Relational Framework," Stanford Technical report CS-89-1245, January 1989, Stanford University.

[Wiederhold 91] Gio Wiederhold, Thierry Barsalou, Byung Suk Lee, Niki Siambela, and Walter Sujansky, "Use of Relational Storage and a Semantic Model to Generate Objects: The PENGUIN Project," Database `91: Merging Policy, Standards and Technology, The Armed Forces Communications nd Electronics Association, Fairfax VA, June 1991, pages 503-515.