Reflections on Object-Relational Applications

Paul Turner,[1] Arthur M. Keller[2]

1.0 Abstract

This paper presents several of the issues, problems, and solutions encountered by developers when creating object-relational applications with C++ and RDBMS using Persistence. Architectural, modeling, and RDBMS issues are the primary areas covered. The intent is provide some practical insights, hints, and guidelines for building such applications.

2.0 Introduction

For the past several years, many large companies have been building object-relational systems utilizing C++, various relational databases and Persistence. Along the way, the development groups constructing these (largely) business applications have encountered many issues, problems, and solutions. We have had the opportunity to listen to and sometimes work with these developers during this time. The intention of this paper is to present, in a practical manner, some of the insights and lessons learned by these developers.

By way of background, the vast majority of the companies involved have been telecommunications companies. Other industries include transportation and finance. At the companies' request, we will not reveal their identities. The types of applications covered include network management, provisioning, various order entry and other front-end customer service applications, logistics, and scheduling.

The remainder of this paper is organized as follows. First, we will briefly review the general nature of object-relational applications using Persistence to integrate C++ with an RDBMS and how they operate. Then we will discuss some non-technical issues that are common and/or critical for object-relational systems. Following that, we discuss several technical issues roughly broken down into the categories of general architecture, modelling and mapping issues and database issues. Within each of these sections, there are several aspects about object-relational systems we want to touch on, namely: the development process, reliability and integrity, performance and portability. Again, we are attempting to present some practical insights; neither completeness nor objectivity are claimed.

---

1. Author's address: Persistence Software, 1700 South Amphlett Blvd., Suite 250, San Mateo, CA 94402, turner@persistence.com.

2. Author's address: Stanford University, Computer Science Dept., Stanford, CA 94305, ark@cs.stanford.edu

3.0 Persistence Applications

Object-relational applications use Persistence to provide an object-oriented interface to relational data. In such applications, the application object model is mapped to a relational schema for the underlying database. Typically such applications are “two-tiered” or “three-tiered” as shown in Figure 1. Persistence provides domain specific classes which the developer code (“logic”) directly utilizes. Tuples in a relational database are presented as instances of a class and any updates on those instances are translated into updates of the corresponding tuples in the database. The two-tier case is generally straightforward with database vendor supplied network communications software. The three-tiered case is often more complicated. Issues here include object location and the communications mechanism between object server and client.

Persistence applications run on many platforms and environments, including Sun, HP, GIS and several different C++ compilers and databases can be used (Oracle, Sybase, Informix, Ingres).

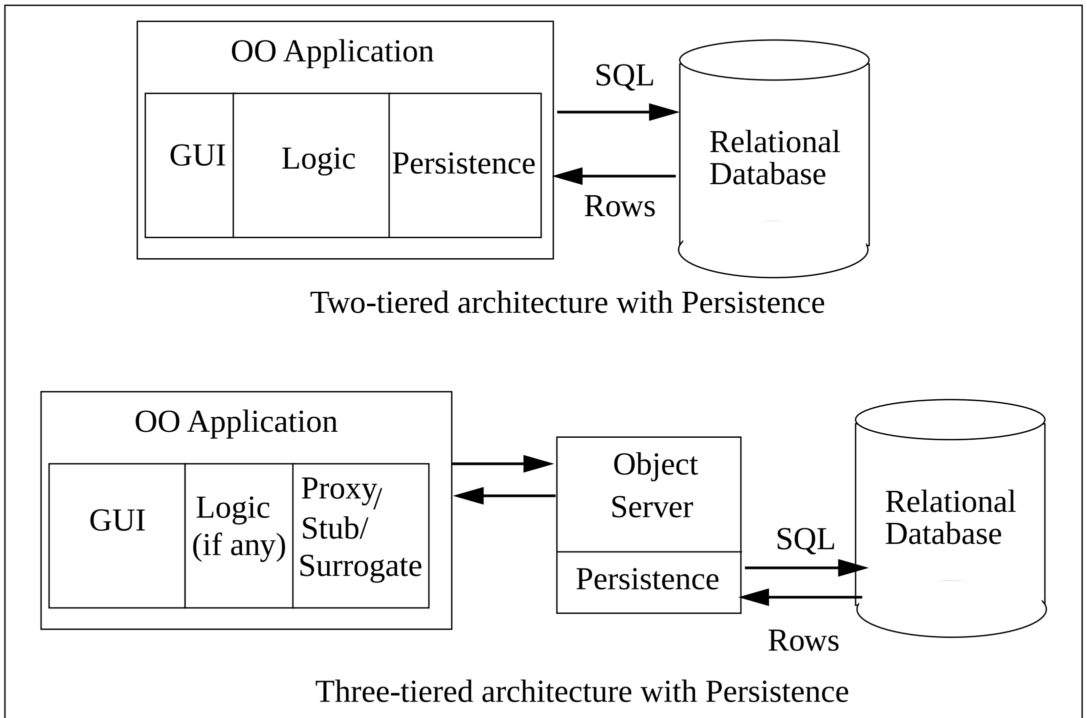

Figure 1: Common object/relational architectures with Persistence

4.0 Non-technical Issues

We have observed several non-technical issues that surface with some frequency on object-relational development projects. Despite a general awareness of some of these issues, it is surprising how often they are ignored or inadequately addressed. One broad category of issues has to do with scoping the development project and then adequately providing resources for the project. Generally speaking, customers tend to be smart about taking on a small OO piece first in order to ease into the technology. Typically, either a small self-contained project is attempted or a component of a larger system. In either case, one needs to put in some forethought about the degree (amount) of object-orientation. For example, should the project use a purely OO language? A purely OO design? Implementation? A lack of planning on this issue often results in "OO creep." Developers, enamored with the technology, find more and more areas within the project where it can be applied.

Another key management issue is factioning. Often, any given developer will have either a strong OO bias or strong RDBMS bias. While one may argue in theory these are orthogonal positions, in practice they are not. For example, OO practitioners will press for a pure object model with no concern for how it will impact the RDBMS being used. RDBMS folk may push a data-centric design and even a more "structured and procedural" application design. It is critical that at least some developers have a firm grounding in both the OO and RDBMS camps.

Prototyping, testing, and measuring have been highly correlated with success in our experiences. Often, object-relational applications are new to both management and developers. The technical newness and uncertainties fuel management worries. Having a working benchmark that quantitatively demonstrates adequate functionality and performance is both reassuring and motivating. A particular problem we have noticed in this area is the failure to allocate adequate time and resources for prototypes and review.

Finally, adequate resources and training are essential. Do not hesitate to bring in outside help and ask for specialized training.

5.0 Technical Issues

We will discuss technical issues and observations roughly categorized into architecture, modeling and mapping, and database issues. Within each of these categories, we will try to touch on the development process, reliability and integrity, performance and portability.

5.1 Architecture

It is becoming more and more commonplace that OO developers desire some form of a three-tiered architecture. Perhaps the most important issue here is clarity of purpose and design. We observe that the two most common answers to the question "why are you using a three-tiered architecture?" are because it's fashionable or because my management said we should. (We suspect the latter is merely a transitive case of the former.) An unfortunate (but common) result of the above is a system where the middle tier is little more than a data gateway. To be sure, sometimes this is the desired outcome. However, many times customers end up with fat clients and a missed opportunity. On the other hand, we have seen some three-tiered designs that seemed forced. In some situations, heretical though it may be, it was clear to us that a simple two-tiered approach would have sufficed and performed much better.

Let’s consider some actual cases of three-tiered Persistence applications. One approach has been to use inflation/deflation techniques (e.g., “saveguts/restoreguts”) to stream an object to another process (usually on another machine). This technique worked well when the situation called for a self-contained detached object that might be passed indefinitely from server to server. Very often, such an object would carry a timestamp with it for consistency purposes. Another situation where object shipment is desirable is when performance is crucial and the object interaction grain-size is small (e.g., many small and frequent operations on an object) or when server reliability and/or availability were not guaranteed. Problems with this approach include the expense of shipping an object versus remote invocation (what is the service grain-size? what are the object sizes?), maintenance of redundant execution code and environments and similarity of execution environments. Care must be taken to ensure that any given object will behave the same in all potential server environments. For example, are floating point operations the same? What about the current time or the way dates are treated in general? In general, we see that the issue of object transmission versus remote invocation as still open and complicated. A hybrid approach where instance state is transmitted but method invocation takes place remotely has been considered by some Persistence customers (see below).

Amongst customers creating three-tiered systems, another basic debate forms around the presentation to object server communication mechanism. We have seen a lot of in-house development in this area with little justification. Many tools are available, ranging from simple messaging packages to full-fledged distributed object frameworks. The argument that commercial packages are too new and risky (yet new in-house development is not?) seems a little dubious. Furthermore, many products are trying to address the object shipment and remote invocation issue. For example, so-called “smart proxies” promise to provide (and manage) local state caching and remote invocation (at the time of this writing, however, we are not aware of any such implementation using Persistence).

Many of our customers believe they have realized many benefits from a three-tiered architecture, including scalability, integrity, reliability, reduced maintenance, and increased portability. Object servers can act as multiplexors to the database(s) and, by migrating much of the application logic to a few (perhaps homogenous) servers, more clients can be maintained at a reduced cost. Object servers are the keepers of the business rules and the existence of multiple servers helps with availability. Persistence customers have built or are building applications that utilize such products as DCE, DOE, XShell and Orbix.

5.2 Modeling and Mapping

The issue of how much to model and how to organize has been a big issue for a few of our customers. There is a lot of literature that discusses this problem, so we will only add a few comments here. Some customers have wrestled with the scope and responsibility for their object model(s). What should be included? Who should own it? Invariably the term “enterprise models” is thrown into the fray. In general, we have seen models increase in scope and shared access and ownership. Different applications require more and more access to classes in others’ models and start placing demands on class design. In one case a “vertical” structuring was used; each application group maintained its own model. Not surprisingly, the communication overhead in this situation quickly approached N^2, redundant classes emerged and modifications became common and ad hoc. Another related issue is class groupings and dependencies. Here, developers want to organize their classes into groups with clear dependencies between groups. For example, there may be a common set of core classes shared by three applications, but each application also has its own group of classes. Presently, only ad hoc solutions exist for our customers; Persistence is working on mechanisms to enable this sort of development arrangement.

Many issues about the intended use of a class's instances can be leveraged. For example, if it can be determined that a class is “read-only” (instances modified only under specific controlled circumstances), one can usually relax database locking issues and utilize client-side caching more effectively. For high performance applications, the ability to cache read-only instances must be stressed; this can result in literally several orders of magnitude performance improvement. Many methodologies support capturing such notions as access, visibility, concurrency, and so forth.

There has been a lot of discussion about how to represent inheritance trees in a relational database. Rather than rehash what has already been said, we would like to add a few minor points to the discussion. First of all, in our experiences, the decision does not warrant the amount labor it often receives. Our suspicion is that because mapping inheritance trees to RDBMSes is an obvious activity, it receives much attention. Using a horizontal scheme is probably the most extensible (at least as far as managing database changes). One problem with a union scheme that is often overlooked is the “nullable mandatory” issue. The developer must provide a mechanism or policy for attributes that are mandatory within their (sub)class but must (technically) be nullable in the database (as not all sub-classes contain the attribute). Another potential benefit of horizontal schemes is that they split up large tables into several smaller ones. This approach can help with table maintenance, improve performance and concurrency.

Something odd we have seen a few times is the (attempted) use of inheritance to support object views. Essentially, developers are trying to provide the equivalent of an external view (in relational terms) of their objects. For example, there is no reason for the sales department to be concerned with the “engineering” aspects of a class. Because there are no well-defined view mechanisms for objects, developers have tried various approaches. Although we can sympathize with the motivation, we don't think that deriving subclasses to support the notion of different interfaces is the right way to go.

5.3 Database

A general theme that has emerged clearly is: know your database, both its limitations and special capabilities. Though this seems obvious, many developers treat their database as little more than a black-box repository, failing to investigate and tune the database. We suspect that an OO focus and lack of RDBMS awareness are to blame for this phenomenon.

For example, consider locking issues. One customer encountered significant lock waits and deadlocks and the issue was not thoroughly investigated until it became intolerable. The problem turned out to be a combination of page-level locking, application access patterns and surrogate keys. Specifically, several tables with poorly distributed surrogate keys were being used as queues. Because several applications were creating new instances simultaneously, this resulted in a “growing tip” hot-spot. In addition to many of the techniques from the RDBMS world (randomizing keys, padding pages, etc.), one may want to consider access patterns due to traversals. For example, suppose there is a one to many association between two classes (which is realized as two tables with the appropriate foreign key arrangement). Locking problems may arise because some transactions go top-down and others bottom-up when traversing instances. In one customer situation, where deadlock avoidance was preferred over maximal concurrency, it was proposed that traversal access start from the “one” side (effectively creating a tree locking mechanism).

Another locking issue is isolation level. While most major RDBMS products support the ANSI isolation levels, they often have different default levels. Along these lines, Persistence supports a form of optimistic locking, which many developers use to support their particular transaction mix or work around vendor limitations. An example of the latter is supplementing a database which does not support shared locks. In general, we strongly recommend investigating your database’s locking defaults and carefully choosing locking policies consistent with your application needs. In one case, a project was extensively using optimistic concurrency unaware that optimistic concurrency control is not serializable if reads are not verified. Fortunately, they realized this fact and the fact that they needed certain transactions to be serializable early in their development cycle.

Whenever one tries to leverage the capabilities of an RDBMS, the dilemma of exploiting features versus portability arises. There are numerous extensions often supplied by vendors that enhance both functionality and performance, including array/vector interfaces, special querying constructs, and so forth. The problem is how to cleanly exploit such extensions and maintain portability (and avoid vendor “lock-in”) at the same time. With object-relational systems, we have seen two basic approaches: encapsulate vendor extensions in methods or utilize stored procedures. The method approach is fairly straightforward. Isolate vendor-specific semantics and/or SQL syntax in a few specific classes. Many applications which use Persistence provide a specific example. Persistence includes an interface which accepts (nearly) arbitrary restrictions (SQL “where” clauses). Rather than sprinkling “where” clauses throughout their programs, developers often provide some specific parameterized querying methods (often application specific). This, of course, increases portability significantly. In addition to hiding SQL and RDBMS specifics, these querying methods also promote a strong domain-specific interface.

The other approach, stored procedures, allows for further leveraging of RDBMS extensions. Another motivating factor is performance; we know of one case where the use of stored procedures increased DML operations by a factor of four. In some cases, use of stored procedures was necessary for security and integrity reasons. Even though the Persistence applications would only modify data through a well-specified interface, there was nothing to prevent other forms of access from directly modifying data. The appropriate use of grants and stored procedures solved this problem nicely. Of course, there are costs associated with the use of stored procedures, notably re-writing the procedures when porting. (Although the calling interface to stored procedures is similar across different RDBMSes.)

While planning, analysis and design are crucial; there is no substitute for observing and testing your running applications. By watching the SQL traffic to the server and then separately testing suspect statements, one customer was able to identify and correct serious performance and semantic errors. One example involved an incorrectly implemented “in-subselect” construct by an RDBMS vendor that caused a query to increase from 23 milliseconds to 67 seconds! Through careful analysis and testing, the customer was able to identify the exact problem and get the vendors involved.

6.0 Conclusions

We have presented several issues, problems, and solutions that Persistence customers have had to face while developing object-relational applications. While many of these issues are still open and many solutions not obvious, we feel that many of the lessons learned can provide valuable guidance to those considering developing object-relational applications in general, and particularly those using Persistence with C++ and an RDBMS.

We have seen many successful applications completed and many more are currently under construction. Though there are outstanding issues and problems, we believe strongly that object-relational approaches are practical and produce good applications. We expect to continue working closely with customers in the future, learning and sharing experiences along the way.