Architecting Object Applications for High Performance with Relational Databases

Shailesh Agarwal[1] **Christopher Keene**[2] **Arthur M. Keller**[3]

1.0 Abstract

This paper presents an approach for architecting OO applications for high performance with relational databases. The key ideas of this approach are:

- Optimize business object mapping: tune the mapping the between business objects and relational tables to leverage relational technology.
- Perform client object management: use a client-side object manager to minimize database traffic.

This approach enables organizations to derive the benefits of OO technology while leveraging their investments in relational technology.

2.0 Introduction

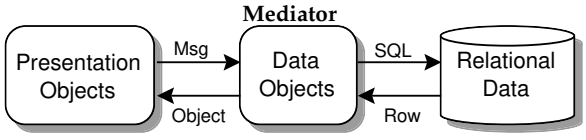
Object-oriented software development is rapidly becoming the leading approach for building flexible, scalable software systems in client/server environments. Additionally, over the past decade, relational technology has matured and has been widely adopted for managing corporate data. Relational databases have now become the standard data stores for on-line transaction processing (*OLTP*) applications. These two trends are motivating the need for building object-oriented applications that access relational databases.

Developers building such *object-relational* applications face some difficult problems such as (i) Mapping the objects from the application model to the relational schema in the database, (ii) Managing the locking and transactions to ensure data integrity, and (iii) Optimizing with consideration of the performance characteristics of relational databases. Each of these problems present several interesting issues worthy of discussion (see for example [Keller93]). However, this paper focuses primarily on the performance issues for such object-relational applications and presents our experiences with architecting high performance object-relational applications.

3.0 Object-Relational Mediators

An object-relational application provides an object-oriented interface to relational data. In such applications the application object model is mapped to a relational schema in the underlying database. Typically such applications build or use a *mediator* for transforming object operations to relational database calls and vice-versa [Wiederhold92].

FIGURE 1. Object/relational mediator



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

2. Author's address: Persistence Software

3. Author's address: Stanford University, Computer Science Department, Stanford, CA 94305-2140 or ark@cs.stanford.edu

As shown in Figure 1, the mediator maps objects to relational tuples and also ensure data integrity with appropriate locking and transaction management. 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. Additionally, the mediator could also provide a client-side object cache for enhancing performance. In a sense, such object-relational mediators enable an OODBMS interface to a legacy RBDMS.

Persistence, an object/relational environment from Persistence Software, is a mediator which creates and manages C++ objects linked to relational databases. Examples of high performance systems built with Persistence include:

- AT&T materials logistics: the LOGIC system supports over 300 warehouses worldwide to manage all of AT&T's network systems inventory. The application, which took 50 developers over two years to build, is the largest object-oriented application ever deployed by AT&T.
- CBIS cellular billing: the Project 2000 system built by Cincinnati Bell Information Systems manages cellular billing for Cincinnati Bell and Sprint. The system, which uses Persistence teamed with an Oracle database and Tuxedo, processes over 1 million transactions daily.

Other examples include the Federal Express Ground Operations Command and Control System, and the Conrail Locomotive Maintenance Scheduler. We have found that there are a number of "ground rules" for achieving high performance from object/relational applications. Developers who do not follow these rules risk building systems which do not scale well in operation.

In one memorable example, a team ported an application from an object database to a relational database using Persistence. The port was completed without changing the object model at all. The resulting application ran properly, but where inheritance had assisted clustering in the object database, it imposed a high performance penalty in the relational database. Through our experiences with companies implementing large scale object systems, we have developed an approach to architecting object applications for high performance.

4.0 Achieving Performance

Our experience has shown that object-relational mapping and high performance can go together, provided proper consideration is given to object mapping and object management issues. Developers can optimize the mapping between a set of interrelated business objects and a relational schema to leverage relational technology. Further developers can use a client-side object cache to enhance application performance.

The basic approaches for optimizing object access to relational data are similar to the objectives for tuning the relational database itself. The two approaches are:

- Maximize Server Performance: Applications should be written to ensure that the queries and updates posed to the server can be processed efficiently (e.g., minimize the use of joins).
- Minimize Server Requests: Applications should be written to minimize the total number of queries and updates sent to the relational data server (e.g., send all the changes to a record at once, rather than sending an update as each attribute is changed).

Architecting applications with these two approaches will enable them to achieve high performance. The following sections discuss these approaches in greater detail.

5.0 Server Performance

The key objective of this approach is to pose only those queries and updates that can be processed efficiently by the server. There are potentially three ways to achieve this objective:

- Choose the appropriate object to relational mappings.
- Use query capabilities of relational databases.
- Take advantage of special performance features of relational databases.

Mapping efficiency between object classes and relational tables is the most critical factor for achieving high performance. “Pure” object models sometimes map poorly to relational structures, for example by requiring many joins to construct simple objects. Such a mapping can significantly deteriorate application performance. Hence, starting with an object model and attempting to map to a relational schema can sometimes lead to poor performance. On the other hand Entity-Relational (E-R) models convert naturally to object models, mapping entities to objects and relationships to associations. E-R models can then be optimized through selective denormalization based on expected usage. The advantage of this approach is that such denormalization is well understand while optimizing object models is not. The best way to achieve this efficiency is to use a normalized relational schema (E-R model) as the basis for the corresponding object model.

Developers will find that the simplest mapping between objects and normalized table typically provides the best performance. Object-relational mapping requires the mapping of the following three major modeling concepts:

- Class to relational rows
- Relationships to foreign keys
- Inheritance to relational rows or joins

5.1 Mapping Classes

The most efficient mapping is most often the direct mapping of a class to a relational table. A more complex mapping can lead to performance penalties. Classes which represent a join between several tables will be expensive to construct and potentially impossible to update [Keller85]. Hence, in general, developers should map each class to a single table; more complex objects can always be built up out of simple, “table-based” classes.

Two exceptions to this rule include creating projection objects to minimize network traffic and view objects for decision support.

- “Projection objects”: for tables with many columns, it may be efficient to map a projection of the table to a “projection class”, and retrieve a full row only when it is needed. For example, a customer row may contain 50 columns, but only the name and phone number are needed most of the time. By creating a class which maps to just the needed columns, the developer can avoid having to pass unnecessary information across the network. Such a class must contain the primary key columns at a minimum.
- “View objects”: for decision support applications, it is often useful to create a view table which represents a database join, and then map this view to an object class. This allows developers to take full advantage of relational algebra to hide the physical data model from the object application [Barsalou91].

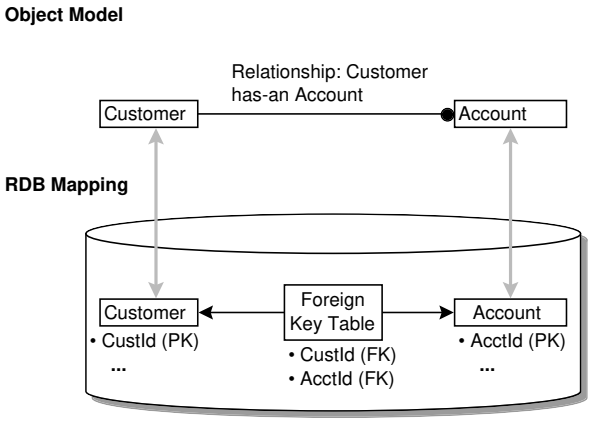
5.2 Mapping Relationships

Relationships between objects map to foreign keys between rows. How each mapping is implemented, however, has a significant impact on the performance and flexibility of

the resulting application. Developers have several choices for mapping object relationships to relational tables:

- Distinct table: In this approach the relationship is represented as a distinct table in the database. This approach provides the most modelling flexibility since it makes adding and removing relationships transparent to other tables. However, this approach can be expensive if the relationship is frequently traversed.

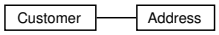
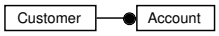
FIGURE 2. Distinct table mapping



- Embedded foreign key: This is the most common approach for one-one and one-many relationships. In this case, for a given class the primary key of a related class is embedded in the class itself. This results in a performance characteristic better than the "distinct table" approach but worse than the "embedded class" approach.
- Embedded classes: In this case, two classes are merged into a single table. Such merging improves performance at the cost of extensibility and violation of normalization. In a one-many relationship, this approach requires a fixed length array of the "many" class to be embedded into the "one" class table.[1]

As shown in the following table, depending on the cardinality of the relationship and the requirement for speed versus flexibility, different relationship mapping techniques are appropriate.

FIGURE 3. Relationship mapping

Relationship Type	Mapping Approach: Embedded Class	Embedded Key	Distinct Table
One-to-one 	- Efficient	- Flexible	- Inefficient
One-to-many 	- Requires embedded array	- Efficient	- Flexible
Many-to-many	N/A	N/A	- Efficient

5.3 Mapping Inheritance

As with relationships, how object inheritance is mapped to the database can have a profound impact on performance. Here, particular attention to expected query paths plays an important role on the mapping choice.

A parent class can be either abstract or concrete. An abstract parent class is one which will never be instantiated on its own, and consequently does not have any physical data associated with it (e.g., no corresponding table). A concrete parent class is one which can exist on its own, and typically will have its own associated table.

Developers also have choices about how to map inheritance associations (for example, the customer class inherits from the person class) into relational tables.

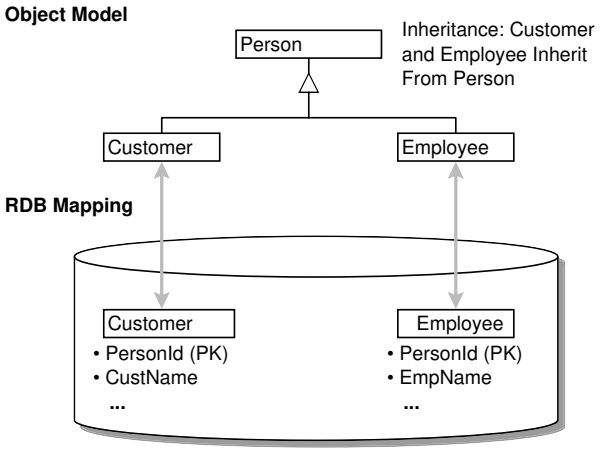
- Vertical Partitioning: In this type of mapping each object model class maps to a corresponding table. The classic object-relational mapping "mistake" is to create

1. Theoretically, an alternative is to embed the "one" class in the "many" class. This option is not supported by Persistence because of resulting update anomalies.

deep inheritance hierarchies which require multi-way joins to retrieve basic object information.

- Horizontal Partitioning: In this case, only leaf classes are mapped to tables and include all of their inherited attributes. This approach may give improved performance since only one table needs to be accessed for instances of a given leaf class.

FIGURE 4. Horizontal partitioning



- Typed Partitioning: Another way to handle inheritance is to map all classes in an inheritance tree to a single table, using a type column to distinguish between subclasses. This enables the retrieval of objects from multiple classes in a single query, but violates normalization.

Queries made at the parent class level will be implemented very differently depending on the inheritance mapping. If the parent class is concrete, a parent class query can be efficiently handled by querying the table corresponding to the parent class. If the parent class is abstract, the query must be run against each child table, a potentially time consuming operation.

On the other hand, retrieving a single instance of a child class is much faster if all the parent classes are abstract, as all necessary columns are located in the child table. Child classes with concrete parent classes must perform expensive joins to retrieve instances.

As shown in the table below, depending on whether the inherited superclass is abstract or concrete and the requirement for speed versus flexibility, different inheritance mapping techniques are appropriate.

FIGURE 5. Inheritance mapping

Inheritance Type	Mapping Approach: Horizontal	Vertical	Typed
Abstract parent Person Customer	- Efficient child fetch	- Inefficient child fetch	- Efficient but inflexible
Concrete parent Customer Retail Customer	- Inefficient parent queries	- Efficient parent queries	- Efficient but inflexible

5.4 Database Query Support

Developers can gain significant performance benefits by using the sophisticated querying capabilities available with relational databases for ad-hoc access of objects. For the initial select of a set of objects, ad-hoc access via the relational query mechanism using indexes is significantly faster than navigational access.

Once the data has been retrieved from the database then in-memory navigational queries allow efficient use of the cached object information. In order to take advantage of this query support, appropriate indexes should be built on columns that are to be used for class and relationship queries. In general, indexes to be built include all primary and foreign keys in the database schema.

5.5 Database Optimizations

Developers of object applications should make sure that their applications take advantage of

any special features that the underlying database provides for optimizing performance. Some examples of such special features are:

- Stored procedures: Implementing object methods to create, read, update, and delete objects through stored procedures can speed performance by 20 percent or more.
- Array and bulk interfaces: Many databases provide array interfaces where updates on a collection of tuples can be sent in a batch. A batched operation is significantly faster than the corresponding operation done on a tuple by tuple basis.
- Asynchronous queries: Some databases also support non-blocking queries where the database server returns control immediately after receiving the query request from the application and the application can then poll the database server for the results of query at a later time. This approach allows the application to optimize its response time and also use multi-threaded capabilities to perform background processing of the query results.

6.0 Minimize Server Requests

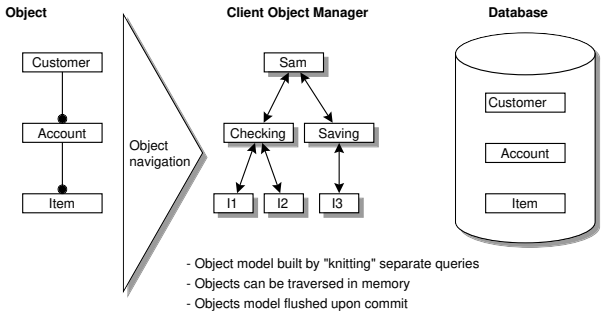
The previous section described how to optimize performance for an object application by choosing the appropriate object-relational mapping and by taking advantage of the features of the underlying database. This section discusses how to enhance performance by using a smart cache on the client side. Intelligent caching of data on the client can significantly reduce the database traffic for an application and provide orders of magnitude performance enhancements [Cattell94].

The basic approach to optimizing client object management is to reduce number of database accesses by using the client side cache as much as possible. This provides a significant performance benefit since: (i) In-memory access is three orders of magnitude faster than disk access, and (ii) Costly network traffic is avoided.

6.1 Client-side cache

The basic model for client object management is to cache data instances read from the database, register their primary key values, and respond to queries based on this cached data. As tuples are retrieved from the database, they are converted to objects and “knit” together according to the object model mapping to form a network of in-memory objects.

FIGURE 6. Object cache



Cached data can also be changed repeatedly without accessing the database, holding the database update until the transaction is committed. This prevents sending multiple updates for the same object to the database. In the case that the transaction is aborted, no changes need to be sent to the database at all. Client-side caching is also critical for ensuring object data integrity [Keller93].

6.2 Optimizing object navigation

Using such an in-memory network, queries which follow foreign key pointers (navigations queries) can be performed fully in the cache once the basic object model information has been retrieved [Keller95]. There is however an issue of how to tell if a query can be satisfied fully from cache data. This is related to the

well-known problem of “phantoms” [Gray93] in databases. There are no simple solutions to the problem in the general case. However, often for specific applications it is possible to assert that the cache contains all the data necessary for the query. In such cases, using the cache for responding to the query can provide significant performance benefits without compromising the validity of the response.

6.3 Concurrency and cache synchronization

Efficient cache management can also speed transaction throughput, particularly by making use of techniques such as optimistic concurrency. In an optimistic transaction model no database locks are placed for cached data. Cache updates are made assuming that the underlying data has not changed. Before sending the cache updates to the database, a check is performed to ensure that the data update is valid. The basic premise of this scheme is that only a small fraction of the updates will be found to be invalid. Such a scheme enables high concurrency without data inconsistencies and lost updates.

Another approach for cache synchronization is to use a notification mechanism which notifies applications of changes in the database that are of interest to those applications. This feature is not commonly available in commercial relational databases.

7.0 Benchmarking Results

Based on our experience, a well-constructed object application should be somewhat faster than a native relational database application: for the “cold” database operations (create, read, update and delete) the application should be within 5 to 10 percent of native database access; while for “warm” operations performed by querying or navigating the object cache, the application should be roughly three orders of magnitude faster than database access.

The actual performance is greatly dependent on the degree to which the application can take advantage of data stored in the object cache. As [Nag95] reported, “Persistence offers many of the performance benefits of an OODBMS while retaining the reliability and portability of the underlying relational database.”

8.0 Conclusions

Developers can achieve high performance building object systems linked to relational databases, provided they make the appropriate trade-offs. Object models must be modified to take into account the strengths and weaknesses of the underlying datastore. Particular care should be given to the complexity of the query required to return a instances of commonly used classes - joins can be deadly in these situations.

Here is a summary of the design guidelines for achieving high performance in object applications linked to relational data:

1. Classes: keep it simple. Map classes to tables where possible, except for special cases such as projection classes to minimize network traffic or view classes for decision support.
2. Relationships: keep it fast. Embed one-to-one relationships into a single table, use foreign keys to navigate one-to-many relationships, avoid many-to-many relationships if possible.
3. Inheritance: balance retrieval speed against query efficiency. In general, horizontal inheritance is the best choice as it minimizes retrieval speeds for child classes. For special situations, vertical or typed inheritance provides faster query responses.

4. Object caching: reuse retrieved data. Caching object data allows multiple updates and is critical to ensuring object integrity.
5. Object navigation: perform client-side relationship queries. Knitting retrieved objects into an in-memory object model provides orders of magnitude performance benefits for data-intensive applications.

By following these guidelines and using the appropriate tools, developers can achieve the performance required for deploying large scale object systems.

9.0 For More Information

For further information on Persistence Software, please contact info@persistence.com or http://www.persistence.com.

10.0 References

[Barsalou91] Barsalou, T., Siambela, N., Keller, A. M., and Wiederhold, G., Updating Relational Databases through Object-Based Views, *ACM SIGMOD Proceedings*, Denver, CO, May 1991.

[Cattell94] Cattell, R., *Object Data Management: Object-Oriented and Extended Relational Database Systems*, Addison-Wesley, Menlo Park, California, revised edition, 1994.

[Gray93] Gray, J., and Reuter, A., *Transaction Processing: Concepts And Techniques*, Morgan Kaufman, San Mateo, California, 1993.

[Keller85] Keller, A. M., "Updating Relational Databases through Views," Ph.D. Dissertation, Department of Computer Science, Stanford University, February, 1985.

[Keller93] Keller, A. M., Agarwal, S. and Jensen, R., "Enabling The Integration of Object Applications With Relational Databases," *ACM SIGMOD Proceedings*, 1993.

[Keller95] Keller, A. M., and Basu, J., "A Predicate-based Caching Scheme for Client-Server Database Architectures," to appear in *VLDB Journal*, 1995.

[Nag95] Nag, B., and Zhao, Y., "Implementing the 007 Benchmark on Persistence," Master's Thesis, Computer Sciences Department, University of Wisconsin-Madison, 1995.

[Wiederhold92] Wiederhold, G., Mediators in the Architecture of Future Information Systems, *IEEE Computer*, March 1992.