Last Updated
January 09, 1998

Architecting Object Applications for High Performance with Relational Databases

Shailesh Agarwal, Ph.D. and Arthur M. Keller, Ph.D.

Abstract
This paper presents an approach for architecting OO applications for high performance with relational databases. This approach enables organizations to derive the benefits of OO technology while leveraging their investments in relational technology.

The key ideas of this approach are:

1. Optimize business object mapping: tune the mapping between business objects and relational tables to leverage relational technology.
2. Perform object cache management: use an application server object manager to share "high activity" objects among many clients, thereby minimizing databasetraffic.

The paper presents concrete examples, benchmarks and customer experiences based on the product, Persistence.

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.

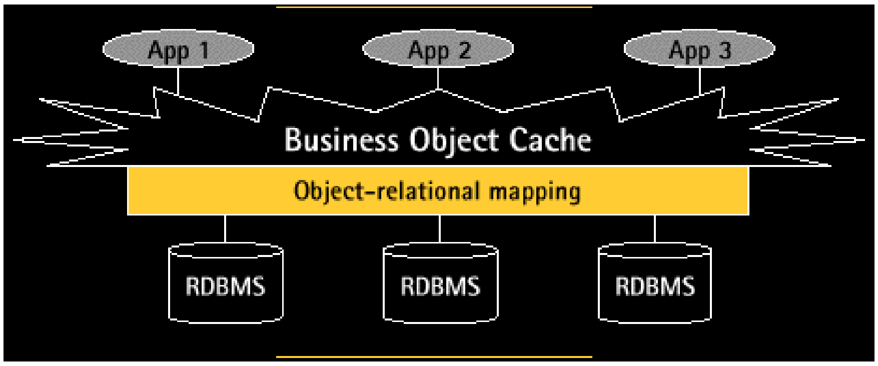
Developers building such object-relational applications face some difficult problems such as (i) Mapping the objects from the application model to the relational schema in the database, (ii) Managing objects in an application server to optimize performance (iii) Managing the locking and transactions to ensure data integrity, and (iv) 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.

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



As shown in Figure 1, the mediator maps business objects to relational tuples and also manages business objects in a shared cache on behalf of several applications.

The object management provided by the mediator can enhance performance through a technique we call "live object caching." The mediator can 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 for building high performance applications based around the notion of a "live object cache." A live object cache maps information from relational tables into business objects, which are easy for developers to use. Accessing and manipulating these business objects in the live object cache is faster than querying the relational database, speeding application performance.

- High Performance Object-Relational Systems Examples
- AT&T network provisioning: the ASOS system provides end-to-end network provisioning for 9 million subscribers, achieving 150 transactions per second against a 600 GB Oracle database
- CBIS cellular billing: the Project 2000 system built by Cincinnati Bell Information Systems manages cellular billing for Cincinnati Bell and Sprint. The system uses Persistence teamed with Oracle and Tuxedo to process over 1 million transactions daily.

Based on over five years helping companies build large scale object-relational systems, we have found that there are a number of "ground rules" for achieving high performance. 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.

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:

- Optimize object-relational mapping: applications should be written to ensure an efficient mapping between application business objects and relational tables.
- Optimize object-relational caching<: applications should be written to take full advantage of object caching and navigation within an application server

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

Optimize object-relational mapping
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

1. Choose the appropriate object to relational mappings.
2. Use query capabilities of relational databases.
3. 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 have less semantic content than object models, but can be converted 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.

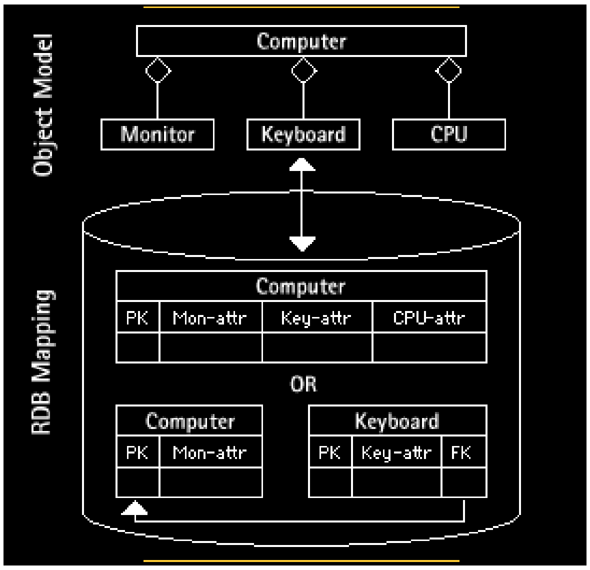
Developers will find that the simplest mapping between objects and normalized table typically provides the best performance. Object-relational mapping requires the mapping of the following three major modeling concepts:

- Simple or aggregate classes or aggregation to relational rows
- Relationships to foreign keys
- Inheritance to relational rows or joins

Mapping Simple and Aggregate Classes
The most efficient mapping is most often the direct mapping of a class or aggregation of classes to a relational table. A more complex mapping can lead to performance penalties. Classes which represent a join between several tables can be expensive to construct and potentially impossible to update [Keller85].

A simple class contains only simple attributes such as integer and character, while an aggregate class can contain attributes which are themselves classes. The following figure shows mapping an aggregation of three classes to a single table.

FIGURE 2. Aggregate class mapping



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].

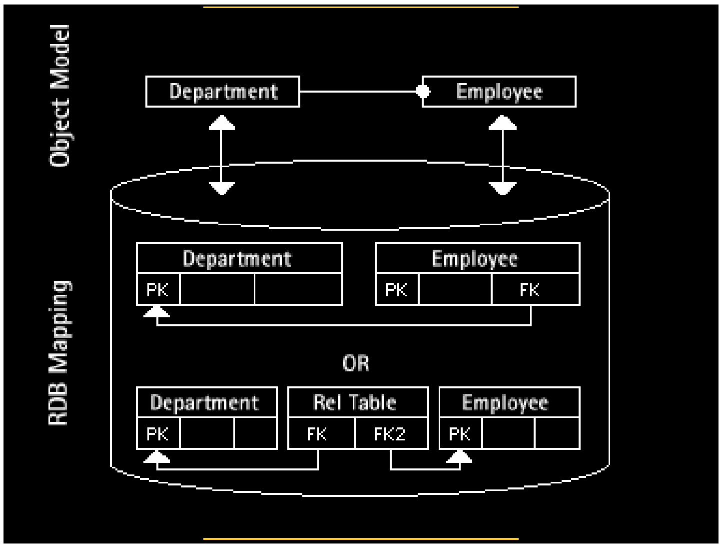
Persistence supports class aggregation, mapping many classes to a single table, and mapping many tables into a single class using views. Persistence also supports join queries, which allows users to construct a network of related objects from a single join query to the database.

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:

> ***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".

FIGURE 3. Relationship mapping



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.

In general, the embedded key approach is most efficient for one-to-one relationships and one-to-many relationships, while the distinct table approach is required for many-to-many relationships.

Persistence supports both the embedded foreign key and the distinct table approaches for mapping relationships. In the case of pre-existing tables, Persistence allows the developer to specify the foreign key mappings, while for new schemas, Persistence can automatically generate the appropriate foreign keys to implement the required mapping.

Mapping Inheritance

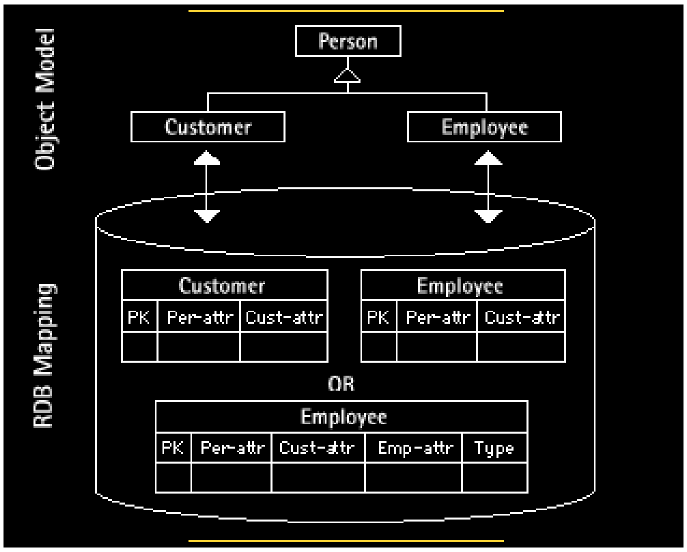
As with relationships, how object inheritance is mapped to the database can have a profound impact on performance. Here, particular attention to expected query paths plays an important role on the mapping choice.

Each parent or child class can be either abstract or concrete. An abstract 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 class is one which can exist on its own, and typically will have its own associated table.

Developers have several choices about how to map inheritance into relational tables.

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. Inheritance mapping



> ***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.
>
> ***Vertical Partitioning:*** In this type of mapping parent classes in the object model class maps to a corresponding table.

The classic object-relational mapping "mistake" is to create deep inheritance hierarchies which require multi-way joins to retrieve basic object information.

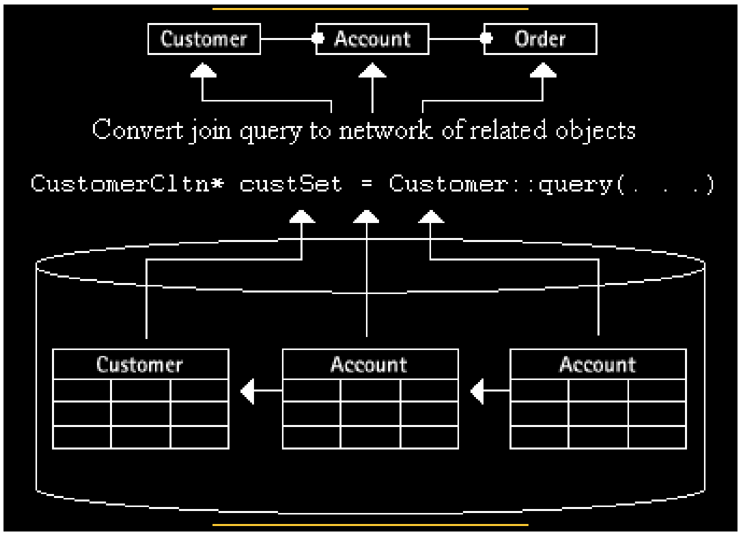
Queries made at the parent class level will be implemented very differently depending on the inheritance mapping. If the parent class is concrete, a parent class query can be efficiently handled by querying the table corresponding to the parent class. If the parent class is abstract, the query must be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

By default, Persistence makes only the base classes concrete in an inheritance hierarchy. To make root classes concrete, the developer can generate the root class using Persistence and then subclass from the root class in their application.

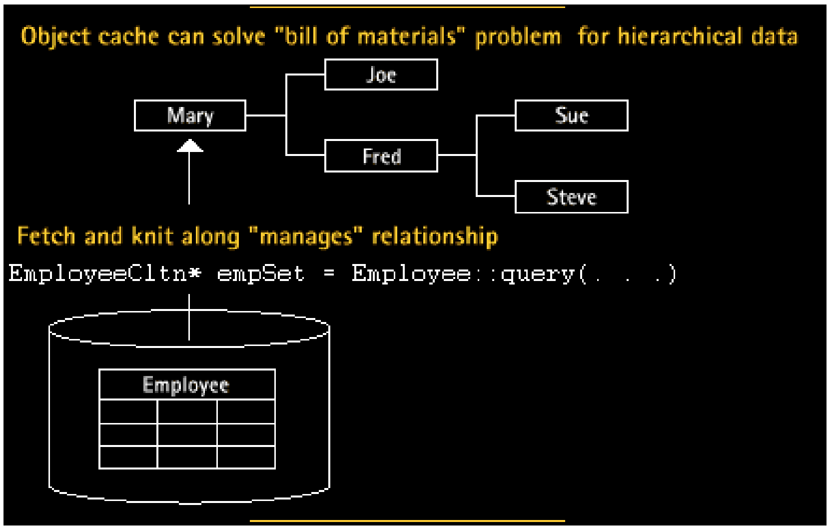
Multi-class join queries
Developers can gain significant performance benefits by using the sophisticated querying capabilities available with relational databases for ad-hoc access of objects. For the initial select of a set of objects, ad-hoc access via the relational join query mechanism using indexes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

FIGURE 5. Join queries



Once the data has been retrieved from the database then in-memory navigational queries allow efficient use of the cached object information. One very important benefit of the object-relational mapping is that it allows developers to create hierarchies of related objects from "flat" tables, providing an elegant solution to the classic bill of materials problem for relational databases.

FIGURE 6. Object knitting



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.

Persistence provides query methods to support inner and outer joins on the database. Persistence also provides flexible "knitting" facilities to turn join rows into hierarchies of related objects.

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.

Persistence can implement any object method as a stored procedure. Persistence also provides class collection operators which implement a set of class operations (such as insert or update) using the underlying database array facilities.

Optimize Object-Relational Caching

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 in distributed systems by using object caching in the application server. Intelligent caching of object data can significantly reduce the database traffic for an application and provide orders of magnitude performance enhancements [Cattell94].

The basic approach to optimizing object data management is to reduce number of database accesses by using the application server 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.

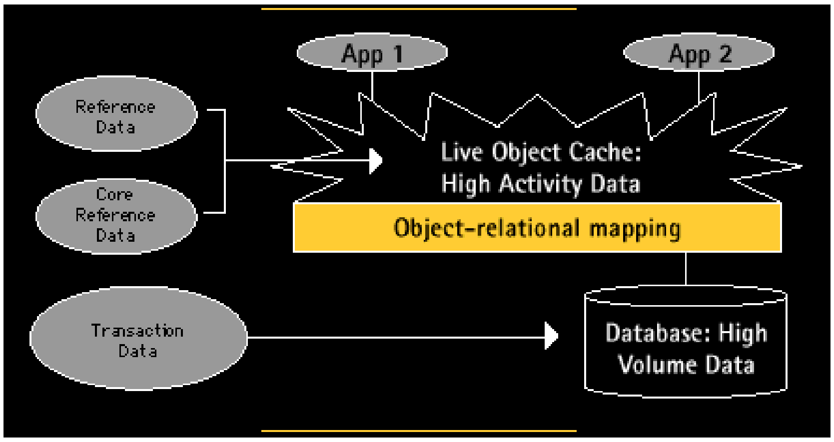
Live object cache

The basic model for 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.

The key to achieving high performance with cached objects is to pre-fetch "high activity" objects into the live object cache before an application starts. High activity objects include reference data as well as core business data which is likely to be shared across transactions. High activity objects might include data such as network elements, flights or portfolios which are being shared by many clients.

In a live object cache, high volume transaction data, such as accounts, tickets or trades are kept in the database and only loaded into the cache on an as-needed basis.

FIGURE 6. Live object cache

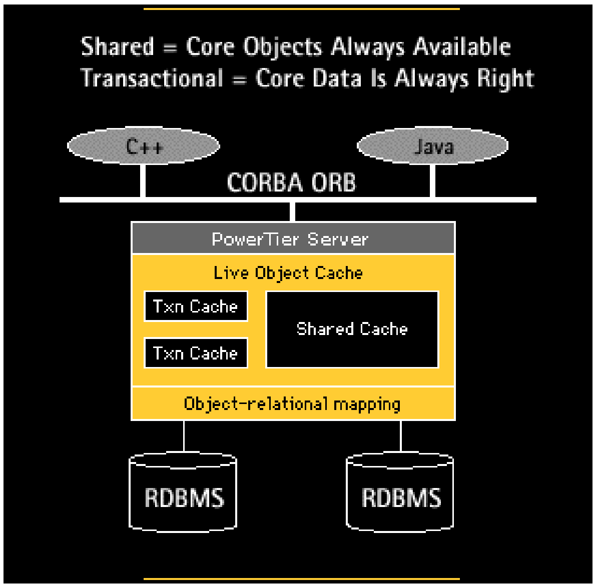


Cached objects can be shared between many users without paying the repeated overhead of querying the database. Further, navigating relationships between cached objects is orders of magnitude faster than querying the database.

Cached objects may 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. Object caching is also critical for ensuring object data integrity [Keller93].

Optimizing object navigation
Using such an in-memory network, queries which follow foreign key pointers (navigation queries) can be performed fully in the shared cache once the basic object model information has been retrieved [Keller95].

FIGURE 7. Transactional object cache



As the following table shows, reads and updates in the object cache are three orders of magnitude faster than reads and updates against the database.

Performance Benchmark

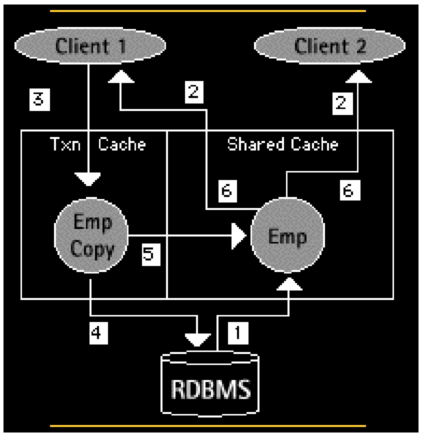
Operation	Time (ms)	Change
RDB Read	22.81	
Read object cache	0.04	533 x faster
RDB Update	27.75	
Update obj cache	0.06	422 x faster

The results reported represent the average time to perform an operation over 5,000 repetitions. The environment for the benchmark was Sun IPX with 32MB RAM, SunOS 4.1.3 and Sybase SQL Server 4.9

Isolation and Concurrency
Efficient management of a shared object cache can speed transaction throughput greatly. In a shared object cache, however, it is critical that uncommitted object changes from each client are isolated from other clients. The following diagram shows how object isolation is achieved in Persistence.

FIGURE 8. Transaction isolation

1. Map data into "Employee" object
2. Object shared in LOC by two clients
3. Client 1 modifies Emp, creating copy in private transactional cache
4. Client 1 commits changes in RDB
5. Emp updated in LOC
6. Clients notified of changes



To speed concurrency, an optimistic locking technique can be used. In an optimistic transaction model no database locks are placed for cached data. Before sending the cache updates to the database a check is performed to ensure that the data update is valid.

Conclusions
Developers can achieve high performance building object systems linked to relational databases, provided they follow a sound approach to object-relational mapping and object caching.

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 simple or aggregation classes to tables where possible. Use projection classes to minimize network traffic, and view classes to map multiple tables to a class.
2. Relationships: keep it fast. Use foreign keys to navigate 1:m and 1:1 relationships. Use join queries to return networks of related objects efficiently.
3. Inheritance: balance retrieval speed against query efficiency. In general, horizontal inheritance is the best choice as it speeds retrieval for child classes.
4. Object caching: optimize performance with a shared transactional cache.

Prefetching and caching "high activity" objects can speed access by three orders of magnitude.

5. Object navigation: Knit retrieved objects into an in-memory object cache. Navigate objects within the shared cache to avoid less efficient database joins.

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

For More Information
For further information on Persistence Software, please contact:
info@persistence.com

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

[Cattell94] Cattell, R., Object Data Management, Addison-Wesley, Menlo Park, California, 1994

[Gray93] Gray, J., & 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. & Jensen, R., Enabling The Integration of Object Applications With Relational Databases, ACM SIGMOD Proceedings, 1993

[Keller95] Keller, A. M., & Basu, J., A Predicate-based Caching Scheme for Client-Server Database Architectures, VLDB Journal, 1995

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

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

Previous

Persistence Software Inc.
1720 S. Amphlett Blvd. Suite 300, San Mateo, CA 94402
Tel: 650.372.3600 Fax: 650.341.8432 info@persistence.com