# Migrating Relational Data to an ODBMS: Strategies and Lessons From a Molecular Biology Experience

**Jon Oler and Gary Lindstrom**

Department of Computer Science
University of Utah
50 S. Central Campus Drive, Rm. 3190
Salt Lake City, Utah 84112-9205 USA
{oler, gary}@cs.utah.edu

**Terence Critchlow**

Lawrence Livermore National Laboratory
P.O. Box 808 L-560
Livermore, CA 94550
critchlow@llnl.gov

## Abstract

The growing maturity of ODBMS technology is causing many enterprises to consider migrating relational databases to ODBMS's. While data remapping is relatively straightforward in most cases, greater challenges lie in economically and non-invasively adapting legacy application software. We report on a genetics laboratory database migration experiment, which was facilitated by both organization of the relational data in object-like form and a C++ framework designed to insulate application code from relational artifacts. Although this experiment was largely successful, we discovered to our surprise that the framework failed to encapsulate three subtle aspects of the relational implementation, thereby "contaminating" application code. We analyze the underlying issues, and offer cautionary guidance to future migrators.

## 1. Introduction

Relational database (RDB) management systems are the dominant database technology in use today. Initially developed in the 1970's, RDB technology is mature, robust, flexible, and broadly applicable. However, in recent years traditional RDBMS's have come to be viewed as deficient in data representational power in comparison to modern application software, which is increasingly object-oriented. This RDB shortcoming is being addressed by extended relational systems (e.g., Postgres [SK91]) and middleware such as object oriented relational database gateway products (e.g. Persistence [KJA93]). Such RDBMS extensions have been spurred by competition from object-oriented database management systems (ODBMS's), which combine comprehensive database management functionality and full-fledged OO data modeling [ABDDMZ89].

Enterprises are understandably cautious in adopting new technology such as an ODBMS due to risks including lack of prior experience in effective ODBMS use, concerns for vendor stability, slow standardization, disruption of application software development, and fear of failure --- with associated loss of investment and an embarrassing retreat to prior technology. Hence a cost effective, reversible, risk mitigating migration strategy has great appeal. In fortunate cases, the increasing OO sophistication of the enterprise's application software may have steered the enterprise's relational data design to a *de facto* object-based organization. Indeed, the database architects may have been blessed with the foresight to encapsulate RDB representation details in an OO framework, delivering to applications an ODBMS-like view on the relational data. Not surprisingly, such frameworks are a great aid to migration, in that they embody a ready solution to the first problem one encounters: converting data from relational to object form.

We report our experience in employing such a framework as a migration vehicle. In many ways, the framework fulfilled our expectations as a migration aid, especially in terms of ease of data conversion. However, the thrust of this paper is on unforeseen semantic and pragmatic issues encountered in the migration, arising from subtle aspects of RDB technology "leaking" through the framework and "contaminating" our application software. After sketching our application setting, framework-based migration strategy, and lessons learned along the way, we conclude with a chart of *dysfunctions*, *diagnoses* and *remedies* which may be instructive to other database developers contemplating a similar migration path.

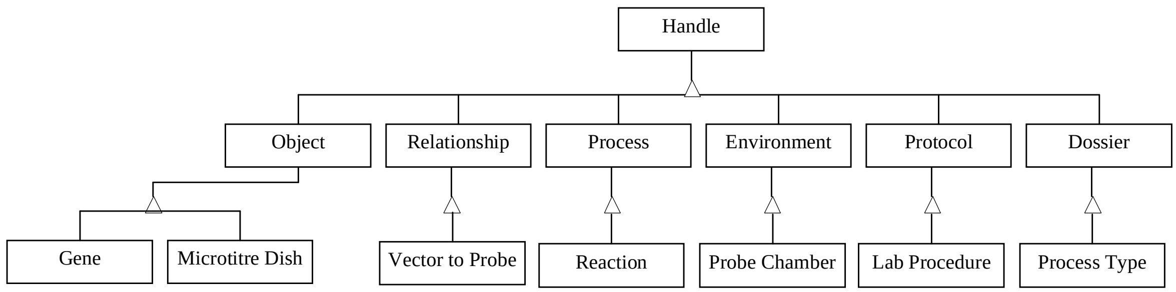


Figure 1: Simplified GORP class hierarchy

## 2. The Utah Center for Human Genome Research Database

Over the past six years, the Utah Center for Human Genome Research (UCHGR) has developed a comprehensive data model, database implementation and application suite for molecular biology laboratory information. The key characteristics of this database are: (i) an object-based meta data model comprising five fundamental concepts (*objects, relationships, processes, protocols*, and *environments*) in terms of which all concrete entities are expressed; (ii) an implementation of this model using a commercial RDBMS (Sybase), and (iii) a framework permitting application software to manipulate database contents as though they were a collection of persistent C++ objects, i.e., an ODBMS [SFCDML96].

Underlying this database design is a defensive posture with respect to the most vexing problem the UCHGR database implementers have faced over the years: frequent but unanticipatible schema evolution. Extensive use is made of meta information which guides access within a "hyper normalized" implementation by which object attributes are dispersed in individual tuples logically associated by object identifiers (OIDs) internal to the database. The result is an exceptionally supple data representation, permitting both (i) application data schema evolution by ordinary RDBMS transactions on meta data tables, and (ii) representational tolerance to data in many schema versions, both current and historical. These two features are crucial to the rapidly changing, yet archival, nature of molecular biological data.

These advantages notwithstanding, it rapidly became clear that the generality of this meta data representation, plus its lack of conventional OO structure, make it inappropriate for direct access by application programs. Hence a C++ framework was developed to act as an API to the database. This framework, called *GORP* (for *Generic Objects, Relationships* and *Processes*), presents to applications a reconstructive view of the database contents, consistent with the current concrete OO data schema expressed as C++ classes. Historical data, which is needed far less frequently, is either accessed through a lower level interface, or through GORP code specially written to do data evolution on demand. Like all class libraries worthy to be termed frameworks, GORP makes extensive use of abstract classes serving as interfaces to hidden implementation classes *completing* the framework. The production database currently comprises the GORP framework plus a completion library implementing GORP abstract methods in terms of SQL stored procedures.

## 3. Migrating the UCHGR Database to an ODBMS

Beyond its support for fluid schema evolution, the UCHGR database strategy is also defensive in that it anticipated eventual adoption of ODBMS technology, while protecting the developers from the pains of being an "early adopter" [GRS94]. The advent and commercial success of well-engineered ODBMS products, such as ObjectStore [LLOW91], indicate the time is ripe to seriously investigate migration to a true ODBMS.

The potential advantages of ODBMS's are well known, the most important to UCHGR being (i) direct storage of application-pertinent objects, eliminating the run-time overhead and the software maintenance cost of representation conversion code; (ii) faster overall performance, due to direct pointer navigation rather than multi-way joins (an unfortunate consequence of UCHGR's meta data representation); (iii) more seamless integration with C++ software development tools; (iv) more flexible data structuring representation possibilities, and (v) a true OO representation upon which application understanding can guide performance tuning and development of customized consistency and concurrency control policies.

In addition to assessing the practical merit of these potential advantages, the migration experiment provided a litmus test for GORP's representation encapsulation power. The experiment was approached through a novel migration strategy exploiting cooperating completions of the GORP framework. We describe this strategy in subsequent sections, along with some surprising pitfalls encountered, and lessons learned. Throughout, we focus on effective ODBMS exploitation by applications accessing current version (rather than historical) data. In the concluding section we offer some speculative remarks on how our migration strategy might be extended to support schema evolution.

## 4. Framework-Based Data Migration

The hypothesis underlying our experiment was simple: that GORP sufficiently encapsulated all RDB-specific aspects of the UCHGR database, such that no application software changes would be necessary if the RDBMS (Sybase) implementation were replaced by a genuine ODBMS (ObjectStore). To a first approximation, our hypothesis was validated.

| Mode | Uses |
|---|---|
| Read RDB / Write RDB | Preserves existing working RDB implementation; "benchmark" for alternate implementations |
| Read ODBMS / Write ODBMS | Complete conversion to ODBMS |
| Read RDB / Write ODBMS | Migrate data from RDB to ODBMS; populate ODBMS using real data in RDB |
| Read ODBMS / Write RDB | Migrate data from ODBMS to RDB |
| Read RDB / Write Both | Access data through RDB, maintain RDB / ODBMS consistency |
| Read ODBMS / Write Both | Access data through ODBMS, maintain RDB / ODBMS consistency |

Table 1: GORP framework: modes of operation

## 4.1 Dual Database Strategy

Our experiment began by building a dual-completion instantiation of the framework capable of operating in several modes. Table 1 summarizes how these modes might be used. We started with the *Read RDB / Write RDB* mode already implemented. Since this code is in production use, we made the baseline assumption that it is correct, and relied on it as a validation standard for other modes. This mode when will be used as a performance benchmark for the *read ODBMS / write ODBMS* mode when the latter is fully deployed..

Given the purely relational (*read RDB / write RDB mode*) completion of the framework, implementing the other modes described in Table 1 was, with the exception of several nuances described in the next section, very straightforward. The interface exported by the framework was left unchanged; in fact, the portion of the framework describing GORP database objects directly formed the basis of the ODBMS schema.

The *read RDB / write ODBMS* mode was applied on a wholesale basis to transfer data from the RDB to the ODBMS. The *read ODBMS / write both* mode allows both databases to be updated in tandem. Each database can then be read to verify that they returned the same result. In keeping with our risk mitigating strategy, this dual write mode constitutes a reassuring fallback to the fully robust relational version after deploying the ODBMS version. That is, if the ODBMS performance lags or other problems are encountered, it could be pulled from production and the RDB completion could be quickly redeployed since its associated database would be a "warm spare". The other two modes, *read ODBMS / write RDB* and *read RDB / write both*, were added for completeness but have not been extensively used.

## 4.2 Extent Sets

Since the class definitions already existed, the only work required to complete the ODBMS schema was to organize the database types into extent sets. Some ODBMS products automatically create and maintain class extents for every type in the database. While convenient, this service may be wasteful both in time and disk usage because some object types may only be accessed exclusively by navigation from another object rather than through key-based or exhaustive lookup on the extent set of the type. ObjectStore, in contrast, leaves the creation and maintenance of class extents entirely to the user.

Determining how to allocate class extents and index them for efficient access was quite challenging. We experimented with three design alternatives before identifying the one that best suited our needs. For background, we provide a simplified version of the GORP class hierarchy in Figure 1.

Under the first design we allocated class extents for every class in the database: an extent set representing all GORP database objects (the Handle extent), an extent set each comprising the Objects, Relationships, Processes, Environments, and Protocols, and finally an extent set for each of the leaves in the class hierarchy. As a result, each GORP database object was referenced from three extent sets. This led to further redundancy when indices defined on the base class were recreated in child classes. For example, an index was defined on the Handle extent with Handle's **id** attribute as key. However, this index was also duplicated on the Object extent as well as on many of the extents of classes that inherit from Object. Obviously, this approach resulted in a potentially large waste of storage.

Unfortunately, this was not the only problem we encountered with out initial approach: each time an object is created (or deleted) it had to be inserted (removed) from each of the three distinct extent sets, and more importantly, the indexes defined on each of the extents had to be updated. We discovered that this cost dominated the object's creation (deletion) time when we began to populate the database. Since the database loading process only required an extent over the Handle class with one simple index, we deferred building the other class extents and indexes until after loading the database. This dramatically reduced the time required to load the database.

The second design alternative was based on the

```
objects := db.all_objects                    // Retrieve all objects
relationships := db.all_relationships        // Retrieve all relationship objects
foreach object in objects
   object.get_relationships                  // Associate each object with its relationships
foreach relationship in relationships
   relationship.get_objects                  // Associate each relationship with related objects
```

Figure 2: Pseudo-code for retrieving all GORP objects and relationships

observation that the redundancy of explicitly creating class extents for each object type is unnecessary in principle. That is, the Process extent is simply the union of the extents of its child classes; likewise, the Handle extent is the union the Dossier, Object, Relationship, Process, Environment, and Protocol extents. Conversely, assuming the availability of run time type information, extents for any class in the GORP class hierarchy can be derived by filtering the Handle class extent.

However, neither of these options is a compelling choice. Consider the first; to derive extent sets, a class must be aware of all of its subclasses — generally bad object-oriented programming practice. For example, in order to execute a query over all processes, the Process class must either first build the Process extent from the extents of its subclasses, or ask each subclass to perform the query and combine the results. The second option is even more problematic because it is impossible to create indexes on the Handle extent based on attributes defined in subclasses of Handle. In addition, queries that deal with a single subclass will take longer to execute. For example, to find a Process with a given **id**, only Process instances need to be queried; however, if there is only one massive extent set, all instances participate in the query.

Note that completely eliminating redundancy is only one option; in fact, we utilize redundant sets to improve efficiency for common queries, while eliminating them where not needed. In our third and final design, the Handle extent is eliminated once the database is built. This does not dramatically effect query performance since interesting queries on Handle are rare and the direct subclasses of Handle are a fixed part of the framework unlikely to change. The reduction in insert/delete time as well as the space required by the database were significant. Additional savings were recognized by eliminating extent sets on classes that are accessed solely by relationships. The other extents were kept for performance reasons. Since retrieval/update is much more frequent than object creation/deletion our approach balances the extra space required by redundant extent sets and slower object creation/deletion against faster query execution speed.

## 4.3 Database Open and Close

Generic database functions such as opening and closing the database, and beginning, committing and aborting transactions were already expressed in the GORP framework and implemented in the RDB completion. The interfaces to these functions required no substantial changes to support the ODBMS completion. Their implementations, of course, were modified to perform the operations on the RDB, ODBMS, or both, depending upon the mode of operation. The lower-level database interface code of the GORP framework was extended to provide ODBMS-specific functionality such as object clustering, query facilities, and the creation/deletion of indexes. Since an API for some of these services has been defined in the ODMG standard, we wrapped the ObjectStore API within an ODMG-compliant interface where possible. This will ease porting to another ODBMS in the future, should the need arise. In all material respects, the GORP framework design proved to be adequate to encapsulate these DBMS-specific features, and hide them from applications.

The bulk of the work in implementing the ODBMS GORP completion involved modifying the query and update functions to access an ODBMS rather than an RDB. As explained earlier, the relational database employs a meta data representation which performs reconstructive querying in order to deliver concrete objects. In the RDB completion, this service is provided by SQL stored procedures, which apply meta data querying and component-wise accesses to reconstruct GORP objects for application presentation. However, because the ODBMS queries don't have to dynamically reconstruct objects and relationships between objects, the queries are much simpler than their relational counterparts. In contrast to the RDB completion where application data is stored in decomposed, meta data mediated form, the ODB completion has the luxury of retaining persistent objects in concrete application form.

## 4.4 Populating the Database

The ODBMS was populated by a transfer program using the GORP interface to traverse all objects in the database using the *read RDB* / *write ODBMS* mode. A simple traversal algorithm identifies each base object, and all data accessible from it. As each object is retrieved, a check is performed to see if an object with the same GORP OID has already been recorded. If the object was previously entered, it is ignored; otherwise, it is persistently allocated in the ObjectStore database. Once each object has been instantiated in the ODBMS, relationships between the objects can be established as shown in simplified form in Figure 2. Due to the simplicity of the underlying data model, less than 500 lines of C++ code were required to perform this migration.

# 5. Issues

We now examine three areas in which the migration did not proceed as smoothly as expected. In some areas, the causes can be attributed to inadequate foresight in the GORP framework design. However, since the relational completion of the GORP framework was implemented by experienced developers with extensive experience in both relational database development and object-oriented frameworks, we believe that their approach is typical of many projects exploiting an object-oriented interface to a relational database. In other areas, more fundamental semantic disparities emerge between RDBMSs and ODBMSs, and the application software architectures they commonly engender.

## 5.1 Issue 1: Object Mapping

Four basic operations on GORP database objects are exported to application programs by the GORP framework: *create*, *delete*, *retrieve*, and *update*. One of the most compelling ODBMS virtues is the elimination of object copying between application memory and the supporting database. Unfortunately, this virtue introduces subtle differences between the RDB and ODBMS semantics. This is a theme we will return to frequently throughout this paper.

There are currently a large number of commercial middleware products available to aid in mapping objects between an RDB and an object oriented programming language. These products vary widely in their approach to transactions, caching, mapping capabilities, scalability, database access (RDB-like vs. ODB-like), etc. Virtually everyone who has worked on the RDB completion of the GORP framework concurs that the necessity of writing custom code to map objects between the programming language and RDB representations is one of GORP's most unpleasant aspects. Mapping code occupies approximately 30 percent of the RDB version of the framework and must be maintained as the database schema changes. Employing middleware to do at least part of this job would provide a significant boost to productivity. Whether a project develops custom code, uses an RDB middleware product, or a true ODBMS, the mapping of objects from the database to application programs is an important issue. We now examine object mapping issues for each of the four basic GORP database operations.

### 5.1.1 Object Creation

The ODBMS completion of the framework invokes ObjectStore's rebindings of the C++ new and delete operators to create and destroy objects in persistent storage. Implementing this rebinding was facilitated by the framework design in which every persistent class has at least one static create method. This function originally allocated class instances on the application's transient heap, e.g., for containing the results of a database query. We easily modified these methods to accept a boolean flag indicating whether the object should be allocated on persistent storage instead.

Persistence-awareness also dictated that supporting classes allocate their internal data structures on persistent storage. Container classes (i.e. lists, bags) and a string class are examples of such classes. Of course, these implementation classes are part of the framework completion, rather than the GORP application interface, so their modification was completely transparent. To maintain their generality, these classes were augmented so they could allocate storage in either persistent or transient memory, depending on which creation method was invoked. These modifications were easily accomplished. For example, strings are implemented using an array of characters. The string constructor was modified to recognize whether the enclosing structure is allocated on persistent or transient memory (which is easily done using ObjectStore's os_database::of or os_segment::of operators), and to allocate the character array appropriately.

With some ODBMS products, such as the Java version of ObjectStore, object persistence may be determined at transaction commit time by identifying all objects reachable from a persistent root; with others, objects may be electively migrated from transient to persistent memory. With the C++ version of ObjectStore, persistent objects must be explicitly allocated in persistent memory when the object is created. Unfortunately, the relational completion of GORP allows applications to create a new GORP object transiently by invoking a class constructor, and subsequently confer persistence on the object by calling the object's save( ) operation. This is problematic because it is complex, costly, and perhaps ill-advised to move an object from transient to persistent memory in this version of ObjectStore. Indeed, this is particularly expensive if an entire graph of objects were created in transient memory, because a *deep* copy of this graph must occur when save( ) is invoked on a transient object.

Since all existing GORP applications know at object creation time whether an object will persist or not, we decided against implementing object migration from transient memory to the ODBMS in the GORP framework. Instead, we changed the semantics of the class constructors for GORP objects in the framework: persistent objects must be created exclusively with a call to the static create( ) method provided by each GORP class. Temporary objects may be created either through this create( ) method or through a class constructor. This allows temporary objects to still be efficiently allocated and deallocated on either the stack or the heap, and prevents applications from having to use DBMS-specific calls to persistent new. The difference between the two methods is demonstrated in Figure 3. This experience suggests that creation time determination of object transience or persistence should be mandatory in future GORP application development.

### 5.1.2 Object Deletion

Object deletion is encapsulated via destroy methods in GORP interface classes. Typically, applications (all initially written for the RDB completion of GORP) use the delete operator to free the transient memory occupied by GORP objects. The ODMG standard (as well as the ObjectStore API) overrides the delete operator to remove a persistent object from the database. We chose to reimplement the delete operator of each GORP class to be operative only if the object resides on the transient heap or the stack. If the object is persistently allocated, no action is taken. However, finessing the issue in this manner has the side effect that useless delete operations remain in the code, potentially confusing future maintenance programmers. It should be noted that this problem could be averted altogether by using an language and an ODBMS that supports persistent garbage collection.

<table>
<tr><th>RDB Implementation</th><th>ODB Implementation</th></tr>
<tr><td><pre>transaction.begin( );







DNA_fragment * frag =
   new DNA_fragment;

frag->operation1( );
frag->operation2( );
frag->save( );  // Inefficient with
                // ODBMS

transaction.commit( );</pre></td><td><pre>transaction.begin( );

bool persist = true;

// Make persistent when created
// with ODBMS
DNA_fragment * frag =
   DNA_fragment::create( persist );

frag->operation1( );
frag->operation2( );
frag->save( );  // No-op with
                //  ODBMS

transaction.commit( );</pre></td></tr>
</table>

Figure 3: Modifications to GORP for object creation

### 5.1.3 Object Retrieval/Update

We now consider object retrieval and updating, which exposed additional, more subtle differences between the semantics of these operations in the RDB and ODBMS completions of the GORP framework. Figure 4 gives pseudo-code for a typical interaction between an application and the GORP framework. Invoking get_unprocessed_microtitre_dishes( ) in the RDB version causes the GORP framework to issue an SQL query which returns a set of tuples representing unprocessed microtitre dishes. The GORP framework maps each microtitre dish tuple returned to a transiently allocated C++ microtitre dish object. The process( ) member function of class microtitre_dish modifies the microtitre dish as a C++ object. Note, however, that the persistent representation of the microtitre dish is not affected until the microtitre dish save( ) operation is invoked. The save( ) operation performs an SQL update synchronizing the transient C++ microtitre dish representation with its persistent representation in the RDB. Thus the GORP framework, and consequently the application software it supports, fundamentally embodies a copy in, copy out view of persistent data (the “client / server” viewpoint).

By contrast, the ODBMS completion of GORP handles the interaction of Figure 4 quite differently. The method invocation get_unprocessed_microtitre_dishes( ) queries the database as before, but no translation or explicit copy is required to convert the database representation of a microtitre dish to the C++ representation. Although a transient C++ replica of each unprocessed microtitre dish object is still created (by the ObjectStore ODBMS, in the application’s address space, operating as a database cache), this replication is transparent to the GORP framework and application code. In reality, modifications made to microtitre dish objects by invoking process( ) are made to the transient copies as before. However, unlike in the RDB GORP completion, the save( ) operation is an empty function in the ODBMS GORP completion. This is because the modifications made to the microtitre dish objects are automatically updated in the persistent store by the ODBMS when the surrounding transaction commits. Just as no code is required to translate the object from persistent memory to transient memory, no code is required to translate the object from transient memory to persistent memory.

## 5.2 Issue 2: Transactions and Swizzled References

As mentioned earlier, the GORP framework includes basic operations for starting, committing, and aborting transactions. However, the *copy in / modify / copy out* paradigm of the RDB version, plus ambivalence concerning the appropriateness of strict serialization of GORP applications as long running transactions, resulted in a *laissez faire* utilization of these features by UCHGR application programmers. Although database consistency issues were recognized clearly to be a concern, we encountered a different, rather subtle issue as a consequence. This issue concerns the lifetime and binding of object references, and their relationship to transaction semantics and duration. Although this problem manifests in various ways among ODBMS products, we believe them to be endemic to ODBMS technology.

Currently, and into the foreseeable future, real databases must be able to grow larger than the address space of the machines that access them. Unfortunately, this poses obstacles in fully integrating persistent data into a programming language, i.e., converting an object-oriented programming language into an ODBMS data manipulation language. If persistent objects are to be accessed in the same way as transient objects, applications must be able to access them through references native to the programming language, i.e. in *swizzled* form [EM92]. These references are bound to a block of memory into which the persistent object is mapped in the address space of the application process. However, if a process references a working set of persistent objects that exceed the size of its address space, some objects need to be removed to make way for new objects.

This requirement is benign if the evicted objects are not referenced again; however, it is difficult to determine at runtime which objects may be accessed again and which can be safely evicted. Hence it is often necessary to maintain valid swizzled references to persistent objects, even if they have been invalidated and evicted from an application’s address space. To address this requirement, the API of most ODBMS products provide a “long pointer” data structure constituting a universally valid persistent object reference. This provides a reliable way for an object to be recovered and remapped into a process’ address space in the event it has been evicted between references. Indeed, some ODBMS’s require persistent objects to be referenced only through unswizzled pointers. Unfortunately, this encapsulated access requires an extra level of indirection each time an object is accessed. Other ODBMS products, like ObjectStore, allow both swizzled and unswizzled references.

The ObjectStore ODBMS unmaps all persistent objects from an application’s address space at transaction commit time. As a result, all swizzled pointers in an application become invalid at that time. However, applications written presuming the RDB completion of GORP expect that pointers to persistent objects remain valid across transaction boundaries --- which is a reasonable assumption because the application is operating on transient copies rather than the persistent objects.

Figure 5 shows a gene object referenced in two different transactions. With the RDB completion, the framework caches a transient copy of the gene object until the application explicitly deallocates the object. Therefore, the gene reference in the second transaction is valid, but may not reflect changes to the database between the time the first transaction commits and the second transaction begins. With the ODB completion, the gene reference is invalidated when the first transaction commits. This leads to two error conditions when an attempt is made to access the gene in the second transaction: an invalid pointer dereference exception, or the worse possibility of an undetected erroneous

```
transaction.begin( );
   microtitre_dishes = GORP.get_unprocessed_microtitre_dishes( );
   foreach microtitre_dish in microtitre_dishes
       microtitre_dish.process( );                    //microtitre dish object is mutated
       microtitre_dish.save( );
transaction.commit( );
```

Figure 4: Example GORP framework operation

reference to an arbitrary location in a database segment subsequently mapped into that memory region.

There were several options available to us to overcome this problem. First, we could simulate the RDB version of GORP by making transient copies of each object read from the ODBMS. References to these transiently allocated objects would not be invalidated across transaction boundaries. This approach also had the advantage of overcoming the object mapping concerns previously identified as Issue 1. However, this proposal was quickly rejected as ODBMS heresy, as well as because the additional overhead was deemed unacceptable.

A second option was to adopt longer duration transactions whereby transactions do not commit until it is no longer necessary to reference any object. Because of the programming style used to implement the applications, this amounted to wrapping the entire application within a single transaction. This approach was rejected not only due to the resulting poor throughput but also because these applications may reference more data than can fit within the application's address space. When the address space is exhausted, ObjectStore aborts the transaction.

ObjectStore's default behavior of unmapping the virtual address space occupied by persistent objects at transaction commit time can be disabled. In this case, once an object is mapped to a virtual address, the mapping is retained until the application exits. Like the previous option, the application's address space can be exhausted. Potentially, applications could free portions of the address space at appropriate times through the ODBMS API, but this is not very elegant and is difficult to do in some GORP applications with dynamic transaction boundaries.

The final and most general approach is to use the unswizzled ("long") pointers supplied by ObjectStore. The primary disadvantage of this approach is that application source code must be modified to use long pointers to persistent objects referenced across transaction boundaries. Alternatively, the GORP framework could encapsulate the swizzled pointers in a GORPPointer class and modify all GORP functions to return references to this class rather than C++ pointers. This option was rejected because it introduces at least one additional level of indirection, and resulting overhead, for every pointer deference. The prospect of changing all applications to use GORPPointer references was daunting, as well.

The approach we ultimately adopted constitutes a hybrid of the last two approaches. Where appropriate, we extended transaction boundaries to encompass multiple object references. We also modified application source code to use unswizzled pointers for the remaining database references that cross transaction boundaries.

One of the lures of ODBMS technology is that the programming language becomes the database data manipulation language. The hallmark of such a DML is uniform ("seamless") access of persistent and transient data alike. Unfortunately, native programming language pointers are not sufficient to support all references to persistent objects. Today's programming languages were not designed to support transactions and concurrent access to shared data by multiple processes. Encapsulated access through unswizzled pointers is the price to pay for such features.

## 5.3 Issue 3: Object Identity

As mentioned briefly in an earlier section, the GORP framework defines unique object identifiers for all objects in the database which may be accessed by applications. In the original specification of the RDB version of the GORP framework, GORP OIDs were defined to be opaque data types with only one valid operation, a test for equality. Concretely, the RDB completion of GORP implements OIDs as integers. Unfortunately, in the rush to push applications into production, application developers were allowed to rely on the implementation of OIDs as integers. They took advantage of OID stability and external significance, e.g., a user could retrieve an OID, jot it down in a lab notebook, and later initiate a GORP object retrieval using it as a key.

In contrast, OIDs are a hidden implementation artifact in most ODBMS's. Hence in the ODBMS completion of GORP, it is not *ipso facto* appropriate to maintain a separate, GORP specific notion of OID. Had OIDs originally been implemented correctly as opaque data types, we could have easily changed the implementation of GORP OIDs to use the ODBMS OID. Although there is never any reason for an application to do anything but compare two OIDs for equality, application programmers have used the integer representation of OIDs in

| RDB Implementation | ODB Implementation |
|---|---|
| Gene * gene;<br><br>transaction_1.begin( );<br><br>  gene = Gene::get_gene( "X" );<br><br>transaction_1.commit();<br>// framework maintains transient copy of<br>// gene<br><br><br>transaction_2.begin();<br><br>  GeneBag * bag = gene->related();<br>  // gene is valid, but may not be current<br><br><br>transaction_2.commit(); | Gene * gene;<br><br>transaction_1.begin( );<br><br>  gene = Gene::get_gene( "X" );<br><br>transaction_1.commit();<br>// gene is now invalid<br><br><br><br>transaction_2.begin();<br><br>  GeneBag * bag = gene->related();<br>  // invalid pointer dereference<br>  // exception!<br><br>transaction_2.commit(); |

Figure 5: Differences in lifetime of object references

their code in so many ways that it is unfeasible to undo it. Consequently we are reluctantly maintaining both forms of OID in the ODBMS version of the framework for backwards compatibility with older applications.

The lesson offered here to framework designers is that the concept of OIDs and externally visible keys should be built into the framework. However, we feel strongly that these two concepts should be strictly separated from each other. That is, it is ill-advised to use OIDs to implement external keys or external keys to implement OIDs.

### 5.4 Final Remarks on Portability

When the RDB version of GORP was conceived, it was designed to accommodate an ODBMS port with relative ease. Ideally, no application code would need to be modified. For the most part, this has proved to be true. With the exception of adding transaction boundaries and some long pointers, we have not modified any application code. In short, encapsulating all data accesses to persistent objects within a framework like GORP has enabled us to port many applications with very little modification of source code between two very different DBMS products.

However, it would also be desirable to port the framework across ODBMS products with minimal effort. We believe that the GORP design sufficiently abstracted the notion of a relational database to make porting it from one RDBMS to another a fairly painless task requiring very few changes to the framework. It is important to note that one of the fundamental reasons this abstraction is successful is because of the acceptance of SQL as a standard interface to relational databases. Unfortunately, considerably more work would be required to port the current ODBMS version of GORP from ObjectStore to another ODBMS. Because a standard API does not currently exist for ODBMSs, all the queries within the GORP object completion would have to be converted from ObjectStore's proprietary query facilities to another proprietary API. We are optimistic current work by ODMG on OQL [C96] and ANSI/ISO on SQL3 [SQL3] may make ODBMS queries much more portable in the future.

## 6. Related work

The complexity of representing genomic data has been recognized by many other researchers [F91] [GRS94b]. MapBase, and its successor, LabBase, are genomic information systems developed at the Whitehead Institute similar in scope to that developed at the UCHGR [GRS94a] [RSG95] [G94]. However, both MapBase and LabBase were implemented using an ODBMS (ObjectStore) from the beginning. The fact that both the Whitehead Institute and the UCHGR have independently chosen to use an ODBMS is evidence of the difficulty in representing complex genomic data in a relational format.

The creators of Intermedia, a hypermedia framework developed at the Institute for Research in Information and Scholarship, considered porting their framework from an RDBMS to an experimental ODBMS [SZ87]. Although ODBMS technology was in its infancy at the time, the Intermedia researchers were mainly interested in overcoming the need to make transient copies of persistent objects stored in the RDBMS as well as the impedance mismatch between an object's representation in an RDBMS and an object-oriented programming language.

A comparison of performance for various pointer swizzling and non-swizzling schemes is described in [M92]. The Texas [SKW92] persistent store implemented pointer swizzling mechanisms very similar to that used by ObjectStore. The developers of Texas also recognized the problem of address space consumption and made some novel suggestions of how to deal with this problem by means other than invalidating all references to persistent objects at transaction boundaries [WK92].

As described above, the original version of the GORP framework had an ill-defined form of relaxed consistency due to the creation and manipulation of transient copies of database objects. This problem can be generalized in terms of a cache consistency problem [F96]. Efficient protocols for allowing appropriate degrees of consistency in a distributed computing environment with long running, interactive transactions remain an open research question.

## 7. Conclusions and Future Work

The lessons learned from our migration experience are summarized in Table 2, relying on a clinical metaphor. In the words of Waverly Root, "*Every virtue is accompanied by its inseparable vices*" [W66, p. 14]. For ODBMS's, the virtue is direct manipulation of persistent objects by application software. The inseparable vices are the semantic and operational burdens attending such direct manipulation. Perhaps it is too much to ask for an application framework to support deft and natural manipulation of objects in both *off line* (RDB) and *on line* (ODBMS) form. In any case, we offer the humble opinion that data representation issues --- the subject of much research in the academic database community --- are not the difficult problems. Instead, the core issues lie in areas long recognized to be among the most vexing of persistent data: object identity (copying vs. replication), transaction semantics (nature and lifetime of data ownership), and object naming (significance of OIDs and reference binding).

Despite the cautionary tone of this paper, we are pleased with the relative success of this experiment, and are encouraged to pursue several promising directions for future work. From a practical standpoint, UCHGR developers remain enthusiastic regarding the original goal of achieving a risk mitigating RDB to ODBMS migration strategy. Consequently a full-fledged port and performance comparison is underway. The project staff is particularly keen on exploiting the ODBMS version to explore relaxed concurrency control mechanisms appropriate for molecular biology applications, in which database modifications are mostly monotonic, and some degree of data inconsistency is part of daily life [BK91].

On a research level, we continue to be intrigued by the question of data evolution within this dual database environment. As remarked early on, among the many services provided by GORP framework is meta to concrete data representation conversion. The question thus arises: if the ODBMS port is a complete success, and the RDB is retired, how will data evolution be accommodated? We speculate that this dual database approach constitutes a "best of both worlds" solution: the ODBMS provides direct, fast, application-pertinent object access, and the RDB provides a generalized evolution-tolerant representation.

The long term solution thus may be a hybrid system, in which the ODBMS manages the live data, which is flushed to the RDB when data evolution is required. The GORP framework is then updated to present the new concrete data model, recompiled (along with applications, as necessary) and live data are loaded (or faulted in) as production resumes. The upshot is an ironic *denouement* of our plot: the RDB is now the cache.

| Dysfunction | Diagnosis | Remedy |
|---|---|---|
| Application code relies on low level access to database representations. | Lack of a framework providing encapsulated database access will require significant source code changes to each application. | Create a framework. |
| Creation of transient copies of database objects. | 1. Weak object identity semantics.<br>2. Uncontrolled or irregular updates to persistent objects.<br>3. Deallocating memory associated with an object is no longer the responsibility of the application.<br>4. Depending upon the ODBMS chosen, object creation model may be difficult to maintain. | In framework and application code, distinguish between object replication (same identity) and copying (new identity).<br>Perform copying only when coherence between copies is not expected or appropriate. |
| Reliance upon object copying to implement a relaxed consistency protocol. | 1. Retaining this will require many small transactions.<br>2. Might be desirable to take advantage of ODBMS transaction facilities to devise a sound and appropriate consistency protocol. | Determine appropriate consistency and concurrency control model for application domain.<br>Implement as generalized transaction concept. |
| Amount of data accessed by applications exceeds the address space. | Individual transactions too large.. | Use of unswizzled pointers may be required.<br>Restructure database into segments, offering finer grain control of swizzled pointer validity. |
| OIDs are manipulated by applications as a concrete data type. | OIDs may represent additional semantics, e.g., temporal creation order. | OIDs should be opaque data types.<br>May require re-engineering of application code. |
| Difficulty in porting the framework from one ODBMS to another. | Proprietary languages and inconsistent features. | Investigate industry joint or standard efforts such as OMDG's OQL or SQL3 |

Table 2: A "clinical" analysis of porting a relational database to an ODBMS

*Acknowledgments*: The authors are indebted to the designers and implementers of the UCHGR database and GORP framework: Peter Cartwright, David Fuhrman, Rob Sargent, Robert Mecklenburg, Tony Di Sera, and Chunwei Wang.

This research was funded in part by the National Institutes of Health grant *Utah Center for Human Genome Research*.

# 8. References

[ABDDMZ89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. *In Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 223-40, Kyoto, Japan, December, 1989.

[BK91] Naser S. Barghouti and Gail E. Kaiser, Concurrency Control in Advanced Database Applications, *Computing Surveys*, vol. 23, no. 3, September 1991.

[C96] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93*. 1996. Morgan Kaufmann Publishers, Inc.

[CMFRAD94] Judy Cushing, D. Maier, D. Feller, M. Rao, D. Abel, and M. DeVaney. Computational Proxies: Modeling Scientific Applications in Object Database, In *Proceedings of the Seventh International Working Conference on Scientific and Statistical Database Management*, p. 196-206, Portland, Oregon, September, 1994.

[EM92] J. Eliot and B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Transactions on Software Engineering*. Volume 18, Number 8, August, 1992. (pages 657-673).

[F91] Karen A. Frenkel. The Human Genome Project and Informatics. *Communications of the ACM*, Vol. 34, Number 11. (pages 41 - 51).

[F96] Michael J. Franklin. *Client Data Caching: A Foundation for High Performance Object Database Systems*. 1996. Kluwer Academic Publishers.

[G94] Nathan Goodman. An Object Oriented DBMS War Story: Developing a Genome Mapping Database in C++. In Kim, W., editor, *Modern Database Management: Object-Oriented and Multidatabase Technologies*. ACM Press.

[GRS94a] Nathan Goodman, Steve Rozen, Lincoln Stein. Building a Laboratory Information System around a C++-Based Object-Oriented DBMS. *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994.

[GRS94b] Nathan Goodman, Steve Rozen, Lincoln Stein. A Glimpse at the DBMS Challenges Posed by the Human Genome Project. Available via anonymous ftp from genome.wi.mit.edu as file /pub/papers/Y1994/challenges.ps.Z.

[KJA93] Arthur M. Keller, Richard Jensen, and Shailesh Agarwal. Persistence software: bridging object-oriented programming and relational databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington DC, 1993, pp. 523-528.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*. Volume 34, Number 10, October, 1991. (pages 50-63).

[ODI96] Object Design, Inc., 25 Burlington Mall Rd., Burlington, MA 01803-4194, USA. Manual Set for ObjectStore Release 4.0, March, 1996.

[RSG95] Steve Rozen, Lincoln Stein, and Nathan Goodman. LabBase: Managing Lab Data in a Large-Scale Genome-Mapping Project. *IEEE Engineering in Medicine and Biology*, Volume 14, Number 6, p. 702 - 709.

[RSG94] Steve Rozen, Lincoln Stein, and Nathan Goodman. Constructing a Domain-Specific DBMS using a Persistent Object System. *Sixth International Workshop on Persistent Object Systems*. Available via anonymous ftp from genome.wi.mit.edu as file /pub/papers/Y1994/labbase-design.ps.Z.

[SFCDML96] Rob Sargent, Dave Fuhrman, Terence Critchlow, Tony Di Sera, Robert Mecklenburg, Gary Lindstrom, and Peter Cartwright. The Design and Implementation of a Database for Human Genome Research. In *Proceedings of the Eighth International Conference on Scientific and Statistical Database Systems*, IEEE Computer Society Press, p. 220-225, Stockholm, Sweden, June, 1996.

[SK91] Michael Stonebraker and Greg Kemnitz. The Postgres Next Generation Database Management System. *Communications of the ACM*. Volume 34, Number 10, October, 1991. (pages 78-92)

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. *Proceedings of the Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, September, 1992.

[SZ87] Karen E. Smith and Stanley B. Zdonik. Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems. *OOPSLA '87*, Orlando, Florida, October, 1987.

[SQL3] ISO and SQL3 working draft, available via anonymous ftp from speckle.ncsl.nist.gov in directory /isowg3.

[W66] Waverly Root. *The Food of France*. Vintage Books, 1966.

[WK92] Paul R. Wilson and Sheetal V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. *Workshop on Object Orientation in Operating Systems*, p. 364-377, Paris, France, September, 1992.