Migrating Relational Data to an OODB: Strategies and Lessons From A Molecular Biology Experience

Jon Oler, Gary Lindstrom and Terence Critchlow

UUCS-97-001

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

February, 1997

Abstract

The growing maturity of OODB technology is causing many enterprises to consider migrating relational databases to OODBs. While data remapping is relatively straightforward, 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. To our surprise, the framework failed to encapsulate three subtle aspects of the relational implementation, thereby "contaminating" application code. We describe the underlying issues, and offer cautionary guidance to future migrators.

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 has begun to be 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 (OODBMS'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 OODBMS due to risks including lack of prior experience in effective OODB use, concerns for vendor stability, slow standardization, disruption of application software development, and fear of failure --- with 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 had the foresight to encapsulate RDB representation details in an OO framework, delivering to applications an OODB-like view on the relational data. Not surprisingly, such frameworks are a great

aid to migration, because they embody a ready solution to the first problem one encounters: data representation conversion.

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 list of symptoms, diagnoses and remedies that may be instructive to other database developers contemplating a similar migration path.

The Utah Center for Human Genome Research Database

The Utah Center for Human Genome Research (UCHGR) has developed over six years 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 it were a collection of persistent C++ objects, i.e., an OODB [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, guiding access within a "hyper normalized" implementation in which object attributes are dispersed in individual tuples associated by object identifiers (OIDs) internal to the database. The result is an exceptionally supple data representation, permitting both (i) logical 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. Both 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, made it inappropriate for direct access by application programs. Hence a C++ framework was developed, of the kind alluded to in the previous section. This framework, called *Gorp* (for *Genetic 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 (a *completion* of the framework). The production database currently comprises the Gorp framework plus a completion library implementing Gorp abstract classes in terms of SQL stored procedures.

Migrating the UCHGR Database to an OODB

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

The potential advantages of OODBs 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 determine customized consistency and concurrency control policies. Finally, the migration experiment would provide a litmus test for Gorp's representation encapsulation power.

The result was a novel migration strategy exploiting cooperating completions of the Gorp framework, which we describe in subsequent sections, along with some surprising pitfalls encountered, and lessons learned. Throughout, we will focus on effective OODB exploitation by applications accessing current version data. In our

concluding section we will make some speculative remarks on how our migration strategy might be extended to support schema evolution.

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 OODBMS (ObjectStore). To a first approximation, our hypothesis was validated.

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 was also used as a benchmark for validating the *read OODB / write OODB* mode.

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

Table 1 Gorp Framework Modes of Operation

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

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 OODB completion. Their implementations, of course, were modified to perform the operations on the RDB, OODB, or both, depending upon the mode of operation. The lower-level database interface code of the Gorp framework was extended to provide OODBMS-specific functionality such as the creation and deletion of class extent sets, object clustering, query facilities, and the creation/deletion of indexes. In all material respects, the Gorp framework design proved adequate to encapsulate these DBMS-specific features, and hide them from applications.

We next considered Gorp object allocation and deletion. The OODB completion invokes ObjectStore's rebindings of the C++ new and delete operators, which create and destroy objects in persistent storage. Every persistent class in the Gorp framework has at least one static create function allocating class instances on the application's transient heap, e.g., for containing the results of a database query. We modified these create functions to use the persistent versions of new to allocate objects on persistent storage.

Persistence-awareness also dictated that classes in the Gorp framework be adapted to allocate their internal data structures on persistent store. Examples of these are some container classes (lists, bags, etc.) and a string class. These implementation classes are, of course, part of the framework completion, rather than the Gorp application interface. These modifications were easily accomplished. For example, the string class was implemented using an array of characters. It had to be modified to determine whether a string object is persistently or transiently allocated (which is easily done using

```
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 1: Pseudo-code for retrieving all Gorp objects and relationships

ObjectStore's os_database::of or os_segment::of operators), so that each time it is resized, its character array is allocated on persistent or transient memory as appropriate.

Fortunately, object deletion is encapsulated via destroy methods in Gorp interface classes; consequently, applications do not directly invoke the C++ delete operator. Typically, applications (all initially written for the RDB completion of Gorp) use the delete operator to free the transient memory occupied by Gorp objects. This fortuitous encapsulation of memory recycling enabled us 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 was persistently allocated, no action is taken.

The bulk of the work in implementing the OODB Gorp completion involved modifying the query and update functions to access an OODB rather than an RDB. As explained earlier, the relational database employs a meta data representation which requires reconstructive querying to deliver concrete objects. In the RDB completion, this service is provided by SQL stored procedures, which laboriously apply meta data querying and component-wise accesses to reconstruct Gorp objects for application presentation. However, because the OODB queries directly access Gorp objects in their persistent C++ form , they are much simpler than these RDB stored procedures.

The OODB was populated by a transfer program using the Gorp interface to traverse all objects in the database using the *read RDB / write OODB* mode. As each object is retrieved, a check is performed to see if an object with the same Gorp OID has already been retrieved. If the object not is not yet present, it is persistently allocated in the ObjectStore database. Once each object has been instantiated in the OODB, relationships between the objects can be established, again by querying the RDB, as shown in simplified in Figure 1. Due to the simplicity of the underlying data model, less than 500 lines of C++ code to perform this migration.

Issues

We now examine three areas in which the migration did not proceed as smoothly as expected. In some cases, the causes can be attributed to inadequate foresight in Gorp framework design. In others, more fundamental semantic disparities emerge between RDBs and OODBs, and the application software architectures they commonly engender.

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*. As observed above, OODBMS's eliminate object copying between application memory and the supporting database; indeed, this constitutes one of the most compelling OODB virtues. Nevertheless, this benefit required adaptation of Gorp object creation and deletion operations. We now consider object retrieval and updating, which exposed additional, more subtle differences between the semantics of these operations in the RDB and OODB completions of the Gorp framework.

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.

Figure 2 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 OODB completion of Gorp handles the interaction of Figure 2

```
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 2: Example Gorp framework operation

quite differently. The method invocation get_unprocessed_microtitre_dishes( ) queries the database as before, but no mapping code 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 OODBMS, in the application's address space, operating as a database cache), this replication is transparent to the Gorp framework and application code. In effect, 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 OODB Gorp completion. This is because the modifications made to the microtitre dish objects are automatically updated in the persistent store by the OODB when the surrounding transaction commits. Just as no code is required to map the object from persistent memory to transient memory, no code is required to map the object from transient memory to persistent memory.

As described above, we redefined Gorp class delete operators to deallocate an object only if allocated in transient memory. This relieved us from the burden of modifying applications to only deallocate transient objects, and allowed us to use the same application code in either RDB or OODB mode. However, finessing this issue 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 garbage collection.

With some OODBMS products, such as the Java version of ObjectStore, object persistence may be determined at transaction commit if the object is reachable from a

Original Implementation	Modified Implementation
transaction.begin( );	transaction.begin( );

DNA_fragment * frag = new DNA_fragment;	// Make persistent when created
	// with OODB
	DNA_fragment * frag = DNA_fragment::create(

frag->operation1( );
frag->operation2( );
frag->save( ); // Inefficient with
// OOBMS
	persist);
	frag->operation1( );
	frag->operation2( );
	frag->save( ); //No-op with
	//OODB

transaction.commit( );	transaction.commit( );

Figure 3: Modifications to Gorp for object creation

persistent root; alternatively, objects may be explicitly migrated from transient to persistent memory. With the C++ version of ObjectStore, however, persistent objects must be explicitly allocated in persistent memory when the object is created. Unfortunately, the relational completion of Gorp allows applications to first transiently create a new Gorp object by invoking a class constructor, then later 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 ObjectStore, particularly if an entire graph of objects was 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 OODB 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( ) function provided by each Gorp class. Temporary objects may be created either through this create( ) function 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 indicates that creation time determination of object transience or persistence should be a mandatory consideration in future Gorp application development.

In contrast to the issues surrounding persistent object creation, mechanisms for removing objects from persistent store have the same semantics in both the RDB and OODB completions of the Gorp framework.

Issue 2: Transactions and Swizzled References

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. Although database consistency issues were recognized clearly to be a concern, we encountered a different, rather subtle issue as a consequence. This concerns the lifetime of object references, and their relationship to transaction semantics and duration.

Although this problem manifests in various ways among OODBMS products, we believe them to be endemic to OODB 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 making a programming language fully integrate persistent data, i.e., become an OODB data manipulation language. If persistent objects are to be accessed in the same way as transient objects, then 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. For this reason, OODBMS products implicitly manage the lifetime of swizzled references, according to certain protocols.

This requirement is benign if the evicted objects are not referenced again; however, it is difficult to determine at runtime which objects can still be accessed and which can be safely evicted. Hence it is often necessary to maintain valid swizzled references to persistent objects, even if they may have been invalidated and evicted from an application's address space. The API of most OODBMS 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 OODBMS's require persistent objects to be referenced only through long, unswizzled,

pointers --- but this typically entails encapsulated and slower access. Other OODBMS products, like ObjectStore, allow both swizzled and unswizzled references.

The ObjectStore OODBMS unmaps all persistent objects from an application's address space at transaction commit time. The result is that all swizzled pointers in an application become invalid once a transaction commits. However, our applications written with the RDB completion of Gorp expect that pointers to persistent objects remain valid across transaction boundaries --- which is reasonable because the application is operating on transient copies rather than the genuine, persistent objects. 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 OODB. References to these transiently allocated objects would not be invalidated across transaction boundaries. This option was quickly rejected as OODB heresy.

A second option was to have longer duration transactions where transactions are not committed until it is no longer necessary to reference an object within the application. This amounted to making each application a long duration transaction, which was rejected due to poor throughput. Long transactions are also undesirable because they may become so long that the amount of data accessed exceeds an application's address space. In this case ObjectStore would abort the transaction. In anticipation of this difficulty, ObjectStore supports the option of maintaining pointer validity across transactions. This is unrealistic for our project, however, since we have applications that will exhaust application address space unless regions are reclaimed from time to time.

The final and most general solution is to use the long pointers supplied by ObjectStore (class os_reference). The disadvantage here is that application source code must be modified to use long pointers for references to persistent objects that must be maintained across transaction boundaries. Another way to do this would be to have the Gorp framework encapsulate long pointers in a GorpPointer class and modify all Gorp functions to return GorpPointer references to objects rather than C++ pointers. This method was rejected because it introduces another level of indirection, and significant overhead, for every pointer dereference. The prospect of changing all applications to use GorpPointer references was daunting, as well.

The approach we adopted was a hybrid of these approaches. We modified the application code to use long pointers between transactions where necessary, but also extended transaction boundaries to encompass multiple object references.

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 specifications 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. 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 OODBMS's. Hence in the OODB completion of Gorp, it is not 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 OODB 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 their code in so many ways that it is infeasible to correct their code. Consequently we are reluctantly maintaining both forms of OID in the OODB version of the framework for backwards compatibility with older applications.

Final Remarks on Portability

When the RDB version of Gorp was conceived, it was designed to accommodate an OODB port OODB with relative ease. Ideally, no application code would need to be modified. For the most part, this has proved true. With the exception of adding transaction boundaries and the use of some long pointers as well as Gorp OIDs, 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 OODBMS products.

It would also be desirable to port the framework itself between different DBMS 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.

On the other hand, considerably more work would be required to port the current OODB version of Gorp from ObjectStore to another OODBMS. All the queries within the Gorp framework would have to be converted from ObjectStore's proprietary query facilities to another proprietary API. Current work by ODMG on OQL [C96] and ANSI/ISO on SQL3 [SQL3] may make such queries much more portable in the future.

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.

Symptom	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 representation of database objects.	1. Updates to persistent objects are synchronized with the database at different times. 2. Deallocating memory associated with an object is no longer the responsibility of the application. 3. Depending upon the OODBMS chosen, object creation model may be difficult to maintain.	In framework and application code, distinguish between object replication (same identity) and copying (new identity). 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. 2. Might be desirable to take advantage of OODBMS transaction facilities to clean up the consistency protocol.	Determine appropriate consistency and concurrency control model for application domain. Implement as generalized transaction concept.
Amount of data accessed by applications exceeds the address space.	Application code will have to be modified to use long pointers.	Careful design of database segments, within which swizzled pointers are stable
OID's are manipulated by applications as a concrete datatype.	1. New implementation of OIDs is likely desired. If so, application code must be modified. 2. OIDS may represent additional semantics, for example relative ordering	OIDs should be opaque data types.
Porting the framework from one OODBMS to another OODBMS.	Proprietary query languages make this difficult. ODBC doesn't work well with OODBMS's.	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 OODB

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. The port was never completed.

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 address this problem besides 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.

Conclusions and Future Work

We summarize the lessons learned from our migration experience 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 OODBs, 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* (OODB) 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 wherein the difficult problems lie. Instead, they 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 naming (OID significance and 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 concerning the original goal of achieving a risk mitigating RDB to OODB migration strategy. Consequently a full-fledged port and performance comparison is underway. The project staff is particularly keen on exploiting the OODB 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 OODB 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 OODB 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 OODB manages the live data, which is flushed to the RDB when data evolution is required. The Gorp is then updated to present the new concrete data model, recompiled (along with applications, as necessary) and live data is loaded (or faulted in) as production resumes. The upshot is an ironic *denouement* of our plot: the RDB is now the cache.

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.

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. (I really don't know if this needs to be referenced in the paper).

[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*, 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.