Migrating Relational Data to an ODBMS

Strategies and Lessons from a Molecular Biology Experience

Jon Oler and Gary Lindstrom
University of Utah

Terence Critchlow
Lawrence Livermore National Laboratory

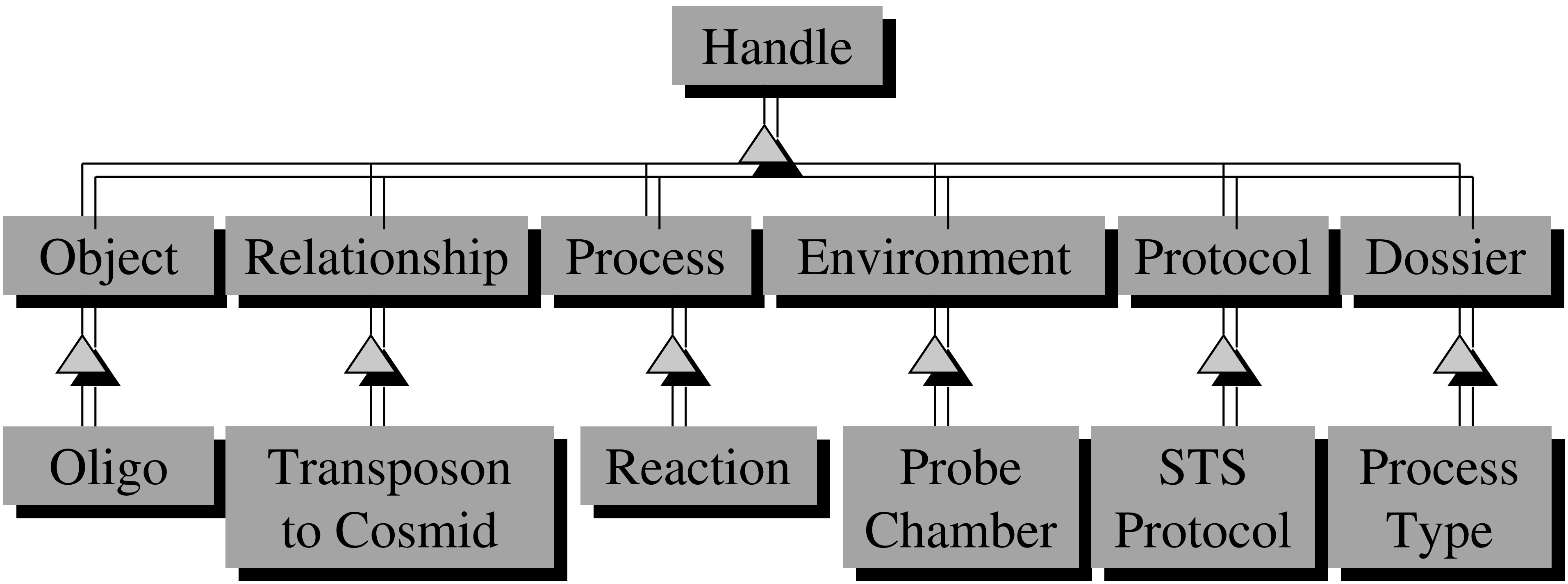
Utah Center for Human Genome Research

- Human Genome Project
- Massive international effort to sequence the human genome
- Laboratory Information Management System (LIMS) increasingly important as sequencing production levels grow.

A LIMS for a Genetic Research Lab

- 5 Basic Concepts
  - Object
  - Relationship
  - Process
  - Environment
  - Protocol
- Meta data
  - Dossier

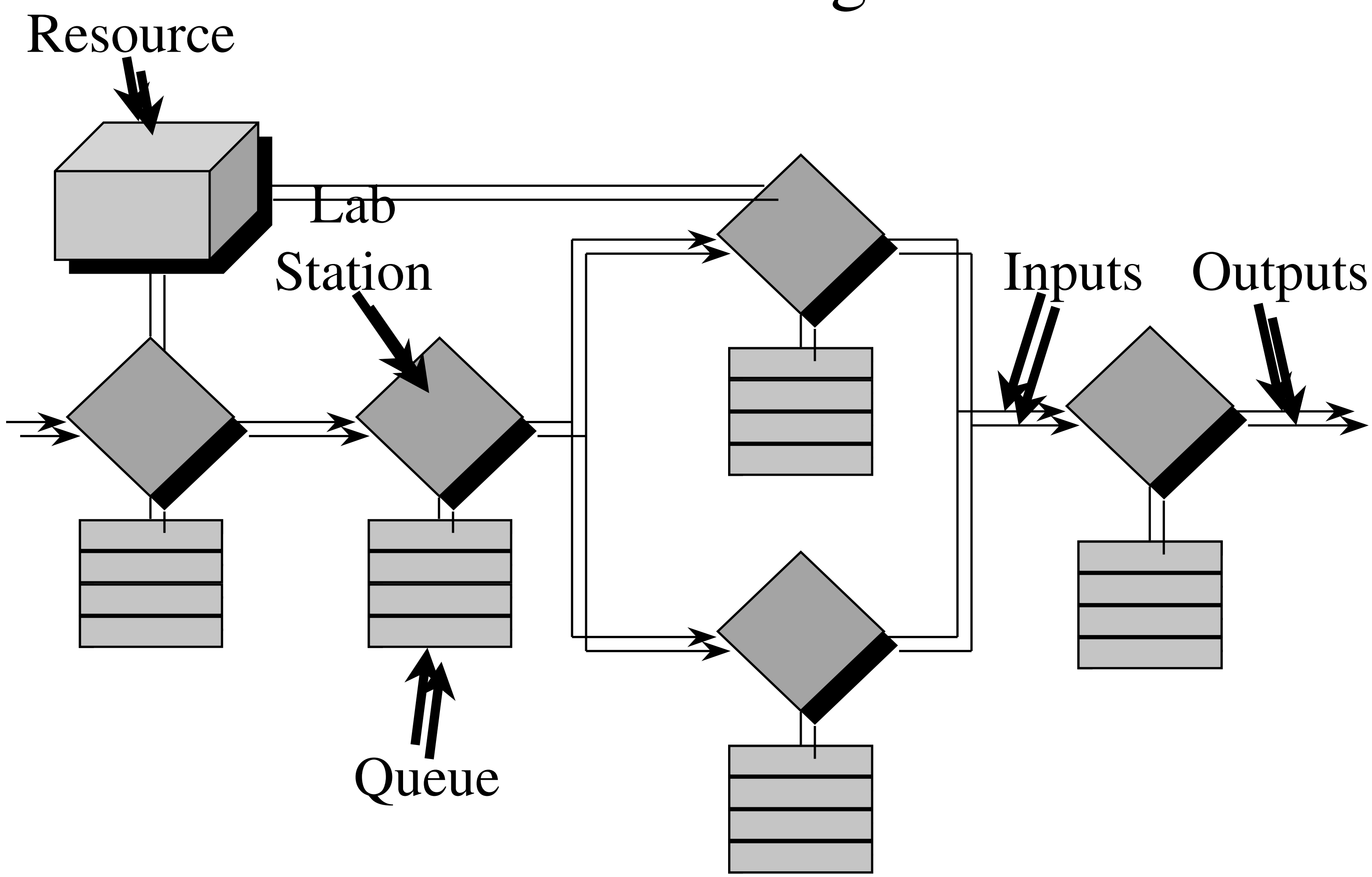
A LIMS for a Genetic Research Lab

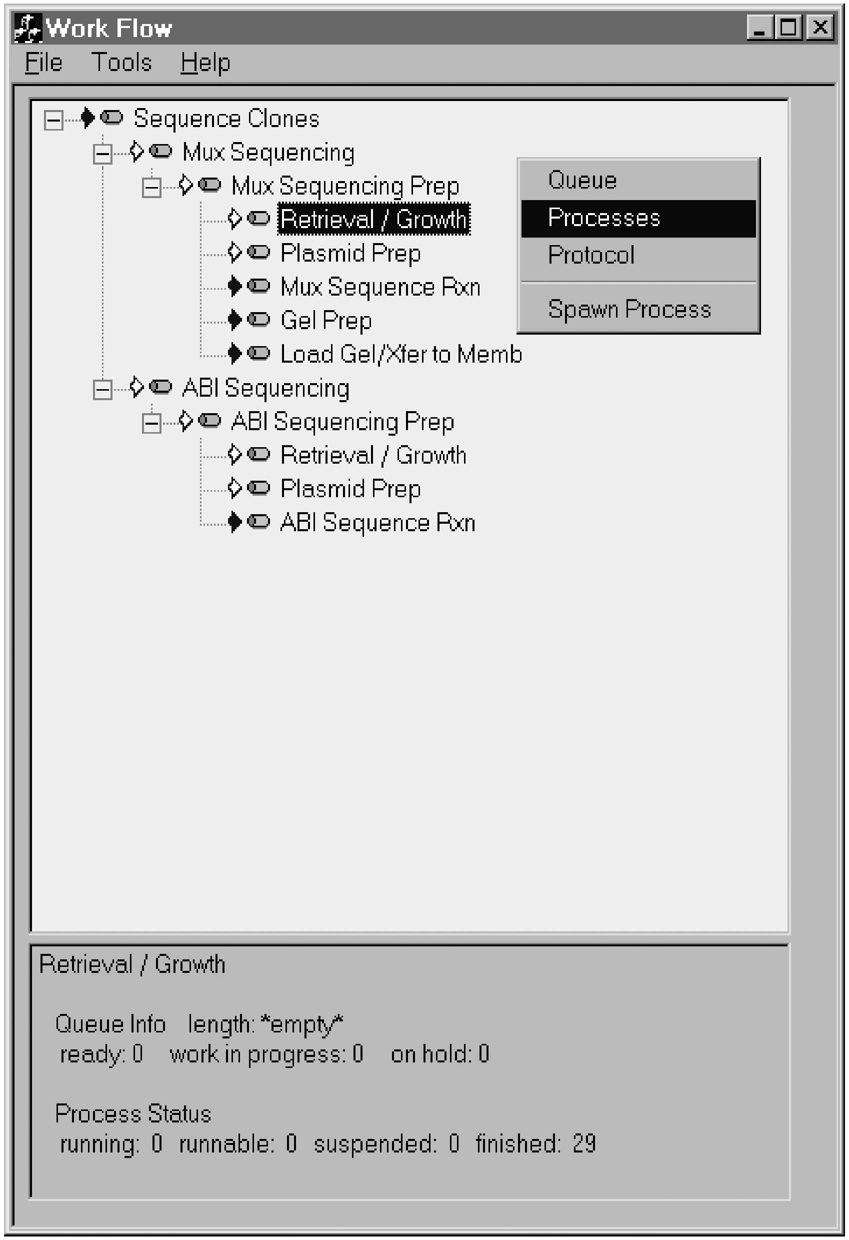


Workflow Management

- Objectives
  - Control lab workflow.
  - Discover bottlenecks.
  - Track the origin and process flow of any result obtained.
- Basic concepts
  - Lab Stations
  - Resources
  - Inputs/Outputs
- Routing rules are defined to route station outputs.
- A scheduler manages scarce resources.

Workflow Management





Workflow Application

- Lab stations are nested.
- The Retrieval/Growth station is selected.
- Possible actions
  - View/Edit the station queue.
  - View/Edit the currently running station processes.
  - View/Edit the station protocol.
  - Start new station process.

RDBMS Implementation

- "Hypernormalized" data representation
- Schema evolution possible through ordinary RDBMS transactions on meta data tables.
- Data access through C++ framework
- Problems
  - Object-relational mapping difficult to maintain.
  - Slow due to large number of joins, impedence mismatch, etc.

Goals of ODBMS Port

- Improved performance
- No object-relational mapping maintenance
- Easier to implement new complex data types in the future
- Modifications to applications should not be necessary

Implementation of ODBMS Port

- ObjectStore
  - Most of the issues apply to other ODBMSs as well.
- C++ framework types become database types.
- Migration via existing RDBMS version of framework
- For the most part, fairly straightforward

ODBMS Issues We Faced...

- Database organization / class extents
- Object mapping differences (RDBMS vs. ODBMS)
- Transactions and object references
- Object identity

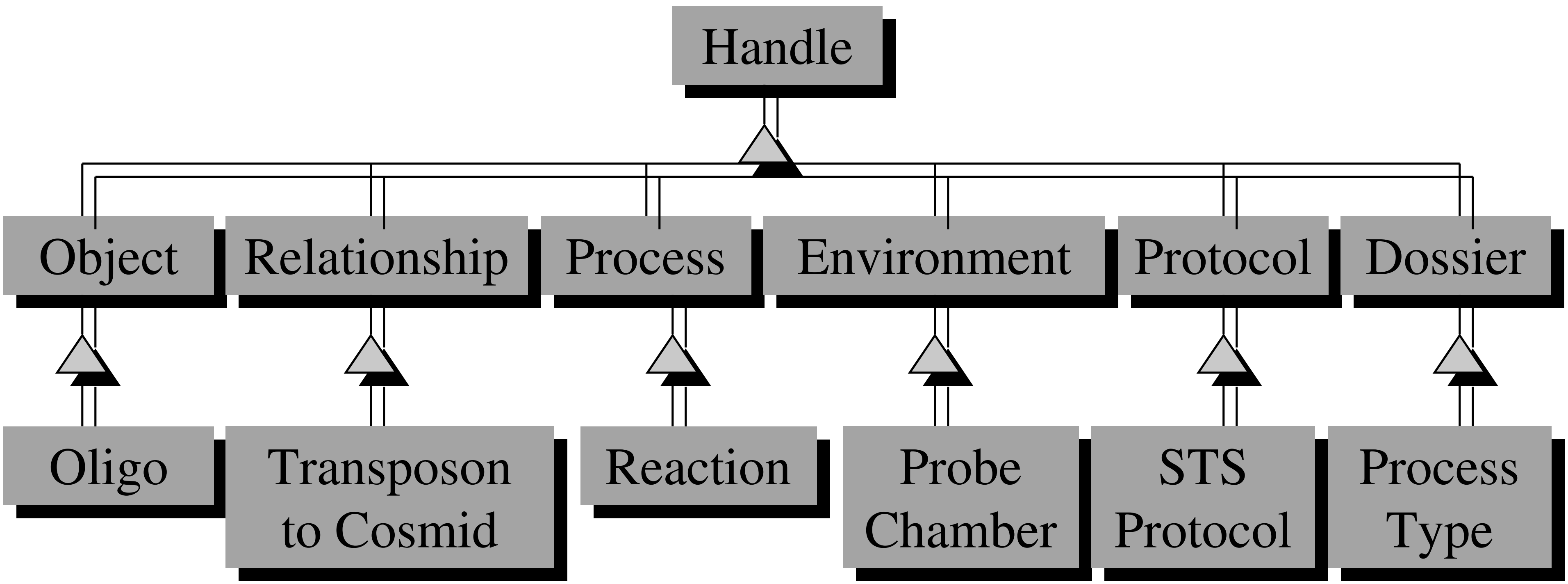
ODBMS Issues We Faced...

- Database organization / class extents
- Object mapping differences (RDBMS vs. ODBMS)
- Transactions and object references
- Object identity

ODBMS Class Extent Maintenance Alternatives

- The ODBMS maintains class extents for every type in the database.
  - Convenient, but could be overkill.
- The user must maintain class extents, no ODBMS support (ObjectStore).
  - More work for the user, but not a difficult task.
- Hybrid. The ODBMS maintains class extents for user-specified classes only.
  - Perhaps the best choice...

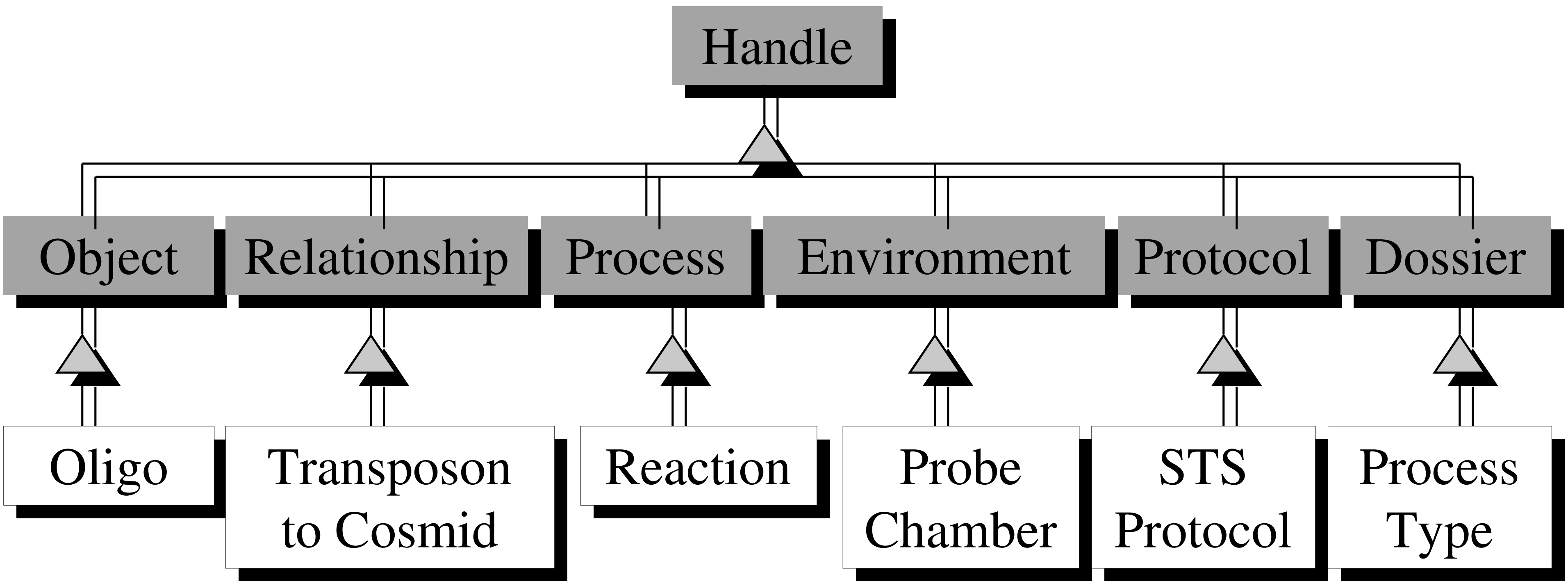
Every Class has an Extent



Every Class has an Extent

- Advantages
  - Fast associative retrieval possible at any point in the class hierarchy.
- Disadvantages
  - Not all classes need an extent (i.e. classes only reached via navigation or classes with no key).
  - More disk storage required for the extent and any duplicated indexes.
  - Slow object creation/deletion (don’t forget index maintenance!)

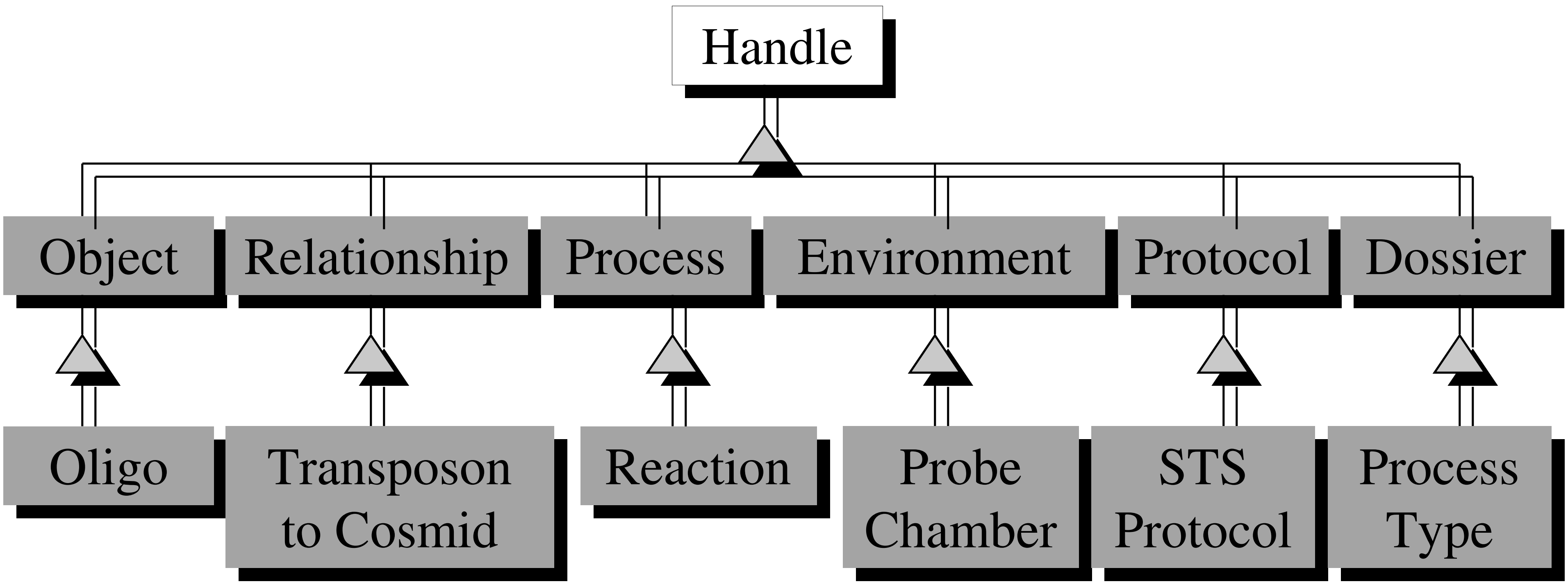
Extents on Leaf Classes Only



Extents on Leaf Classes Only

- Advantages
  - Fast object creation/deletion
    - Only 1 extent and index to maintain
  - Less disk storage
    - Base class derives its extent as the union of the extents of its subclasses.
- Disadvantages
  - Base class must know its subclasses.
  - Slow associative retrieval
    - Generate the extent, create an index, execute the query, OR
    - Have each subclass perform the query, and union the results.

Extent on Base Class Only



Extent on Base Class Only

- Advantages
  - Fast object creation/deletion
    - Only 1 extent and index to maintain
  - Less disk storage
    - Base class derives its extent as the union of the extents of its subclasses.
- Disadvantages
  - Can’t create indexes based on attributes introduces in a subclass.
  - Slow associative retrieval
    - Generate subclass extent, create index, execute query, OR
    - Execute query over base class extent where possible.

Hybrid Approach

- We allocated class extents where :
  - Associative lookup was required over instances of the class, AND
  - Speed of associative retrieval was more important than speed of create / delete.
- Disk usage is also an issue, but wasn't important to us.

ODBMS Issues We Faced...

- Database organization / class extents
- Object mapping differences (RDBMS vs. ODBMS)
- Transactions and object references
- Object identity

ODBMS vs. RDBMS Object Mapping Differences

- Object-relational mapping
  - Traditional client/server model (copy and transform)
  - > 30% of our code devoted to mapping between relational and object representation.
  - Maintenance nightmare
- ODBMS object mapping
  - Requires no mapping code!
  - Coherent replication model (similar to DSM)
  - Will applications really not need to be modified when we use an ODBMS?
  - Not quite…

Object Creation

RDB Implementation	**ODB Implementation**
`Transaction.begin();`	`transaction.begin();` `bool persist = true;`
`//Not persistent yet` **`DNA * frag =`** **`new DNA;`**	**`//Make persistent now`** **`DNA * frag =`** **`DNA::create( persist );`**

`frag->operation1();`
`frag->operation2();`
`//Persistent`
`frag->save();`
	`frag->operation1();`
	`frag->operation2();`
	`//No-op`
	`frag->save();`

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

Object Creation

- With RDBMS, objects are made persistent by calling save().
- With ObjectStore, objects are made persistent at object creation time.
  - Can specify how to cluster the object for performance reasons.
  - May not know if object should persist.
  - Persistence by reachability would help.
  - But still need a clustering mechanism.

Object Deletion

- Applications use the **delete** operator to free the memory occupied by objects.
- ODMG and ObjectStore override **delete.**
  - **delete** removes the object from the database.
  - ODBMS is reponsible for memory management of persistent objects.
  - RDBMS-based applications will delete objects from the database!!

Solution to Object Deletion Problem

- Remove the **delete** calls from all applications.
  - Applications no longer work with relational version of the framework.
- Override **delete** to do the right thing for certain classes.
  - This is what we chose.
  - **delete** now means so many things that it can confuse programmers.
- A language AND an ODBMS that supports garbage collection could help.

ODBMS Issues We Faced...

- Database organization / class extents
- Object mapping differences (RDBMS vs. ODBMS)
- Transactions and object references
- Object identity

Transactions

RDB Implementation

```
Gene * gene;

transaction.begin();

gene = Gene::find("X");

transaction.commit();

//OK, but gene may
//not be current
gene->display();
```

ODB Implementation

```
Gene * gene;

transaction.begin();

gene = Gene::find("X");

transaction.commit();

//Invalid pointer
//dereference exception!
gene->display();
```

Transactions and RDBMS Object Mapping

- We hand-coded the mapping of objects to relational tables.
  - Query returns tuple, tuple contents used to create object.
- Programmer is aware of transient and persistent copies of the object.
- Objects can be referenced outside of transaction, but it may not be current.

Transactions and ODBMS Object Mapping

- ODBMS handles mapping objects from disk to transient memory.
- Programmer can make no distinction between the transient and persistent copies of the object.
- Objects cannot be referenced outside of transaction.
- As we shall see, interesting things happen when database meets programming language.

Cross Transaction References

RDB Implementation	**ODB Implementation**
`Gene * gene;`	`Gene * gene;`
`transaction.begin();`	`transaction_1.begin();`
`gene = Gene::find("X");`	`gene = Gene::find("X");`
`transaction.commit();`	`transaction_1.commit();`
	`transaction_2.begin();`

`//OK, but gene may`
`//not be current`
`gene->display();`
	`//gene is still invalid!`
	`gene->display();`

	`transaction_2.commit();`

Cross Transaction References

- Programming language references are not sufficient for database references.
- With ObjectStore's memory-mapped architecture, the main problem is a database may be much larger than virtual memory.
- Some ODBMS's (including ObjectStore) free the address space occupied by persistent objects at transaction commit.
- Somewhat defeats the ODBMS goal of integrating the database and the programming language.

But Cross Transaction References are Useful

- The solution
  - Introduce an extra level of indirection
  - “Smart pointer” class
  - The ODMG d_Ref parameterized class
- The drawbacks
  - Extra level of indirection for every pointer dereference is slower.
  - Applications not coded with an ODBMS in mind must be modified to use smart pointers rather than regular pointers.

Cross Transaction References

RDB Implementation	ODB Implementation
`Gene * gene;`	**`os_Reference< Gene > gene;`**
`transaction.begin();`	`transaction_1.begin();`
`gene = Gene::find("X");`	`gene = Gene::find("X");`
`transaction.commit();`	`transaction_1.commit();`
	`transaction_2.begin();`

`//OK, but gene may`
`//not be current`
`gene->display();`
	`//If gene is valid,`
	`//it is current.`
	`gene->display();`

	`transaction_2.commit();`

Always Retrieving the Most Current Version Can Be Slow

- Inside a transaction, the most current version of an object is always used.
- The most current version is not always required.
- To access an object outside a transaction, an explicit copy of the object must be done (a la object-relational mapping).
- We actually do this for some of our read-only GUI applications.

Accessing ODBMS Objects Outside a Transaction

- Some ODBMS's allow objects to be retained and used outside of a transaction.
  - C++ ObjectStore does not have this feature.
- Many tricky issues here
  - When to release these objects?
  - Should they be made current when a new transaction begins?
  - What if they have been updated?
- Things do indeed get interesting when database meets programming language!

Effect of ODBMS Transactions on our Applications

- Added transactions where necesssary.
- Added “smart pointers” for cross transaction references.
- Explicitly copy objects for reference outside of a transaction where performance was critical and the most current version was not required.
- These kinds of problems are difficult to foresee when developing a framework with an RDBMS in mind.

ODBMS Issues We Faced...

- Database Organization / Class Extents
- Object Mapping Differences (RDBMS vs. ODBMS)
- Transactions and Object References
- Object Identity

Object Identifiers

- OIDs were in the relational design from the beginning.
- C++ framework exposed the implementation of OIDs as integers.
- Applications abused this in many ways.
  - OIDs used as keys.
  - Dependence upon ordering of OIDs.
- ODBMSs provide their own object identity mechanism.
- Too difficult to remove old OIDs from existing applications.
- Lesson:
  - Implement OIDs as opaque data types. We only need an equality test.
  - Don’t use OIDs as keys or keys as OIDs.

Developing Applications that can be Easily Ported Between RDBMS and ODBMS

- Encapsulating database access within a framework is a good idea!
- Specify that an object is to be persistent at object creation time.
- Arrange for the framework to manage the memory occupied by persistent objects.
- Encapsulate all pointers to persistent objects in a "smart pointer" class.
- Always request that the framework return a copy of an object to be accessed outside of a transaction, if necessary.
- Encapsulate the implementation of OIDs within the framework.

Summary

- The port was largely a success.
- Initial performance experiments are very promising.
- The elimination of object mapping code is a great boost to productivity.
- Applications needed to be modified more than we anticipated.
  - Modifying many applications took a lot of time.
  - However, the modifications are relatively easy to understand.

Migrating Relational Data to an ODBMS

Strategies and Lessons from a Molecular Biology Experience

Jon Oler and Gary Lindstrom

University of Utah

Terence Critchlow

Lawrence Livermore National Laboratory