Advanced Techniques for Object Warehousing

Matthew Wakeling
FlyMine Group, Department of Genetics
University of Cambridge
matthew@flymine.org
Introduction

Most of what you will see in this talk is implemented by the FlyMine group, in the InterMine system (http://www.intermine.org/)

This talk concentrates on the techniques that are used by the InterMine system for O/R mapping, and data integration and warehousing.

- Hopefully these techniques can be applied to other applications.
Introduction

Background

- The FlyMine project is a large-scale data warehouse for biological data about the fruit-fly and the mosquito.
  - It incorporates data from many published data sets, in many formats, and presents this data to researchers in an object-oriented manner.
  - It allows researchers to run complex arbitrary time-consuming cross-reference queries.
- InterMine is the generic model-independent data integration and warehousing system that we wrote to achieve this.
Introduction

Assumptions

- You do not have a tyrannical DBA breathing down your neck and changing the database schema under your feet.

- You want to build a database from data sources and present it to the data users as a read-only database.

- You wish to use data that does not fit into current O/R mapping systems

- You may wish to make use of extra storage space to improve performance.
Introduction

Coverage

- Multiple Inheritance in an O/R mapping tool
- Objectstores, Queries and Results
- Writing, and transaction processing
- Inserting complex data
  - Models and OWL
  - Data transformations, and why a simple pipe won't work
  - Translating data into a useful format and model
  - Integrating multiple conflicting data sources
Multiple Inheritance and CGLib

- Motivation: I am writing a database to store biological data, and it doesn't fit into a model:
Multiple Inheritance and CGLib

- Instances of normal classes in the model are easy:
Multiple Inheritance and CGLib

- Instances of subclasses in the model are also easy:
Multiple Inheritance and CGLib

More difficult: objects that are only implementations of an interface in the model:
Multiple Inheritance and CGLib

- An object that extends and implements various unrelated classes and interfaces from the model:
Multiple Inheritance and CGLib

- An object that implements “DB Object” (so it will be stored) but contains data independent of the model:
Multiple Inheritance and CGLib

What does the model describe?

- A searchable class hierarchy
- A collection of classes – all objects in the database must inherit from at least one of them
- A collection of fields attached to a given class that are definitely present in an object of that class, and are therefore searchable

The model is not:

- A limit to the classes that can be stored
- A limit to the type of data that can be stored
Multiple Inheritance and CGLib

- The model provides descriptions of Java classes, containing fields (attributes, references, and collections).
  - The Java classes are generated automatically from the model description.

- A field is represented in a Class by the presence of a getter and a setter.
  - The database objects are Beans.
  - Therefore, an Interface can effectively “have a field”.
Multiple Inheritance and CGLib

- We use CGLib to generate objects of a “composed” class:

  ```java
  Object o = net.sf.cglib.proxy.Enhancer.create(Class superclass, Class interfaces[], Callback callback);
  ```

- This creates an object that extends `superClass` and implements all `interfaces`, and passes method calls to the `callback`.

- That callback (`MethodInterceptor`) can then simulate the dynamically-created object.

  ```java
  public Object net.sf.cglib.proxy.MethodInterceptor.intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws java.lang.Throwable
  ```
Multiple Inheritance and CGLib

- CGLib combines capabilities from existing classes and interfaces into a new dynamic class. Instances of this class:
  - implement and extend the classes and interfaces from which the class was generated.
  - contain fields and methods that are present in the original classes (albeit simulated).

- CGLib cannot add new fields or methods that do not already exist in a compiled class.

- Not all the classes used need be in the model.
  - Data accessible using Bean methods will be stored.
Database Schemas

- Some possible strategies for mapping a model onto a schema:
- Map all classes onto one table
  - That table must have a “classname” field
  - The table must have a column for every field in the entire model.
    - Really bad data density for interesting models.
    - Possibility for similarly-named fields with differing types to conflict
  - Only fields described in the model may be stored.
Database Schemas

- Map each class onto an individual table, and store each object in only one table
  - Rather inflexible – cannot store dynamic class objects.
  - Very cheap writes.
  - Queries on superclasses need a SQL UNION, making such reads slow and complicated.
  - Suitable (and fast) for models with no hierarchy.
Database Schemas

- Map every class onto an individual table, containing fields defined by the class and not the superclass, then store objects in every table they implement or extend.
  - Expensive write.
  - Expensive read – must use an SQL join (probably more expensive than a UNION).
  - Only fields described in the model may be stored.
Database Schemas

- Map every class onto an individual table, containing all fields defined by the class and superclasses and interfaces, plus an OBJECT column, then store objects in every table they implement or extend.
  - OBJECT field is an XML-encoded representation of the object stored. This allows full flexibility.
  - Cheap & simple reads, and very easy to query.
  - Expensive write.
  - Bad data density, but that does not affect read performance.
Database Schemas

- A refinement to the last mapping:
  - If you have a huge hierarchical model, there may be many tables, and the SQL server may not perform well.
  - You may wish to map all of the tables corresponding to the subclasses of a particular class onto the same table.
  - This new table would need a “classname” field, and there would be the danger of many columns or conflicting columns.
  - However, it may give you a significant performance improvement.
Database Object Members

- Objects stored in the database have named fields, which are attributes, references, and collections:
  - Attributes are database primitives (numbers, dates, strings, boolean).
  - References are to other database objects.
  - Collections are of database objects.

- References and collections should be proxied.
  - The references and collections are loaded from the database when they are accessed.
Database Object Members

- References are of two types:
  - 1 to 1: This is represented by a field on both sides containing the ID of the other object. These must be kept consistent.
  - Many to 1: This is represented by an ID field.

- Collections also have two types:
  - 1 to many: An ID field in the remote object.
  - Many to many: For this relationship, an indirection table is used, which contains two columns, which are IDs of objects on either side. Each row represents a relationship.
The ObjectStore Interface

- A read-only interface to an object database.
- Takes queries and returns results.
- Provides object lookup by object ID (a primary key attribute of the DB Object class).
- There can be different implementations:
  - ObjectStoreInterMineImpl (SQL O/R mapping)
  - ObjectStoreClient (webservice client)
  - ObjectStorePassThruImpl (subclassed for easy interception of requests)
The ObjectStore Interface

- Caches:
  - Performing caching on queries is complex and probably unnecessary
    - It is hard to write clever equals() and hashcode() methods for a Query, so a hash table cache would be tricky. Besides, such methods could use sufficient CPU resources to counteract any benefit.
    - Caching can be provided elsewhere (more later).
  - A cache on object lookup by ID is sensible
    - IDs are just numbers, so hash tables work.
    - Assumes IDs are unique database-wide
Queries and Results

- Queries are a description of results, given the state of the database.
  - “SELECT DBOBJECT FROM DBOBJECT” describes the results containing the whole database.
  - “SELECT Company.name FROM Company WHERE Company.establishedYear = 1996” describes the results containing all the names of companies established in 1996.
  - Note that results can contain database objects, database primitives, constants, and results of functions and aggregate functions (SUM, etc.).
Queries and Results

- A Results object is a List of rows
  - The list has an order (either imposed by the query or defaulted), and elements can be accessed by row number.

- A “row” in a Results object can have multiple columns.
  - “SELECT Company, Department FROM Company, Department WHERE Company.departments CONTAINS Department” describes results containing two (database object) columns.
  - A row can therefore be a list of columns.
Queries and Results

- A “row” in a Results object can have just one column.
  - It may be more convenient to have a List containing the objects in that column, rather than a List of singleton Lists of those objects.
- Therefore, both types of Results object should be provided.
- Results are loaded on-demand.
  - A single-column Results object is ideal for use as a Collection proxy.
Queries and Results

- Results objects are constructed with a Query and an ObjectStore. Creating the Results does not actually access the database – that happens when the Results object is used.

- When the Results object is accessed, it fetches rows in batches from the database.
  - The batches are stored in a cache.
  - A prefetch manager spots regular access patterns and prefetches batches using a pool of helper threads, over a separate DB connection.
Queries and Results

- **SQL query generation:**
  - If we use the last database schema, SQL generation is very easy – the object query maps straight onto an SQL query.
  - Where in the object query you SELECT a database object (rather than a primitive), in the SQL query SELECT the table's OBJECT field, containing the XML serialised object.
  - Results parsing is also easy – just transfer between a multi-column SQL result to a List of values, converting from XML to object.
Queries and Results

- Warning: if you fetch results in batches from an SQL database using LIMIT and OFFSET.
  - The SQL server uses a query optimiser, which attempts to find the best algorithm for the query.
  - Different algorithms will result in the rows being returned in a different order.
  - Different LIMIT and OFFSET values may result in the optimiser choosing different algorithms.
  - Therefore, you MUST impose an order on the results by using ORDER BY in the SQL query.
Queries and Results

- Another warning: the Results object holds a reference to the Query object, and uses it every time it fetches a batch.
  - Do not alter the Query object, and expect the Results object to keep working!
  - In fact, if you want a cache in the SQL generator (useful for performance), then never change a Query once it has been used in any way by the ObjectStore.
  - Alternatively, write an ObjectStorePassThruImpl subclass that makes fiddling safe. (ObjectStoreSafeImpl)
The ObjectStoreWriter Interface

- This interface provides the means by which the contents of an ObjectStore are changed.
- It is a subinterface of ObjectStore.
- An ObjectStore is optimised for read-only access. It generally will cope with multiple simultaneous threads making requests, and will use a pool of connections to the database (SQL or webservice) server.
- The ObjectStoreWriter handles transactions, so it only uses one connection at a time.
The ObjectStoreWriter Interface

- An ObjectStoreWriter is constructed from an ObjectStore. When the writer is constructed, it takes a connection from the ObjectStore's pool, and uses it exclusively until closed.

- All write operations rely on the object's ID.

- Deleting an object:
  - Remove all rows that have the given ID, from all tables that are relevant to the class of the object.
  - NOTE: The given object must implement all the classes of the version in the database.
  - Why not read the DB version? Bad performance!
The ObjectStoreWriter Interface

- Storing an object:
  - Firstly, if the object or any object in its references and collections has a null ID, it must be given a unique ID.
  - Any previous version of an object with the same ID in the database must be deleted.
  - Rows must be written to all relevant tables.
  - No other object in the database is altered, so 1 to 1 relationships may be left inconsistent. 1 to many relationships are ignored, and many to many relationships are combined.
The ObjectStoreWriter Interface

- Writes consist of multiple actions, so a write performed outside a transaction must be done inside a temporary transaction.
- During a transaction, the ObjectStore is oblivious to changes made in the ObjectStoreWriter.
- When the transaction is committed or aborted, the ObjectStore and every ObjectStoreWriter connected to it must have their caches flushed.
Loading Data
Motivation

- The next section will describe techniques used for a particular type of data warehousing.

- The purpose of the FlyMine project is to bring together data from lots of published data sets into one database, so that queries can be run that cross-reference the data.

  ➢ So, we need to generate a model capable of representing all the data from the different sources, and load all the data into the database.
Loading Data
Creating a Model – OWL

- OWL is a vocabulary for RDF:
  - http://www.w3.org/RDF/
  - http://www.w3.org/OWL/

- A declaration in RDF consists of a “triple” of the form `subject property object`.

- Each of these three is a URI, although shortcuts can be declared.

- Probably the easiest way to describe the relevance of OWL for object models is to show you some.
Loading Data
Creating a Model – OWL

@prefix : <http://www.blarg.com/mymodel#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

Subject Property Object

:Person a owl:Class.

Subject Three Properties

:Person__name a owl:DatatypeProperty;
  rdfs:domain :Person;
  rdfs:range xsd:String.

:Bloke a owl:Class;
  rdfs:subClassOf :Person.
Loading Data
Creating a Model – OWL

- So OWL can be used to describe a model.
- OWL can also be used to describe how two models relate to each other.

  ➢ This can then be used to translate data from one model (A) to another (B).

```
A:Person a owl:Class.
A:Person__name a owl:DatatypeProperty;
  rdfs:domain A:Person;
  rdfs:range xsd:string.
B:Human a owl:Class;
  owl:equivalentClass A:Person.
B:Human__nom a owl:DatatypeProperty;
  rdfs:domain B:Person;
  rdfs:range xsd:string;
  owl:equivalentProperty A:Person__name.
```
Loading Data

Creating a Model – OWL

- Also, it allows for merging multiple models to create a model capable of describing all the data from all the models.

```
modelA:Person a owl:Class.
modelA:Bloke a owl:Class;
  rdfs:subClassOf modelA:Person.
modelB:Human a owl:Class;
  owl:equivalentClass modelA:Person.
```

- **This produces a class in modelB called “Bloke”, which is a subclass of “Human” - unless you do:**

```
modelB:HumanMale a owl:Class;
  rdfs:subClassOf modelB:Human;
  owl:equivalentClass modelA:Bloke.
```
Loading Data
Creating a Model – OWL

- OWL allows many other neat tricks, such as restricted subclasses:

  ```
  B:HumanMale a owl:Class;
  rdfs:subClassOf A:Person;
  rdfs:subClassOf
    [ a owl:Restriction;
      owl:onProperty A:Person__sex;
      owl:hasValue “male”
    ];
  ```

- The rest of OWL is beyond the scope of this talk.

- RTFM. Very carefully.
Loading Data

Data Transformations (no simple pipe)

- In order to integrate data from widely varying sources, some quite complex transformations need to be performed.
  - Some of these transformations involve how objects relate to other objects.
  - Therefore the transformation process may need to search in the data it is transforming, in order to link up the objects together in the destination database.
  - At FlyMine, we chose three distinct data transformation steps.
Loading Data

Data Transformations (no simple pipe)

- DataConverter – converts data from native format (SQL, flat file, acedb, etc.) into our object-oriented database format.
  - The converter needs to transform foreign keys to references, and translate many-to-many collections properly.
    - This involves searching in the source database.
    - For flat files, it often needs to hold significant amounts of data back in memory (for example, objects with collections – the collection will build up as other objects are converted).
Loading Data

Data Transformations (no simple pipe)

- DataTranslator – converts data from the native object model into the desired model, using OWL-described transformations.
  - Many of these transformations need to search in the source database.
    - For example, attributes can be moved from a referenced object into the parent object – the referenced object needs to be searched for.
    - Also, the class of an object may depend on data held in objects referenced by the object.
Loading Data

Data Transformations (no simple pipe)

- DataLoader – loads multiple databases into one, resolving overlaps and conflicts, *etc*.
  
  - Applies the source database as an object graph over the top of the already-existing one in the destination database.
    
    - A very logical step – takes a database that is already in your format, and merges it with others. Therefore, if you only had one data source, you could miss this step.
    
    - Needs to search in the source database, to build the object graphs to merge with the destination database properly.
Loading Data
The DataLoader

- Actually, the DataLoader is not that simple.
  - Our intermediate databases are actually “Items databases”, where each object is represented by an Item object that has a classname, and Attributes, References, and Collections.
  - This is in order to simplify the DataConverter and DataTranslator.
  - But the DataLoader really does retrieve objects from the source ObjectStore in the same format as the final destination ObjectStore.

- We use an ObjectStoreTranslatingImpl, which modifies the queries and reconstructs the results.
### Loading Data

**The DataLoader**

- **Mode of Operation:**
  - Take an object from the source.
  - Find all the objects in the destination that are equivalent to it by whatever primary keys are available to the data source being loaded.
    - Note – this may be more than one object!
  - Merge each field together by data source priority.
    - For references and collections, look up the referenced objects in the destination by primary key equality, so that objects get linked together properly.
Loading Data
The DataLoader

Data source

Destination database

Dr. Fred Bloggs

Likes

Ketchup

Happy Meal

Large Coke
Loading Data
The DataLoader

- Note: Not all data sources can handle all primary keys.
  - Imagine “first name” was a primary key on the previous slide, as well as “surname”.
  - Therefore you can have more than one equivalent object in the database to be merged with the object being stored.

```
first name = “Fred”
age = 34
surname = “Bloggs”
gender = male
```

```
+ first name = “Fred”
surname = “Bloggs”
age = 34
gender = male
```
Loading Data

The DataLoader

- Sometimes the data sources will conflict on the value of a (non primary key) field.
  - One of the data sources will be the “most reliable”.
  - We have to keep track of where each piece of data came from, so we know which value is most right.

```
first name = “Fred”
age = 34
gender = female
```

```
first name = “Fred”
age = 10
gender = male
```

```
first name = “Fred”
age = 34
gender = male
```
Sometimes two objects being merged will be of different classes.

This is not a problem as long as the classes are compatible (i.e. one a subclass of the other, or one or more an interface).
Loading Data

The DataLoader

- The DataLoader must process every single object in the data source.
  - But it also needs to recurse into references and many to many collections, in order to set them.
  - However, it does not want to recurse any deeper than necessary for each object (for performance).

- The large ovals represent objects that are loaded properly, while the others are loaded as “skeletons”.

```
+  +  +  +
```

- ▶
- ▶
- ▶
- ▶
Loading Data Field Types

- Different field types are treated differently by the data integration system:
  - Attributes are database primitive types.
    - They are copied verbatim by the DataConverter and DataTranslator, and chosen by source priority by the DataLoader.
      - Numbers (short, int, long, float, double, Short, Integer, Long, Float, Double, BigDecimal)
      - Dates
      - Strings
      - Booleans
Loading Data Field Types

- One to one references are a reference from one object to another, where the reverse reference is consistent. A reference may be null.

  ➢ The DataConverter enforces the creation of consistent one to one references.

  ➢ The DataLoader may need to override a one to one relationship with more reliable data, so it must occasionally null old reverse references to keep the references consistent.
Loading Data Field Types

- Many to one references need no consistency checks. There may be a reverse one to many relationship in the model, or the reference may be unidirectional.

  - The DataConverter, DataTranslator, and DataLoader can treat these in the same way as attributes, by source priority.
Loading Data Field Types

- One to many relationships are collections where each element of the collection is in no other similar collection. They are stored as a foreign key in the referenced objects.

- Because a one to many relationship can never be unidirectional, the DataLoader never bothers with them – it assumes that the relationship will be sorted out properly when the objects in the collections are stored. Those objects will be stored with a many to one reference to this object.

- The DataConverter and DataTranslator don't bother either.
Loading Data Field Types

- Many to many relationships are stored with an indirection table, containing foreign keys to both sides of the relationship. Such relationships can be unidirectional or bidirectional.

  - The DataConverter creates many to many relationships in the form of lists of object IDs in the Items database, which are presented to the DataLoader as collections of objects by the ObjectStoreTranslatingImpl.

  - The behaviour of the DataLoader is to merge the contents of many to many relationships.
The SQL Optimiser

- The purpose of the FlyMine project is not only to allow people to run cross-reference queries (other tools will do that a little), but to allow people to run really complicated cross-reference queries that would take a long time any other way.

- Therefore, we know that people will want to run complex arbitrary time-consuming multi-table joins.

> We want these to run fast.
The SQL Optimiser

- In SQL, one can create a “view”, which represents the table that is the results of a particular query.
  - One can then query this view exactly as if it was a real table.
  - The SQL server doesn't actually store the contents of the view – it works it out from the query each time it is accessed.
    - Actually, it cheats, by integrating the view query into the query that mentions the view, then optimising the whole query.
The SQL Optimiser

- A precomputed table is the opposite of a view.
  - Views are tables that don't exist, but you can query them.
  - Precomputed tables are tables that do exist, but you don't query them.
- A precomputed table stores the results of a query, with the hope that the data will enable the server to run related queries faster.
The SQL Optimiser

- Say you have a large table (e.g. 10,000,000 rows), and you wish to select 100,000 rows:

```sql
SELECT * FROM table WHERE name LIKE 'Interesting%'
```

- Indexes would help a little, but if you had a precomputed table like this:

```sql
SELECT * FROM table WHERE name LIKE 'Interesting%'
```

- You could just run this SQL query:

```sql
SELECT * FROM precomputed_table
```

- This would run faster.
The SQL Optimiser

- Or, if you wish to select a small number of rows, but your filter is complex and not indexable:

```sql
SELECT * FROM table WHERE name LIKE 'Interesting%'
AND <something complicated>
```

- You could run this instead:

```sql
SELECT * FROM precomputed_table
WHERE <something complicated>
```

- Because this only needs to filter 100,000 rows instead of 10,000,000, it will be 100 times as fast (disregarding indexes).
The SQL Optimiser

- A precomputed table can be the result of almost any query, including multi-table joins.
- The optimiser creates optimised queries by fitting the precomputed tables into the query.
  - A precomputed table “fits” a query if its constituent tables can be mapped onto the tables in the query, its WHERE clause does not throw away any rows that the query's WHERE clause does not throw away, and it has sufficient fields.
- An optimised query can utilise multiple precomputed tables.
The SQL Optimiser

- To find out which is the fastest query, the optimiser asks the SQL server to “EXPLAIN” how long each will take.
- The optimiser will not spend more time optimising a query than the time the database thinks it will take to run anyway.
- We have seen improvements between 10% and 10000% in performance, however it is difficult to measure performance because the gain depends on many variables.
The SQL Optimiser

- The optimiser decouples the database model presented to the user from the underlying database schema.
  - We can present a “nice” model to the user.
  - We can choose a set of precomputed tables, such that the database acts like it has a nasty performance-tuned schema.
  - We can adapt the precomputed tables on the fly to the actual queries that are run by users.
  - Even just providing copies of tables pre-sorted by different fields is valuable to the DB server.
Future Stuff

Relationship Metadata

- In the FlyMine project, we need to store data concerning the relationships between objects.
  - For example, if someone has associated a Gene with a particular Property, we need to store data concerning who made the association, how reliable it is, references to published papers, *etc.*
  - The Gene class should have a many to many relationship with the Property class.
  - However, because we want to store this extra data, we have an object in the middle of the relationship as an “indirection table”.
Future Stuff
Relationship Metadata

- Current object model:

  - Gene
  - Relation
  - Property
  - Evidence

  Gene 1 0..* Relation 0..* 1 Property

- Desired object model:

  - Gene 0..* Evidence 0..* Property 0..*
Future Stuff
Relationship Metadata

- Currently, the Gene object has a `getRelations()` method, which returns a Collection of Relations.
  
  ➢ You then need to call `Relation.getProperty()`.

- In the future, the Gene object would have a `getProperties()` method which would return a Collection of Properties, plus a `getPropertiesWithEvidence()` method that would return a Map from Property to Evidence.

  ➢ `getProperties()` would be the equivalent of calling `getPropertiesWithEvidence().keySet()`.
Future Stuff
Graph Reasoning

- In the field of computational biology, graph reasoning is a hot topic.
  - Graphs can be used to represent biological processes, ontologies (taxonomies and partonomies), chemical reaction flowcharts, etc.

Biologists want to ask questions like:

- What happens if I add/remove this chemical?
- Show me all the genes expressed in the liver, but not in this particular part of the liver, where that information is quite reliable.
- I have a mutated fly that died at a particular developmental stage due to <insert appropriate large word> - which genes might have been broken?
Future Stuff
Graph Reasoning

- Both IBM and Oracle have announced graph reasoning extensions to their database products.
- RDF (and OWL) are probably the most powerful languages in which to perform graph reasoning, so it may become the standard among the bioinformatics community.
- We intend to add graph reasoning capabilities to our system, possibly in collaboration with the universities of Manchester and Newcastle.
Acknowledgements


The InterMine system (www.intermine.org) is a generic object-oriented database and data integration system, open source and licensed under the LGPL.

FlyMine (www.flymine.org) is a biology-specific application of the InterMine system, also licensed under the LGPL.

FlyMine is funded by the Wellcome Trust (grant no. 067205), awarded to M. Ashburner, G. Micklem, S. Russell, K. Lilley, and K. Mizuguchi.

FlyMine Group, Department of Genetics, University of Cambridge, Downing Street, Cambridge, CB2 3EH, UK
Tel: +44 1223 333377 Email: info@flymine.org