ModFetch: A Modular System for Automating Queries to Relational Databases

Ari N. Schulman, B.S.

Submitted in partial requirement for
Special Honors in the Department of Computer Science
The University of Texas at Austin

May 2009

Professor Don Batory
Department of Computer Science
Supervising Faculty

1 Introduction

The rise of object-oriented (OO) programming in conjunction with relational databases as a persistent storage mechanism has led to a great deal of interest in Object-Relational Mappers (ORMs) that automate queries on behalf of OO systems. ORMs are subject to a number of problems that arise when attempts are made to map OO data to conceptually distinct relational databases. These mapping problems are broadly grouped together under the term "object-relational impedance mismatch" [1].

Many problems result from the distinctions in semantics and paradigms between OO and relational models. For example, OO encapsulation makes little sense in principle and cannot easily be achieved in practice in relational databases. Similarly, the arbitrary use of polymorphism and inheritance in OO programs is difficult to map to relational tables which must remain distinct from each other. Additionally, subtle data type distinctions such as limits on string length (which is usually explicitly bound in relational DBs but is bound only by memory size in OO programs) may be difficult to map except by manual runtime checks [2].

Previous research into solving the "object-relational impedance mismatch" has focused largely on how to eliminate the mismatch itself. Often such research focuses on flaws in one or the other type system, suggesting that one ought be eliminated or made to conform more to the other. Database advocates may argue that concurrency problems make OO languages a poor candidate for maintaining persistent data. OO advocates point to the advantages of OO programming and the limitations of relational DBs, and many have conducted research into creating object-oriented database systems that more closely mirror the structure of OO data itself [2].

Whatever the root cause may be, what is hardly disputable is that no dominant framework for automating queries for an OO system has emerged from such research. As noted by Bob Walker in a 2006 panel on the subject, "Current techniques for decomposing complex multidimensional object graphs into storage mechanisms designed for an entirely different purpose are fundamentally flawed. All have advantages and disadvantages; none supply a complete, simple and elegant solution." [3]

The cases in which OO and relational DB systems are mismatched are indeed a serious concern for ORM developers. But perhaps a more immediate and pervasive problem for most OO developers is not the case of *mismatch* but the numerous cases of *overmatch*, in which OO operations can so clearly and regularly be transformed into relational queries that there is a great deal of similarity between queries, and hence a great deal of repeated work by the programmer. Because relational query languages such as SQL are designed to support a much more diverse set of operations than are typically employed in OO operations, much of the complexity of SQL becomes redundant in queries made on behalf of OO systems.

While previous research has dealt extensively with the mismatch between OO and relational DB type systems, attempting to reduce the distance between them, the

importance of the query itself as a means of communicating between the systems has been overlooked. In fact, queries offer distinct advantages over both the OO systems and the underlying relational schemas that they mediate between. For example, an ORM may allow some object `A` to transparently persist some linked set of objects as an attribute `A.B`. But the programmer may need only a subset of `A.B`. It is straightforward to specify the appropriate additional WHERE constraint in a literal query, but when using an ORM, the programmer typically must declare an entire new set and corresponding attribute of `A`. Some research has been conducted into using static analysis to automatically determine such subsets, but this work lacks the accuracy and simplicity of simply allowing the programmer to easily specify such a subset [4][5]. And so the effort to engineer schemes for *full* transparent persistence of OO data into databases may in fact obscure certain advantages of literal queries.

This paper proceeds from the claim that the query is a rich locus of access to a relational database, but that the method of expressing a query need not be limited to the literal strings that are sent to the database. Rather than making queries transparent, we aim to preserve the full power of queries and instead to minimize the amount of repeated effort required by OO programmers in writing queries. To that end, we focus on the problem of more efficiently expressing the types of queries used in an OO program. Additionally, we explore a means of reusing and modularizing such query expressions. We propose a framework called ModFetch that provides these features.

2 Background

In order to explore query automation, we will work with a hypothetical OO program and relational database that manage a university course system. Initially, we define a School and Department class, with a one-to-many relationship. A Department in turn has many Courses. A Course, a Term and a Prof together constitute a Section.

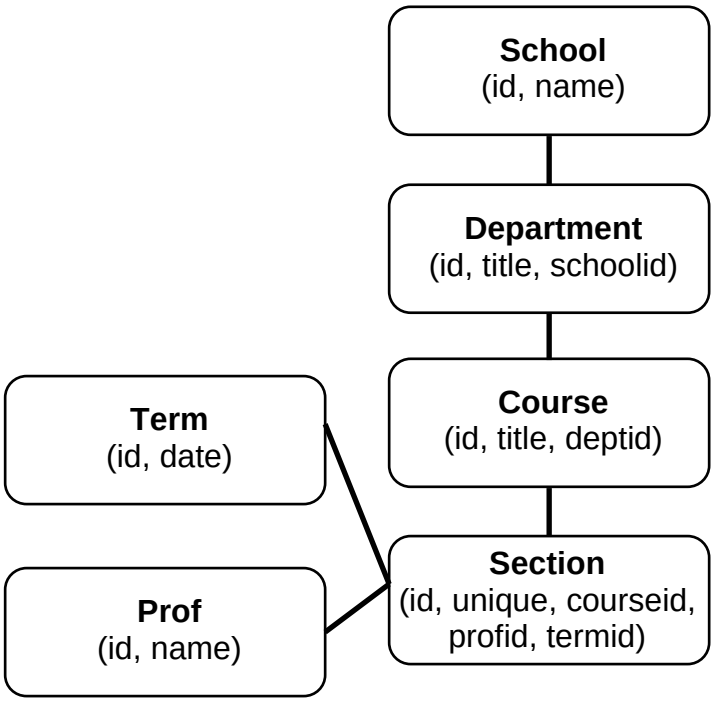

Figure 1: Example course-related database schema

2.1 Problems with literal query strings

SQL queries corresponding to OO operations must use a semantics that is more complex than the semantics of the OO operations that they represent. Consider the basic operation in which a School is to be retrieved by its primary key of value *n* from the database. The SQL query will look like:

```
SELECT schools.* FROM schools WHERE schools.id=n
```

A rough English translation of this query might read: "Lookup the `schools` table; then find only rows such that the `id` column of the `schools` table has a value equal to *n*; then return all of the columns in the selected rows of the `schools` table." But this operation might be represented in OO style by a semantically simpler piece of code such as:

```
DBHandle.retrieveByPrimaryKey("schools", n)
```

The redundancy of SQL semantics becomes clearer when considering join operations. Consider a scenario in which we have some Department `dept` with primary key value *n*, and we wish to retrieve its School. The SQL query will look like:

```
SELECT schools.*
FROM departments
INNER JOIN schools ON schools.id=departments.schoolid
WHERE departments.id=n
```

The rough English translation of this query is: "Lookup the `departments` table; then lookup the `schools` table; for each row in the `departments` table, match it to every

row in the `schools` table where the `schools` row's `id` field is equal to the `departments` row's `schoolid` field…", and so forth. The query necessitates the repeated use of table names, primary key names, and foreign key names, and the programmer's need to analyze how these values are matched both when creating the query and later, when reading it. What would more useful both for writing and maintaining queries is a means of expressing the simpler OO-style semantics being represented—in this case, something like: "Get the Department with primary key *n*, then get its School."

We also observe that literal query strings are not easily amenable to reuse. For example, suppose we have a SQL query representing the OO operation, "Get the Section with primary key *n*, then get its Course, then get the Course's Department." Later we may want to extend the chain one more step and fetch the Department's School. If we want to save effort by reusing the query we have already written, we must divide the original query into several separate strings—one for the SELECT clause, one for the tables clause (containing FROM and JOIN statements), one for the WHERE clause, and in other instances, one for the ORDER BY clause. Even so, only the tables clause could be usefully reused by appending the new JOIN statement to it.

Furthermore, literal query strings are not easily amenable to parameterization of tables themselves. For example, suppose that other objects in our schema link to the `schools` table; there is no easy way to write a function that takes some object *x* as a parameter and returns its School. And the example may be impossible when joining across sets instead of single objects. For example, consider a function that takes as a parameter some arbitrary set `s` of Section objects and executes a single efficient query to return the Course for each Section in `s`. This problem cannot be solved without using reification or extending the query language itself [6].

2.2 Previous work

The Reification Object-Oriented Framework (ROOF) offers a solution to most of the problems outlined in the previous section [7]. The framework presents a general solution to the problem of polylingual interoperable applications, offering a method for one application to seamlessly access and manipulate data in a foreign type system. ROOF first uses type reification to convert foreign type systems into first-class host language objects that can be accessed and manipulated. It then claims that because all type systems can be represented as graph schemas, operations on foreign type systems can be represented as path expressions through such graphs.

ROOF defines a class called *ReificationOperator* that encapsulates reified foreign types and provides for basic operations on them:

```
class ReificationOperator {
   public ReificationOperator getObject(string t);
   public Object[] fetch();
}
```

Suppose we have a ReificationOperator `R` that is a handle to some object `r` in a foreign type system. The `getObject` method returns a handle to `r`'s attribute named `t`, which must be some other object, not a primitive. The resulting ReificationOperator can then be used to return a handle to object attributes of `r.t`.

If we were working natively in our foreign type system (that is, from the host system rather than from some program that is accessing it), we might specify some path through object attributes such as:

```
r.a.b.c
```

With ROOF, we can access this path from another program by specifying the same path expression through ReificationOperators. In this case, we are given a ReificationOperator `R` that is a handle to `r`, so the path can be expressed as:

```
R.getObject("a").getObject("b").getObject("c")
```

Then we can call `fetch()` on this ReificationOperator. It will generate a low-level API call to the foreign type system corresponding to the expression `r.a.b.c`, retrieve the results, and assemble it into some native object that can be manipulated by the programmer. ROOF's approach of using path expressions and elevating foreign type systems to first-class objects will be integral to our solution.

3 Our solution: ModFetch

How might ROOF look when we apply it to our database problem? Let us return to a previous scenario, in which we have some Department `dept` with primary key value *n*, and we wish to retrieve its School. Given a ReificationOperator `Rd` that is a handle to `dept`, we may easily retrieve a handle to its School:

```
Rd.getObject("schoolid")
```

This expression specifies a handle to the School object pointed to by `dept.schoolid`. We can then use the handle to retrieve the data for the object itself, and in turn use the data to construct and populate a new School object:

```
School s = new School(Rd.getObject("schoolid").fetch());
```

This approach is simpler than the previous use of a multi-line literal query. By reifying the type system of our relational database, we are able to construct first-class objects that represent queries, and manipulate those objects directly. Furthermore, ROOF's use of path expressions closely corresponds with the suggested semantics of OO expressions such as "Get the Section with primary key *n*, then get its Course, then get the Course's Department."

3.1 Query reuse

In Section 2.1, we noted that literal queries cannot easily be reused. Consider the following example: we have a ReificationOperator `Rsec` for a Section `s`, and we wish to fetch the Department for `s`. With ModFetch, this is straightforward:

```
ReificationOperator Rdept =
   Rsec.getObject("courseid").getObject("deptid");
Department d = (Department) Rdept.fetch();
```

Now suppose that later we also want to fetch the School for `s`. It is simple to reuse the previous expression and extend it, without having to redeclare it:

```
School s = (School) Rdept.getObject("schoolid").fetch();
```

This feature is particularly useful for longer paths. But its true power may become clear when considering a slightly different example. Suppose that our database administrator notices that it is inefficient to always have to traverse the Courses table when moving from Section to Department. S/he thus decides to add a link from Section to Department:

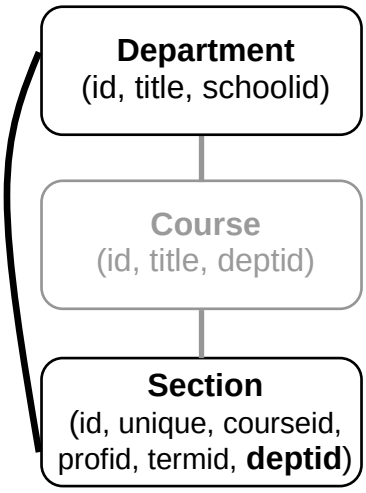

Figure 2: Updated portion of database schema

Were we still using literal queries, we would have to update each and every query string written in our program. But for this example, it is as simple as updating the single ReificationOperator that expresses the path from Section to Department:

```
ReificationOperator Rdept = Rsec.getObject("deptid");
```

3.2 Modular Queries

The preceding example hints at the possibilities that ModFetch opens up for modularizing queries. Consider the following scenario. We wish to create a module that will print information about an arbitrary set of Sections, such as the unique number of each Section. We can achieve this through a parameterized function using ModFetch:

```
public void printSections(ReificationOperator sectionPath) {
   Section[] sections = (Section[]) sectionPath.fetch();
   for (Section s : sections) {
      // print data
      ...
   }
}
```

Note that the function takes as a parameter a ReificationOperator that specifies a path to any set of Sections. So we can abstract away the implementation of our module, and simply pass to it a path expression. We have not yet described the full range of possible path expressions, but we can imagine their rough English descriptions, and how they might work with our module: "Print information about all of the Sections offered by some Department `d`", "Print information about all of the Sections in Course `c` that have been taught by Prof `p`", and so on.

This feature might seem to have only limited advantages; for even with literal query strings, we could create a function that takes a query string that selects some arbitrary set of Sections—or, for that matter, we could create a function that takes the set of Sections themselves as the parameter. However, suppose that at some later date we decide that we also want our function to print the Term for each Section. With ModFetch, this can be easily be accomplished by updating *only our module*. Again, we have not yet specified the full semantics of ModFetch necessary to support this operation, but its rough English translation would read: "Given the set of Sections represented by `sectionPath`, fetch those Sections along with the Term for each Section".

Note that the programmer need not decide at the time that `printSections` is invoked what additional objects will need to be fetched for each Section—only which Sections will be fetched. If all of the desired objects are to be retrieved in a single fetch, this cannot be accomplished by parameterizing a literal query string or a set of objects. The ability to abstract these decisions is the essence of the modularity feature in ModFetch.

ROOF, as noted, leaves incomplete a number of operations that will be necessary for our framework, including the operations necessary to exercise the full power of the `printSections` module outlined here. We turn now to describing these operations that ModFetch provides.

4. Completing query semantics

4.1 Reifying Queries

As noted in the previous section, ROOF provides a simple framework that satisfies the basic requirements for the querying system we wish to build. But some details of how this system would be applied in practice need to be specified.

First, note that to reify the path expressions of queries, two structures in fact must be reified. Since such expressions specify paths across type schema graphs, they must include information about both the nodes and the edges of the graph that is traversed. Nodes are database tables, while edges are the predicates used to join them. Both nodes and edges must be reified in some data structure.

In the database schema graph, a node is a table. But since we are reifying queries, we are interested not only in *which table* to select, but *which subset of the table's rows* we should select, and *which of the table's fields* should be retrieved. A ReificationOperator thus must store these three pieces of data, and offer a corresponding constructor:

```
class ReificationOperator {
   private String tableName;
   private String selectClause;
   private String whereClause;
   public ReificationOperator(String t, String s, String w);
   ...
}
```

This constructor can be used to create the root ReificationOperators from which we construct path expressions. But it is particularly important to be able to easily move from an individual object to its ReificationOperator, since we are typically dealing with queries that correspond to OO operations on objects. Given some object `obj` and its table name, it turns out to be straightforward to construct a ReificationOperator that corresponds to `obj`. Using the table name, we can query the database and determine the primary key name, and then extract the primary key value from `obj`. This provides sufficient information to assemble the table name, SELECT clause, and WHERE clause that represent a fetch of this single object, and so we can construct a ReificationOperator for `obj`. If the programmer chooses to specify a class's corresponding table within the program, then an object's ReificationOperator could be generated through an instance method:

```
ReificationOperator RO = obj.getRO();
```

Finally, a simple data structure named *ReificationNodeLink* will store the join predicate, which is an edge in the query graph. The class links to a parent and a child ReificationOperator that specify which tables are being joined together. It also specifies the names of the column from each table that will joined. Typically one column is a primary key while the other is a foreign key. The parent-child relationship refers to the order in which the nodes are fetched—and, correspondingly, the order in which the ReificationNodes are linked together.

The graph structure is stored by maintaining a handle to a single ReificationOperator, representing one node in the query graph. This node can store two edges, which join the node to a parent and child node. With this structure we can store a query path expression.

4.2 Holes in ROOF

Although we have demonstrated how ReificationOperators for single objects can be generated, a number of OO operations and query features still need to be supported.

4.2.1 Retrieving sets

It is already clear how we may specify one-to-one paths from one object to another, but we need to have a means for specifying one-to-many paths. For example, given a Department `dept`, we can call `dept.getRO().getObject("schoolid")` to retrieve the School that `dept` links to. But in the opposite direction, when we have a School `s`, there is no method for retrieving the set of Departments that are in `s`.

This case can be handled by adding a new method to ReificationOperator that takes as a parameter the name of the foreign table:

```
public ReificationOperator getSet(String foreignTableName);
```

Then the scenario in which we want to fetch the Departments for some School `s` will look like:

```
s.getRO().getSet("departments")
```

The declaration of foreign keys or references in the database structure is used to determine which column in the `departments` table links back to `schools`.

4.2.2 Outer joins

Our system thus far has no way to express outer joins, which are an important feature of relational DB queries. The parent-child semantics in the ReificationNodeLink class can be used to enforce the directionality of outer joins. All we need to add to the ReificationOperator class is a Boolean indicating whether the join is outer or inner.

We also need to add methods to the ReificationOperator class to support specifying outer joins. This can be accomplished by adding a Boolean parameter to our existing `getObject` and `getSet` methods, which for the sake of convenience can default to specifying an inner join. In order to reduce the number of parameters, we might also add methods `getObjectOuter` and `getSetOuter`, which are semantically identical to their counterparts but specify an outer join. Note that paths are directional, so the existing node is always fetched before the new node generated by a method call. That is, `a.getObjectOuter("b")` means "a left join b", which is distinct from `b.getObjectOuter("a")`.

4.2.3 Retrieving multiple sets

Another shortcoming of the system thus far is that we can only retrieve objects from a single table at a time. For example, suppose that we have some Section `s`, and we want to fetch its Course, Department, and School. Currently, this must be accomplished through three separate operations, and three separate database queries:

```
s.getRO().getObject("courseid")
s.getRO().getObject("courseid").getObject("deptid")
s.getRO().getObject("courseid").getObject("deptid").
  getObject("schoolid")
```

Obviously, this method requires redundant work. We should be able to retrieve all three objects in a single fetch, and specify this fetch in a single expression.

In the ROOF system, the `getObject` operator uses reification to move from a handle on one object to a handle on another object. But in ModFetch, our goal is to automatically generate a single query that will accomplish all of our work. Consequently, when a new object handle is created as in the expression `R.getObject("foo")`, the old handle `R` is linked to the new handle instead of just being released to the garbage collector. We are, in other words, collecting a reified structure for the *entire path* through the graph—not simply using the graph to move from one reified node to another node. When the objects are finally fetched, the entire reified path must be visited in order to construct the corresponding query.

Therefore we just need a means of specifying that particular nodes be included in the final retrieval. Again, we simply need to add a Boolean attribute to the ReificationOperator class that specifies whether the node should be selected. And we can add a parameter to our `getObject` and `getSet` accordingly. Additionally, it may be convenient to define methods `retrieveObject` and `retrieveSet` that call `getObject` and `getSet`, respectively, with this parameter set to true. Our original example will then look like:

```
s.getRO().retrieveObject("courseid").retrieveObject("deptid").
  retrieveObject("schoolid")
```

In ModFetch semantics, the lowest node will always be retrieved, so the final method call can equivalently be either `retrieveObject` or `getObject`.

4.2.4 Retrieving node children

Now consider a case in which we have a set of Sections represented by a ReificationOperator `Rs`, and we want to retrieve the Term and Prof for each Section in a single operation. It is straightforward to do either of these operations, but not both in the same query:

```
Rs.getObject("profid")
Rs.getObject("termid")
```

All we need, then, is a method that will add a new node onto the graph, but return a handle to the old ReificationOperator instead of the new one. This method we will call `addChildObject`. It is semantically identical to `getObject` except for which ReificationOperator it returns.

Now we can express our operation. First, we call:

```
Rs.addChildObject("profid")
```

This adds a join from Section to Prof, but returns a handle to the Section node. Consequently, we can invoke another operation on the Section node:

```
Rs.addChildObject("profid").addChildObject("termid")
```

This expression specifies a single query that retrieves both the Prof and the Term for each Section in `Rs`.

4.2.5 Specifying parents

Consider the following scenario. We have some Prof `p` and we wish to fetch the Sections taught by `p` during some Term `t`. Fetching *all* of the Sections taught by `p` is straightforward:

```
p.getRO().getSet("sections")
```

But there is no way to constrain those Sections to only the ones offered during `t`.

So, we must introduce a new method to ReificationOperator that permits the addition of join constraints to a node:

```
public ReificationOperator addParent(ReificationOperator p);
```

Now we can specify our path expression as:

```
p.getRO().getSet("sections").addParent(t.getRO())
```

Like `addChildObject`, the `addParent` method returns a handle to the current rather than the new node, so we could extend the above expressions to include child paths off of the Section node.

Since a node can now have multiple parents and children, we must modify ReificationOperator to be able to point to an arbitrary-length list of ReificationNodeLinks. The ReificationNodeLinks themselves store the necessary information about whether a node is the parent or the child in a link.

4.2.6 Cyclic queries

Consider the following addition to our database schema from Figure 1:

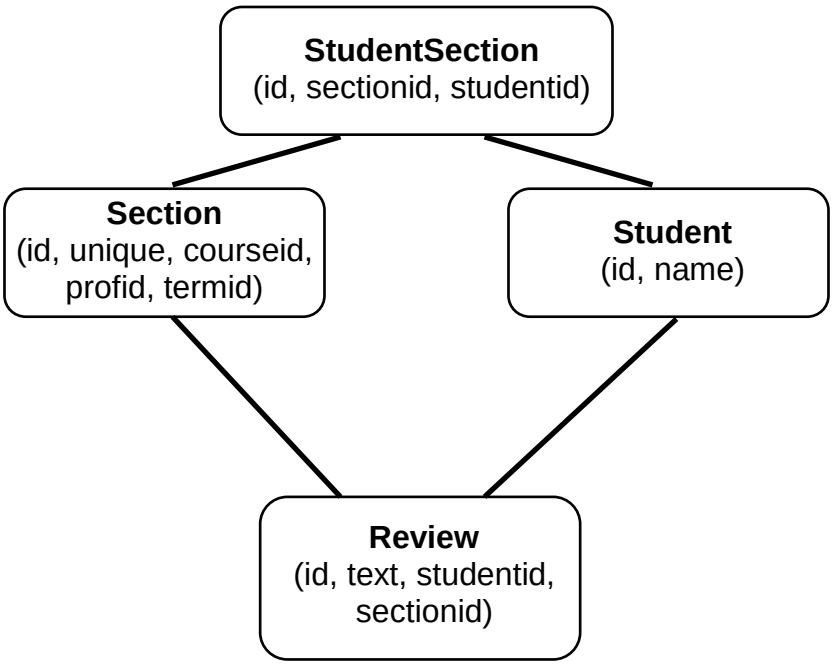

Figure 3: Student-Review type schema

Here we extend our schema to handle Student objects. Students can also write Reviews for particular Sections. The Student-Section relationship is many-many, so we also require an association table, StudentSection.

Now suppose we have some Student `st`, and we want to fetch all of the Sections that `st` has taken, along with any Review that the Student may have written for that Section. But we want to return all Sections, regardless of whether the Student has written any Reviews for them—which means an outer join will be required. The following expression might seem to work:

```
st.getRO().getSet("student_sections").
 retrieveObject("sectionid").getSetOuter("reviews")
```

However, this expression would retrieve *all* of the Reviews written by *any* Student for each Section, not just those written by `st`.

What we need is a way to specify that our ReificationOperator `st.getRO()` is a parent of *both* the StudentSection node *and* the Review node. Our existing semantics in fact implicitly supports this:

```
ReificationOperator st_ro = st.getRO();
Review[] reviews = (Review[]) st_ro
  .getSet("student_sections")
  .retrieveObject("sectionid")
  .getSetOuter("reviews")
  .addParent(st_ro).fetch();
```

The semantics allow us to specify all four of the necessary links between nodes. When `addParent(st_ro)` is called on the Reviews node, no new node is added to the graph. Instead, an edge is added between the existing Review and Student nodes.

However, recall that every method, in modifying the graph, in fact copies the graph and performs modifications on the new copy. Therefore the existing graph will contain a *copy* of `st_ro`, not the instance `st_ro` itself. But the `addParent` method must be able to determine whether its parameter already exists in the graph. The solution is for each ReificationOperator to store a pointer to the original ReificationOperator that it is a copy of (if it is indeed a copy).

In our example, the call to `st_ro.getSet` first creates a copy of `st_ro`, then sets the copy's `originalRO` attribute to point to `st_ro`. This pointer is maintained when the graph is later copied again by subsequent method calls. Finally, when `addParent(st_ro)` is called, it first creates a copy of the graph. Then it checks the `originalRO` attribute of each ReificationOperator in the graph. If any matches `st_ro`, then the method adds an edge between the graph's copy of `st_ro` and the node upon which `addParent` was invoked. Otherwise it creates a copy of `st_ro` and adds it into the graph, along with the appropriate edge.

4.2.7 Specifying arbitrary subsets

Finally, we want ensure that we do not obscure the ability we have with normal queries to perform searches or simply to constrain our result sets to only the rows that we need. For example, suppose that we have some School `s` and want to print out a list of its Departments, but only intend to show Departments starting with a particular letter at a time. Fetching the entire Department set is simple enough, but we want to avoid the need to fetch the entire set and filter it later. Consequently, we need simply permit the programmer to specify arbitrary WHERE constraints as a parameter:

```
public ReificationOperator getSet
  (String foreign, String whereClause);
```

The additional parameter will of course default to the empty string.

Now we can easily handle our example:

```
s.getRO().getSet("departments", "departments.name LIKE 'A%'")
```

5 Fully modular queries

We can now show the implementation of the hypothetical module described in Section 3.2. Recall that we wished to create a module that will print information about an arbitrary set of Sections, such as the unique number of each Section. We can now show a couple examples of what some calls to that function will look like. For instance, we can print all of the Sections offered by some Department `d`:

```
printSections(d.getRO().getSet("courses").getSet("sections"));
```

Or all of the Sections in Course `c` that have been taught by Prof `p`:

```
printSections(
  c.getRO().getSet("sections").addParent(p.getRO()));
```

Consider now the example in which we decide at some later date that we also want our function to print the Term for each Section. With ModFetch, this can be accomplished by updating only our module. For the sake of brevity, we will assume that our `Section.fetch` method stores in each Section any additional objects that are fetched (which can trivially be accomplished). Our updated module will look like:

```
public void printSections(ReificationOperator sectionPath) {
   Section[] sections = (Section[])
      sectionPath.addChildSet("termid").fetch();
   for (Section s : sections) {
      Term t = s.getStoredObject("Term");
      // print data
      ...
   }
}
```

Again, note the advantage provided by the fact that the programmer need not know at the time that `printSections` is invoked what additional objects must be fetched for each Section. ModFetch provides the ability to institute these abstractions. One example application of the ability to encapsulate paths through the database is the ease of enforcing user permissions. The programmer can declare a global variable specifying the path expression to a set of users with permission to access particular types of resources, and link this variable together with arbitrary nodes relating to those resources. If the permissions or the scheme for specifying them in the database later changes, only the global variable needs to be updated in the program, instead of each and every expression that relies on those permission specifications. There are many potential additional applications for this type of modularity.

6 Querying and retrieving the data

With a rich semantics for specifying query graphs outlined, we now turn briefly to the implementation details. The basic outline for generating a query is to traverse the reified graph, collecting SELECT and WHERE clauses, and assembling table names and join predicates into a tables clause. The clauses can then be joined together to form the final query.

The query is then sent to the database. The results can be returned to the programmer as some simple data type. However, the programmer can easily wrap the `fetch` method in an automated helper method that converts the result sets into some more convenient data structure such as OO objects.

A note must be made about the order of the graph traversal. Because ModFetch semantics supports multiple parents and children for each node, multiple traversal orders are possible. But the notions of *parents* and *children* themselves imply constraints on the fetch order (this constraint is particularly important for outer joins). Consequently, we must satisfy the condition that every parent node be visited before its child nodes. This ordering is known as a *topological sort* [8]. There are several known algorithms for determining such an order that are linear in the time of the number of edges plus the number of nodes [9][10]. These algorithms have the additional benefit of detecting any cyclic dependencies.

Once a topological sort is obtained, the algorithm for generating the tables clause is straightforward:

For each ReificationOperator `ro`, do:

1. Add `ro`'s table name to the tables clause.
2. For each of the join links in `ro` for which `ro` is the child, add the join predicate to the `ON` clause. If there are multiple predicates, they are joined together with `AND`s; and if there are no predicates, no `ON` clause is included.

Note that the first node is not dependent on any other node—that is, it has no parents. Consequently, in Step 1, its table name will be added to the tables clause using the `FROM` declaration, and it will not have any `ON` predicate in Step 2. For every other node, in Step 1 the table name will be appended along with a `LEFT JOIN` or `INNER JOIN` declaration. `LEFT JOIN` will be used if any of the links added to the join predicate in Step 2 are outer joins.

7 Related Work

A popular Object-Relational Mapper for the Java language is Hibernate [11]. Hibernate is somewhat similar to ROOF and ModFetch in that it permits the transparent use of linked objects and sets. For example, in our School class, we might add to it a Set<Department> attribute called `depts`, which can be transparently persisted to a database using Hibernate. However, Hibernate is designed to provide a one-to-one mapping between Java classes and database tables. In order to achieve this mapping, every attribute and set must be explicitly declared in an XML mapping file. This introduces additional complexity and overhead in initially establishing the mapping.

The ROOF approach offers the advantage that it is does not attempt to *map* two distinct type systems to one another, but simply provides a framework for manipulating one type system from another. Similarly, ModFetch aims not to provide a means to obscure the distinction between program and database, but rather to remove the hassle of accessing the database by reifying its interface, the query. What this means in practice is that we are freed from the need to explicitly declare all of the queries we might want to use in some external file. We can within the program decide whether we want to fetch all of the

Departments for some School, or just a subset of them, just as we would when using string queries. Moreover, we can dynamically declare path expressions across multiple nodes, and within those expressions declare which of the nodes we want to fetch.

These problems with Hibernate have been the focus of research that performs static analysis of code in order to automatically determine the query that most efficiently avoids lazy loading and the selection of rows that are not used in the program [4][5]. This approach offers significant advantages in *transparency*, in that the programmer theoretically is relieved of the task of specifying an efficient query. However, the approach introduces substantial added complexity to the persistence framework through the use of static code analysis.

ModFetch instead offers the programmer the ability to make the decisions about precisely what data should be fetched, as is normally the case when writing literal queries. The advantage ModFetch offers is that the programmer retains the same power that s/he has when writing literal queries, but is able to exercise these powers more easily. Additionally, it allows for the modular division that is impossible in literal queries, so that a query path can be specified at one point in a program and later passed to another part of the program that can decide to extend that path.

8 Further Work and Conclusions

We believe that ModFetch provides a powerful framework for expressing queries to databases designed to persist OO data. ModFetch has been deployed in a test environment similar to the example course system, and been found to substantially reduce code size and increase the ease of writing queries and OO modules, with minimal overhead. However, the framework remains to be tested and evaluated by programmers not involved in its development. If such testing demonstrates the viability of its approach, several tasks will remain to be completed before ModFetch can provide the full functionality of a database interface.

As outlined, ModFetch only provides means for reading from databases, not writing to them. Additionally, the means of specifying WHERE constraints is somewhat awkward, still relying on manually specifying the literal portion of the SQL query. Table names must be redundantly used, and no means are provided for parameterizing these constraints. These and other similar features can be added, perhaps by adapting the semantics of high-level query frameworks such as Microsoft's LINQ [12].

Additionally, aggregate operations (both grouping and SQL functions such as sum and average) are not yet supported. The approach employed by our framework indicates that supporting these features will be a relatively straightforward task—something as simple as adding to the ReificationOperator class a method `groupBy` that returns a new ReificationOperator. We leave the details of the semantics and implementation of this feature for later work.

When this work is completed, ModFetch may reduce the hassle of using databases as a persistent storage mechanism. While existing schemes for automating persistent database storage focus on solving the "object-relational impedance mismatch", ModFetch instead respects OOP and relational DBs as distinct systems. It focuses on reifying the query itself, which has traditionally been a powerful interface to the database, and can be made more powerful if it is less time-consuming to deal with.

Additionally, ModFetch's focus on the query leads to a system that is simple in its semantics and implementation. It requires no scheme for gluing together two type systems through complicated mechanisms such as extensive configuration files or native language features. It offers the programmer a framework that they can use with minimal setup, minimal overhead, and minimal learning curve. Finally, ModFetch's introduction of modularity to queries may grant the programmer new powers, regardless of hassle, that are not available when using literal string queries.

9 References

[1] R. Elmasri and S. Navathe, "Impedance Mismatch," *Fundamentals of Database Systems*, Addison Wesley, Fifth Edition, 2006, p. 292.

[2] T. Neward, "The Vietnam of Computer Science," The Blog Ride, Ted Neward's Technical Blog, June 26 2006, blogs.tedneward.com.

[3] B. Walker, W. Cook, R. Greene, et. al., "Objects and Databases: State of the Union in 2006," *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006.

[4] B. Wiedermann, A. Ibrahim, and W. Cook, "Interprocedural query extraction for transparent persistence," *ACM Sigplan Notices* 43(10), September 2008, pp. 19-36.

[5] B. Wiedermann and W. Cook, "Extracting queries by static analysis of transparent persistence," *Annual Symposium on Principles of Programming Languages* (8), 2007, pp. 199-210.

[6] Oracle. PL/SQL. http://www.oracle.com/technology/tech/pl_sql.

[7] M. Grechanik, D. Batory, and D. Perry, "Design of Large-Scale Polylingual Systems," *International Conference on Software Engineering*, May 2004.

[8] T. Cormen, C. Leiserson, R. Rivest, et. al., "Topological sort," *Introduction to Algorithms*, MIT Press and McGraw-Hill, Second Edition, 2001, pp. 549-552.

[9] A. Kahn, “Topological sorting of large networks,” *Communications of the ACM* 5(11), 1962, pp. 558-562.

[10] R. Tarjan, “Edge-disjoint spanning trees and depth-first search,” *Algorithmica* 6(2), 1976, pp. 171-185.

[11] Red Hat. Hibernate. http://www.hibernate.org/hib_docs/reference/en/html/index.html.

[12] Microsoft Corporation. The LINQ project. http://msdn.microsoft.com/netframework/future/linq.