

Database: Slave or Master?

Sam Vilain*

October 17, 2006

Abstract

Tangram is built on the premise that any object model, or schema, written in terms of its meta-model may be successfully mapped to a set of database tables. DBIx::Class is built on the premise that it is possible to map from any database schema, or DDL, to a set of Perl classes, using an implicit meta-model.

This paper explores the isomorphism between these two types of database abstraction, traditionally considered to represent an “impedance mismatch” that can not be overcome. Using the semantics of Perl 6, Class identity is related to primary keys as attributes are related to columns; compiled classes are related to tables as instances to rows; parametric collection roles related to foreign key and link table relationships; finally Perl expressions are related to RDBMS query expressions and iterations over expressions related to executing RDBMS queries.

Contents

1	Introduction	2
2	Basic Mapping	3
2.1	Object Identity as Primary Key	3
2.2	Attributes as columns	5
2.3	Inheritance and Roles	5
3	Mapping Relationships	6
3.1	Many-to-One as (Local) Foreign Key	6
3.2	Many-to-Many as Link Table	6
3.3	Relations and Parametric Roles	6
3.4	One-to-Many as (Remote) Foreign Key	7
4	Query Expressions	8
4.1	Junctions for remote objects	8
4.2	Logical Operations on Junctions are still Junctions	8
4.3	Running the queries - evaluating the Junctions	8
5	Conclusion	9
	References	9

*sam.vilain@catalyst.net.nz, Catalyst IT (New Zealand) <http://catalyst.net.nz/>

1 Introduction

Whilst the DBI may be an excellent provider of database driver independence, just about every programmer who starts using the DBI ends up either building their own abstractions to its interface, or using somebody else's. As a result there are a multitude of modules in this space with significant overlap in functionality.

This is at least in part because the typical Perl program is written using such paradigms as *object-oriented programming*, whilst the language of the database is *relational logic*. This leads to developers writing abstractions to the relational logic in a way that suits them, or fits their approach for the task at hand. They end up building *Object-Relational Mappers* (ORMs).

Mapping can happen in either direction; from a *Database Definition Language* (DDL) to Perl classes, or from a set of Perl classes to a DDL. `DBIx::Class::Schema::Loader` demonstrates converting from SQL DDL to `DBIx::Class`'s implicit meta-model, whereas using `Tangram::Schema` you describe Perl classes (perhaps fed from `Class::Tangram` descriptions) so that `Tangram` may map them to a database.

In either case, we desire to follow Don't Repeat Yourself by avoiding specifying the schema twice; once to the database and a second time to the mapping layer. Traditionally, systems that translate from DDL to Perl classes are called *database abstractions* and those that translate from some other meta-model to DDL are *object persistence tools*. However, this distinction is somewhat arbitrary; a tool might even provide both directions of translation.

There are some more interesting differences between database abstraction and object persistence;

- *suitability to object types* – is the tool useful for all object types, or merely those created with the mapping layer in mind, perhaps using a common base class?
- *suitability to database definitions* – can any database schema may be mapped, or merely those conformant with the style of the mapper?

The schism between tools that focus on one or the other is what I am trying to help eliminate, by charting a map from the Perl meta-model to some representation of a relational *Data Definition Language* (DDL), and the entire variety of the DDL to corresponding Perl 6 meta-model components. SQL defines one well-known DDL, but others are possible.

While it may not be possible to cleanly 'round trip' a schema from Perl to DDL and back again without inserting Perl code into the database, the reverse should always be possible. Hence, going from Perl to DDL, back to Perl and back to DDL should end up with the same DDL. I'll call this a *stable* mapping.

I'm going to list recommendations for writing ORM's, for both the DDL-driven and the object model driven camps. For the DDL driven camp, it is an expression of some key relational concepts in object-oriented terms which you can use as a guide for constructing corresponding Perl classes. For the object model driven camp, it is laying down how you need to pepper your classes so that they can be automatically converted to correctly normalized table structures.

I'm not going to talk about everything - views, triggers, rules, stored procedures and sequences will be omitted. They're important, but they should hopefully fit logically into the picture.

So far I have identified these core principles of stable mapping;

- *Object-ness* as a basis for set theory facts – the justification for mapping compiled Classes to tables.
- *Simple Types* of Class and Role attributes are represented as columns; which can be mapped onto the compiled class' table, or separate tables sharing a unique primary key.
- *Complex Types* of Class and Role attributes are represented as (local) foreign keys, being equivalent to regular columns.
- *Collection Classes* such as Sets, Arrays and Hashes can be linked to a common base class using *Parametric Roles* and hence treated alike.
- *Collection* attributes may be represented as link tables, foreign keys in either direction – depending on the type type of the collection, and which constraints are placed on the membership of the collection.
- *Association Classes* are promotions of Collections to first class schema entities. These can either extend the categories of the collection, or add extra properties. They can be made by sub-classing *Collection* classes (or including the *Collection* role).
- *Query Expressions* may be built in terms of relationships between *junctions* representing database-side objects.

2 Basic Mapping

This section describes some basic principles of mapping data types.

2.1 Object Identity as Primary Key

A frequent complaint about Object-Relational mappers is the use of *Surrogate IDs* on all tables. This is a valid complaint; after all, a row in a table is supposed to represent a *fact*: an association between some data points. People performing *normalisation* of database schemas ask questions of tables like, “what fact does this row represent?”

Our basic answer is, “an object of type X exists with this set of attributes”. This might seem like skipping the question, but I think a better way of looking at it is “Ask a meta-question, get a meta-answer”.

That is, in order to answer the question appropriately in a manner that won't ring normalisation alarm bells, you must fill in the gaps in the answer for individual types in your object schema. Such as “there exists a CD with this unique distributor and catalogue number combination, with this title, and this artist, etc.” When the ORM must store objects that provide no hint about the nature of their identity, it must use surrogate IDs.

It turns out that surrogate identifiers are used internally by the database anyway. It is usually hidden, and often called something like `rowid`. The `rowid` is typically not required for *index-organized tables*. By analogy, then, it is not a fault of an ORM to add surrogate ID column, particularly if those surrogate IDs are also hidden (ie, are

not needed to deal with the objects). If they are specified, then a primary key should be defined.

Objects should correctly define a `.WHICH` method (a Perl 6 convention) that returns either the correct single primary key, or something that `.does(Seq)` in the case of multiple-part primary keys. Additionally, the type of this `.WHICH` return value should be declared in the declaration of the `.WHICH` method using the `of` keyword, to make the type of the primary key available in the meta-model.

```
1
2 # simple example: a phone endpoint uniquely identified by
3 #           its extension.
4
5 subset Phone::Number of Str where /^\d**{0,4}$/;
6
7 class Phone {
8     has Phone::Number $.extension;
9     method WHICH of Phone::Number {
10         $.extension;
11     }
12 }
13
14
15 # multi-part key example: CDs identified by record label
16 #           and catalogue number
17
18 class CD {
19     has Label $.distributor;
20     has Label::CatNum $.catalogue_number;
21
22     method WHICH of Seq[of => :(Label, Label::CatNum)]
23     {
24         ($.distributor, $.catalogue_number);
25     }
26 }
```

Don't make the code fragment in your `.WHICH` too complicated; the ORM might have to inspect the syntax tree of the code to figure out which column refers to which attribute, if it can't figure it out in any other way. This required code introspection I considered to be the biggest hack of this specification, but a tolerable one, especially given that `.WHICH` already has some pretty hefty restrictions.

Instead of using a surrogate row ID, you could use the *entire object* as the 'primary key' - this would certainly sit better with Set Theory, and get rid of that 'duplicate row problem' that drives the theorists crazy.

However, much as it would be unworkable if an RDBMS were to require all tables without primary keys to have an implicit primary over all of its columns, defining `.WHICH` so that it would include all the properties of the object would have unexpected side effects for most people. You would effectively make a *value type* or *immutable type* (as defined in Synopsis 12), and therefore the result of changing any object properties is undefined. Depending on the VM implementation, existing variables may still contain

the old object, or there might be other confusing side effects. So, much as Perl 6 will assign an arbitrary value for `.WHICH` by default, the ORM must attach a surrogate ID by default, too, if it wants to store such objects.

2.2 Attributes as columns

Next to the primary key(s), additional object properties can be translated to individual columns.

A stable ORM should make use of the *type constraint* of the attribute to choose a database type to map to. Any attributes that do not have a type constraint defined at all are fair game for storing using a proprietary storage mechanism, similar to that employed by `Tangram::Type::Dump::Any`. Such a schema can not be called ‘normalised’ and will not be stable.

With a rich enough representation of type constraints, and enough use of type constraints through the program, it should be possible to map to all common database column types. This is usually more difficult if the constraints are represented as `Code` objects, in the absence of LISP-style macro-manipulation of the compiled opcode tree.

Using `Moose`, you can provide objects to the `where` clause, instead of the code block recommended by the Perl 6 specification. This sort of practice should be encouraged; using Perl code blocks for constraints presents a formidable task for an ORMs to reverse engineer.

Simple enough types may have directly corresponding database types; date and time types are examples of this.

2.3 Inheritance and Roles

Inheritance can be modelled as tables that share a unique, primary key. The objects are the same; each table defines which attributes the extra subclasses add.

It should be possible, based on the combination of tables in which the primary key exists, to determine the type of the combined object. If this is not possible due to the table types being the same, a *surrogate type identifier* must be used. `Tangram` does this, but indiscriminately.

When retrieving or building queries of objects based on a base type, you must use *polymorphic retrieval*. In a query, an `outer join` is performed based on the primary key to combine the various tables sharing an `isa` relationship. This is called *filtered mapping* by `Tangram` in the documentation file `Tangram::Relational::Mappings`.

The alternative layout (and, perhaps, correctly normalised) is to use wide tables and `NULL` columns, which is called *horizontal mapping*. Horizontal mapping is essential for databases like `MySQL` (which returns bizarre results as soon as you start using nested joins), or `SQLite` (which refuses to answer such complex queries).

Roles present a similar situation; their storage requires an extra collection of columns per-object. Because there is no unique relating element between their columns, the columns should instead be attached to the table that holds the class they were composed into.

In principle, this could work by including a single column in each of the classes that the role is composed into, that relates to a surrogate ID around the collection of role attributes. However, most databases will not be able to enforce referential integrity with a constraint in this case, so it should be avoided.

3 Mapping Relationships

So far, I have only discussed the points that most people seem to agree upon; some rough equivalent of matching up classes with tables, and attributes with columns. However I have only dealt with mundane associations; those of individual objects between their properties. What about references between *complex* objects?

3.1 Many-to-One as (Local) Foreign Key

The humble reference is the simplest example of an object property. A reference in the meta-model is much like a * (whatever) to one (or zero) *relation*.

Lazy loading can be used to allow attributes that are references to other classes to be implemented by placing a foreign key column (or columns) in the source table, and deferring the actual `select` until as late as possible. Developed mapping layers will also support *pre-fetching* to avoid excessive and inefficient database round-trips.

3.2 Many-to-Many as Link Table

Collections with a higher *destination n-ity* than 1 (eg, a Hash or Array) should be mapped to a link table - unless it can be proven that they are the reverse of a relationship with a destination n-ity not exceeding 1 (see below). The link table must also have a *category* (represented as an extra column, a part of a unique key) to represent either the order of items in a collection (an INTEGER column), or the hash key (a TEXT or similar). The Set class sits alongside Array and Hash as an unclassified relationship (no extra column), and an ordered hash (such as the `Tie::IxHash` module) is one with both an integer and a string category.

3.3 Relations and Parametric Roles

Array and Hash seem to have common characteristics - are they really separate fundamental types deserving special treatment?

Perl 6 defines a very useful feature that I will use to demonstrate unification of its collection types; *parametric roles*.

The following listing shows a selection of 'core' Perl 6 types, along with their possible foundation in a single root `Collection` type, which can be mapped directly to a table (as with the basic types of objects dealt with earlier). This is an abstraction; it matters not that the actual interpreter does not use these definitions.

```
1  role Collection[\$types] {
2      has Seq[of => \$types] @.members;
3  }
4
5  role Set[::T = Item] does Collection[T] where {
6      all(.members) === one(.members);
7  };
8
9  role Pair[::K = Item, ::V = Item] does Seq[of => :(K,V)] {
10     method key of K { .[0] }
11     method value of V { .[1] }
12 };
```

```

13
14 role Mapping[::K = Item, ::V = Item] does Collection[Pair[K,V]] {
15     all(.members).does(Pair) and
16     all(.members).key === one(.members).key;
17 }
18
19 role Hash[Str ::K, ::V = Item] does Mapping[K, V]
20     where { all(.members).key == one(.members).key }
21 {
22     method post_circumfix:<{ }> (K $key) of V|Undefined {
23         my $seq = first { .key == $key } &.members;
24         $seq ?? $seq.value :: undef;
25     }
26 }
27
28 role ArrayItem[::V = Item] does Seq[of => :(Int, V)] {
29     method index of Int { .[0] }
30     method value of Item { .[1] }
31 };
32
33 role Array of Collection[ArrayItem]
34     where { all(.members).index == one(.members).index }
35 {
36     method post_circumfix:<[ ]> (Int $index) of Item {
37         my $seq = first { .index == $index } &.members;
38         $seq ?? $seq.value :: undef;
39     }
40 }

```

The Hash-like and Array-like methods such as `post_circumfix:*` (lines 22 and 36) are defined in the appropriate roles. This is how they can appear to be different, yet share a common role.

3.4 One-to-Many as (Remote) Foreign Key

You can look at any relationship from both directions. Is that a person dreaming of a butterfly, or a butterfly being dreamed about by a person? The only difference between the one-to-many database representation and the many-to-one are that the foreign key is either *local* (yeilding many-to-one) or *remote* (yeilding one-to-many).

The difficulty with mapping relationships to a remote foreign key is knowing that it has a *source n-ity* of 1 in the first place - you need to find a corresponding, reversed collection on the target class with a *destination n-ity* of 1. For instance, they may be a simple attribute and not a collection, which looks enough like a `Collection` with a subset constraint limiting the number of items to 1 for us to treat it like that for now.

In the `Class::Tangram` documentation, these corresponding, reversed pairs of collections are referred to as *companion attributes*. This companion nature must be specifically annotated; without help, there is no way to tell that a relationship going back ‘the other way’ to the one you are looking at is a corresponding reverse relationship. Such features fit under the auspices of derived `Container` types that send

messages to (ie, call methods on) removed / added objects in the collection to notify them of the change in state, so that they may update their own back-references. The *reference* and the *collection* objects should share a language for passing messages notifying each other of changes to their state, to maintain consistency. This means that both `Item` and `Collection` will require methods to be added.

This situation is not really so dire, an alternative is to ensure that a given program always accesses relationships between types in one direction. This is simpler, usually faster, and doesn't limit your ability to use the relationships in queries. Tangram demonstrated this by adding read-only, demand loaded *back-references* on one-to-many relationships (see the documentation for `Tangram::Type::Set::FromOne`, particularly the `back` property). One they can be changed, or their values may be cached, the 'companion' system of message passing becomes necessary.

From this vantagepoint, it appears that typed collection roles may provide a solid foundation for a stable mapping between the Perl 6 meta-model and a relational database.

4 Query Expressions

Query expressions are a difficult issue. Nobody seems to want to write SQL for everything, but people still want the full power of SQL available to them. Why is it that most database abstractions leave wide gaping sections of SQL unable to be generated without cumbersome manual SQL fragments?

4.1 Junctions for remote objects

In Tangram, one uses the `Tangram::Storage->remote` method to return a *remote object*; this object represents *an object in the database*. In a sense, it is an *abjunction* of all of the rows in the database. It represents exactly one object at a time.

Similarly, the `DBIx::Class::ResultSet` object represents a *conjunction* of all of the rows in a table (or whatever the result set refers to).

These basic abstractions are powerful tools, providing the basic constructs used for all SQL expressions: table aliases. Relational features such as aggregates, joins, and subqueries may all be expressed cleanly and expressively using this abstraction, as demonstrated by the range of queries in the Tangram test suite.

4.2 Logical Operations on Junctions are still Junctions

When you perform logical operations such as booleans on junctions, a junction is returned. Analogously, the result of using boolean operations or equality on a database result set object is a junction which represents the solution to the problem given.

Result set abjunctions have slightly different properties to regular Perl 6 junctions. For instance, a table alias appearing multiple times in a query must always refer to a single value at a particular instant. The similarity is drawn here not because they should be implemented using junctions, but because they are conceptually very similar.

4.3 Running the queries - evaluating the Junctions

Passing a set of query expressions to a database handle, and asking for an iterator over the members of the set, is the point where the query is run.

In `Tangram`, this happens when you pass the `Tangram::Expr` and `Tangram::Expr::Filter` objects you construct via `Tangram::Storage->remote()` returned objects to a function such as `Tangram::Storage->select()` or `Tangram::Storage->cursor()`.

In `DBIx::Class`, this happens when you call a method returning an iterator on a `DBIx::Class::ResultSet` object.

5 Conclusion

The differences between Object-Oriented Programming and Relational Logic are substantial, and require a developed understanding in order to design an Object-Relational Mapper that does not make excessive design mistakes, undermining the benefits of Set Theory and the Relational Database model. However, the key point is that all the fundamental relational concepts can be mapped to an object meta-model, and round-tripping from any DDL to a Perl 6 meta-model and back again without loss of information should be possible. I hope that this paper can serve as a starting blueprint for such developments.

Finally, it has come to the point where such concepts can be expressed in terms of the tools that grew from both the pure Relational and pure Object Oriented back-ground. Both Jean-Louis Leroy's fantastic `Tangram` module (my own additions - 'companion attributes' aside, generally aided the push away from normalisation principles), and Matt S. Trout's (and the community's) `DBIx::Class` have the necessary abstractions to make this kind of comparison possible.

References

- [1] Leroy, *Tangram - Mapping Inheritance*.
<http://tangram.uts1.gen.nz/docs/Tangram/Relational/Mappings.html>
- [2] Trout, *DBIx::Class::Manual::Example - Simple CD database example*,
<http://search.cpan.org/dist/DBIx-Class/lib/DBIx/Class/Manual/Example.pod>
- [3] Conway and Wall, *Perl 6 Synopsis 6 - Subroutines*. In particular the references to `.WHICH` in the definition of value types.
http://dev.perl.org/perl6/doc/design/syn/S06.html#Immutable_types
- [4] Larry Wall, *Perl 6 Synopsis 12 - Objects*. In particular the sections on Roles and their parameters
<http://dev.perl.org/perl6/doc/design/syn/S12.html#Roles>