Inference-based SQL Programming

By: Brian Rosenthal and Zvi Boshernitzan

I. Overview

The idea behind inference-based SQL programming is to have one class that will allow you to have a programming interface to all of your data entities that is readable, flexible, and powerful, while writing the minimal amount of code for each entity.

It emerged from the idea that there is a lot of information in the database itself that could be used to infer things about what sorts of functions a developer might want to call.

For example, take the following schema:

products (foreign keys for category and brand)
keywords (many-to-many relation between product and keyword)

If you were writing classes to manage these database entities, you really should get a lot of functions for free.

```
products::get(product_id)
products::create({‘name’: ‘12 inch valve’, ‘category_id’:2, ‘brand_id’:100})
products::mget({‘category_id’:2})
products::mget_by_category(...)
products::mget_by_brand(...)
products::mget_by_keyword(...)
products::map_keyword(product_id, keyword_id)
keywords::mget_by_product(...)
products::get_name(product_id)
products::colnames()
```

We asked the following question:

What if we could write a class that would allow all of these functions to exist without ever being written?

Enter PHP 5

This technique heavily relies on the following features of PHP 5:

1. Call handler override.

PHP 5 allows a class to override the default call handler, in the case that a function is not defined. Therefore, it is possible to accept function names that use words that are specific to the entity.

2. MYSQL reflection

The following information is useful to obtain from the database:

1. column names
2. column types
3. foreign key detection
4. required versus not required columns
5. primary key name
6. unique column names
7. enums for the entity

3. Singleton / factory semantics

Because excess database queries may slow down an application, it makes sense to use singleton classes for the management of database entities.

Singleton semantics is the idea of using instantiated classes, but only having one class per page request.

This is implemented by using a function (say, we call it “mg” for management) which itself keeps a static variable which is an associative array of classes.

So, mg(‘products’) would instantiate a “products” class the first time it is called, but the second time it is called, it would just return the same instance.

4. Autoloads

The autoloads feature of PHP5 is particularly useful with this architecture. PHP 5 exposes a special function called “__autoload($classname)” which is called if a class cannot be found.

This function can check for the class in a particular directory so that you don’t have to include every class in each PHP script; they can be dynamically bound at runtime.

We have found this fast and efficient, and a substantial savings in our own coding time.

II. Class: dbentity

The idea here is to have a class that is inherited by other entities that may customize the implementation of the class. The dbentity class contains the logic for common database queries and manipulation.

```
class dbentity {
        ....
}
```

A few notes:

1. All functions of the dbentity base class can be overwritten, and there are some functions which are more specifications than functions.

2. There is no reason at all to define a class for any table. The idea here is to have mg('products') generate an instance of a class for which function calls use the table *products.* If the behavior is very predictable, there should be no need to define the products class at all.

However, you may want to specify relationships, or have custom functions so the idea is that you would define a class:

```
class products extends dbentity {
        function references() {...}
        function mappings() {...}
        function sort_key() { ... }
        function custom_function(...) {...}
}
```

3. Most functions that accept a single dictionary can also accept a list of arguments, and create a dictionary intelligently.

For example, create takes a dictionary, and could look like all of these:

Just a list of arguments, key,value,key,value...
create('first_name', 'Brian', 'last_name', 'Rosenthal', 'hometown', 'Houston')

Just a dictionary:
create({'first_name':'Brian', 'last_name': 'Rosenthal', 'hometown':'Houston'})

A combination:
create({'first_name':'Brian'}, 'last_name', 'Rosenthal', {'hometown':'Houston'})

III. dbentity Interface

1. get(id, args)

Return Value: object with properties based on the fields of the entity.

Takes an id or associative array with a set of unique columns and returns an object with the entity or null if no entity exists matching the condition.

Example
Get product id 3
mg('products')->get(3)

Args can pass in a dbquery structure that would include joins:
get(id, 'dbquery', {ijoin:...})

2. create(args)

Return Value: id of newly inserted entity.
Throws exception on error.

Create takes an associative array and returns the inserted id.
It also:

(a) Enforces referential integrity
(b) Takes care of any "sorting" requirements (assuming there is a sort column)
(c) Validates the arguments
(d) automatically adds mappings if specified
(e) Provides hooks for create_validate...

Example: Create a product
$product_id = mg('products')->create({'name':'Valve'})

3. edit(id_or_where_conditions, update_args)

Return Value: true
Throws exception on error.

Takes an id and an associative array and executes an update statement with the args on the id.

This can be called as follows:
mg('products')->edit($product_id, 'name', 'Valve');
mg('products')->edit($product_id, {'name':'Valve'});
mg('products')->edit({'sku':'RB001'}, 'name', 'Valve');

(notice the unique column sku can specify a product, also, but this requires a unique constraint on sku)

4. delete(id)

Return value: true.
Throws exception on error.

Deletes the entity.

It also:

(a) Fails if any dependent entities exist.
(b) Deletes any mapping relationships
(c) Adjusts sort keys
(d) May cascade, if specified.

Example:
mg('products')->delete($product_id);

Alternate syntax:
mg('products')->delete('name', 'Valve')

5. get_{fieldname}(id)

Return value: field value
Throws exception on error.

Example:

mg('products')->get_name($product_id)

6. get_{parent_entity_name}(id)

Return value: parent entity of the specified entity

Example:

mg('products')->get_brand($product_id)

7. get_{parent_entity_name_plural}(id)

Return value: child entities of the specified entity

Example:
mg(‘brands’)->get_products($brand_id)

8. mget_by_{parent_entity_name}(id)

Example:
mg(‘products’)->mget_by_brand($brand_id)

9. get_by_{unique_column_name}(column_value)

Example:
mg(‘products’)->get_by_sku(‘S1001’);

10. has_{mapping_entity_name}(id, mapping_entity_id)

Example:
mg(‘products’)->has_keyword(45)
mg(‘products’)->has_keywords(array($keyword_id_1,$keyword_id_2,$keyword_id_3));

11. n_{child_or_maping_entity_name_plural}(id)

Example:
mg(‘brands’)->n_products($brand_id)

12. map_{ maping_entity_name}(id, mapping_id)

Example:
mg(‘products’)->map_keyword($product_id, $keyword_id)

13. set_{maping_entity_name_plural}(id, array(mapping_entity_ids))

Example:
mg(‘products’)->set_keywords($product_id, array($keyword_id_1, $keyword_id_2))

14. unmap_{maping_entity_name}(id, mapping_id)

Example:
mg('products')->unmap_keyword($product_id, $keyword_id)

15. unmap_all_{maping_entity_name_plural}(id)

Example:
mg('products')->unmap_all_keyword($product_id)

16. {enum_name}_name(index)

Example:
mg('products')->status_name($active_status_index)

17. {enum_name}_id(name)

Example:
mg('products')->status_id('active')

18. MYSQL reflection

We also expose the following functions:

mg('products')->colnames()
mg('products')->cols() (which returns more information than just the names)
mg('products')->enum_get_as_dict($field_name)
mg('products')->enum_name_to_id($field_name, $enum_name)
mg('products')->enum_id_to_name($field_name, $id)
mg('products')->required_colnames()
mg('products')->unique_colnames()
mg('products')->col_type('colname')

IX. Usage of dbentity class

Overview

The goal of the dbentity class was to require nothing to use it with a general database table, other than the table must be named a "plural" name and it must contain a field called "id".

The "mg" function

Core to the dbentity architecture is the idea of automatic object generation.

Using the dbentity class without inheritance

Without any sort of extra specification, as long as the database with which you are working has a table called “products” with a field “id”, this script...

```
mg(‘products’)->get($product_id)
```

... will return an object of type “generic” with properties based on the structure of the “products” table.

IV. Using the dbentity class by creating an entity class

You may also create a class that inherits from dbentity.

```
class products extends dbentity { ...}
```

This class may alter the behavior of the mg(‘products’) class by specifying other things about the entity.

1. references()

The point of the function “references” is to specify the entity-type of foreign references within the table.

The default implementation is returning an empty array. You override this:

```
class products extends dbentity {
	function references() {
		return array('brand_id' => 'brands', 'category_id' => 'categories');
	}
}
```

This allows the mg(‘products’) class to now know that the columns “brand_id” and “category_id” both refer to “categories” and “brands” respectively.

Now, you can call:

```
mg(‘products’)->get_brand($product_id)
mg(‘products’)->get_category($product_id)
mg(‘products’)->mget_by_category($category_id)
mg(‘products’)->mget_by_product($category_id)
```

2. *foreign references()*

Foreign references specify columns in other entities that reference this entity type.

Example:

```
// here, customers is referenced by sales orders, in the “customer_id” column.
class customers extends dbentity {
        function foreign_references() {
                return array(
                        'sales_orders'     => array('sales_orders','customer_id')
                );
        }
        ...
}
```

Now, you can call functions on the mg(‘products’) object like:
mg(‘customers’)->get_sales_orders($customer_id)

3. *sort_key()*

The point of the function is to specify the sort keys that should be re-sorted on insertions, updates, and deletes.

This function specifies that there is a column “sort_order”, and that products are sorted for a particular category_id. (so within each category_id, the “sort_order” of products will be numbered 0,1,2,3)

Example:

```
// there is a sort key on “sort_order” for particular categories
class products extends dbentity {
        function sort_key() {
                return array('sort_order', array('category_id'));
        }
        ...
}
```

Now, the sort key for the product will keep itself sequential for any particular category.

4. *mappings()*

A specification-type function defined by the class, if there are mapping relationships.

```
Returns an dictionary {‘mapping_name’:
                        [‘mapping_table_name’, ‘near_key’, ‘far_key’, ‘far_entity_name]}
```

So, for example, if there was a table called “product_keywords” which mapped products to keywords, the function would look like this:

```
function mappings() {
	return array(
		'keywords'
		=> array('product_keyword_map', 'product_id', 'keyword_id', 'keywords')
	);
}
```

Now we can call:

```
mg(‘products’)->get_keywords($product_id)
mg(‘products’)->n_keywords($product_id)
mg(‘products’)->set_keywords($product_id, $keyword_ids)
mg(‘products’)->unmap_keywords($product_id)
mg(‘products’)->map_keyword($product_id, $keyword_id)
```

We can also have reflective mapping relationships. Consider related products:

```
class products extends dbentity {
	function mappings() {
		return array(
			'related_products'
			=> array('product_relations', 'from_product_id', 'to_product_id',
		'products')
		);
	}
}
```

5. logged_colnames()

specifies an array of column names for which changes are logged in a history table.

6. rename_keys($arr)

Specified a mapping relationship of fields that are renamed on inputs to most functions

This allows arguments to be passed in with multiple names.
	“category” or “category_id” should really mean the same thing.
Rename keys is called during most function calls, so its main purpose is to be specified if necessary.

```
function rename_keys($arr) {
	rba::rename_keys($arr, {‘category’ => ‘category_id’})
	return $arr;
}
```

7. to_string($id_or_obj)

Function specified by the instance class which takes an id or object and returns a string that may be printed to the user.

V. Summary

Inference-based SQL programming can help you reduce the amount of code in your data tier and make your application quicker to develop and easier to manage.

It is meant to assist with the “model” layer of your application of an MVC framework, and if you wish to have access privileges or interfaces for the management of your entities, those are meant to be contained in separate classes.