

Zend_Db_Table relationships - Bill Karwin

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

Zend Framework: Zend_Db_Table relationships Component Proposal

Proposed Component Name	Zend_Db_Table relationships
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend_Db_Table relationships
Proposers	Bill Karwin
Revision	1.1 - 17 January 2007: initial proposal. 1.2 - 22 February 2007: write section on cascading UPDATE and DELETE. (wiki revision: 10)

Table of Contents

1. Overview
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
4. Dependencies on Other Framework Components
5. Theory of Operation
 - Storing information about references
 - Querying based on relationships
 - Belongs_to relationships
 - Has_many relationships
 - Many-to-many relationships
 - Has_one relationships
 - Inheritance use case
 - Column-partitioning use case
 - Solution for has_one relationships
 - Cascading write operations
 - Cascading DELETE
 - Cascading UPDATE
 - Cascading INSERT
 - Entity Lifecycle
 - Auto-discovery of relationships
6. Milestones / Tasks
7. Class Index
8. Use Cases
9. Class Skeletons

1. Overview

This is a proposal to extend Zend_Db_Table and related classes Zend_Db_Table_Rowset and Zend_Db_Table_Row, to model relationships between relational database tables. The goal is to provide a simple and convenient API for developers to query data from related tables, given an instance of a row from one table.

This solution models relationships between tables in the RDBMS, and provides methods to retrieve rows based on "belongs_to", "has_many", and "has_one" relationships between tables.

The "belongs_to", "has_many" and "has_one" terms are based on Ruby on Rails parlance.

2. References

- [Ruby on Rails associations](#)
- more references to be provided

3. Component Requirements, Constraints, and Acceptance Criteria

- This component **will** model relationships between tables.
- This component **will not** require use of configuration files.
- Given a `Zend_Db_Table_Row` object instance for a master table, the object **will** have a method that returns a `Zend_Db_Table_Rowset` for matching rows in each dependent table. This models the one-to-many or "has_many" relationship.
- Given a `Zend_Db_Table_Row` object for a dependent table, the object **will** have a method that returns a `Zend_Db_Table_Row` for the matching row in its master table. This models the many-to-one or "belongs_to" relationship.
- Given a `Zend_Db_Table_Row` object for a table that is part of a many-to-many relationship with another table, the Row object **will** have a method that returns a `Zend_Db_Table_Rowset` object for the matching rows in the other table. This uses the intersection table that is necessary to implement the many-to-many model.

4. Dependencies on Other Framework Components

- `Zend_Db`
- `Zend_Exception`

5. Theory of Operation

Storing information about references

Classes derived from `Zend_Db_Table` must store information about its declarative referential integrity (DRI) to other tables. Each foreign key must be stored in the class corresponding to the database table that contains the foreign key.

It is important that the information about DRI relationships is declared in only one class. If both the class containing the foreign key and the class containing the referenced primary key included the information, then they could become out of sync.

The reference information is stored in a protected static array. The array is an associative array, mapping strings which identify each DRI constraint to specific information about the constraint. This information includes the column(s) in the dependent table, the name of the master table, and the matching column(s) in the master table. For example:

```

class LineItems extends Zend_Db_Table
{
    protected $_referenceMap = array(
        'Order' => array(
            'columns' => array('order_id'),
            'refTable' => 'Orders',
            'refColumns' => array('order_id')
        ),
        'Referer' => array(
            'columns' => array('referer_order_id'),
            'refTable' => 'Orders',
            'refColumns' => array('order_id')
        ),
        'Product' => array(
            'columns' => array('product_id'),
            'refTable' => 'Products',
            'refColumns' => array('product_id'),
        )
    );
    ...
}

```

The above example is for a class `LineItems`, which contains three foreign keys: two foreign keys reference to the `Orders` table, and one references the `Products` table.

The `Zend_Db_Table` class has a public method to return information about a reference. `getReference($tableName, $ruleKey = null)`, returns the element of the `_referenceMap` that refers to the specified table.

In cases where there is more than one foreign key to a given table, the first element of the array that matches the specified table is returned. The optional second argument specifies the key in the `_referenceMap`, so the user can query secondary foreign keys to the same table.

Querying based on relationships

Belongs_to relationships

The "belongs_to" or many-to-one relationship is the case where a dependent table contains a foreign key referencing the primary key of a master table. It is said that the dependent table "belongs to" the master table, as this relationship is commonly used to model a multi-valued dependent attribute of a master table.

A dependent table may have multiple foreign keys that reference different master tables. So retrieving rowsets from master tables must include some identification of the master table.

A dependent table may have multiple foreign keys that reference potentially different values in the primary key of the same master table. So retrieving rowsets from the master table must include some identification of the columns in the dependent table used for the foreign key reference.

The `Zend_Db_Table_Row` class has a method `findParentRow()`, which returns a `Zend_Db_Table_Row` object for the row in the master table with the matching primary key.

The `Zend_Db_Table_Row` class already contains a protected reference to its source `Zend_Db_Table` object. Using this, it can query the information about the correct reference, based on the information in the `_referenceMap`. `findParentRow()` can instantiate the referenced table class, and create a correct `$where` argument to query the rows matching the values in the foreign key of the current `Zend_Db_Table_Row` instance.

If no such relationship exists between the two tables, then the method throws an exception.

Has_many relationships

The "has_many" or one-to-many relationship is the reverse of "belongs_to". A master table has a primary key and it "has many" rows in a dependent table, which reference the master table by using a foreign key.

A master table may have multiple dependent tables. So retrieving dependent data must include some identification of the dependent table.

A dependent table may have multiple foreign keys referencing potentially different values in the primary key of the master table. So retrieving dependent data from such a table must include some identification of the columns in that table that form the foreign key.

The `Zend_Db_Table_Row` class has a method `findDependentRowset()`, which returns a `Zend_Db_Table_Rowset` object for the row in the dependent table with the matching foreign key.

The `findDependentRowset()` method can instantiate the referenced table class, and query its `_referenceMap`. Then it can query that table with an appropriate `$where` clause.

If no such relationship exists between the two tables, then the method throws an exception.

Many-to-many relationships

A many-to-many relationship is really a relationship between three tables. Two main tables, and one table in the middle. The middle table contains foreign keys referencing the primary keys of each respective main table.

A given table may belong to multiple many-to-many relationships with different intersection tables. So retrieving matching data from the other side of the relationship must include some identification of the intersection table, and of which columns in that intersection table are used for referencing each of the main tables.

The `Zend_Db_Table_Row` class has a method `findManyToManyRowset()`, which returns a `Zend_Db_Table_Rowset` object for the row in the dependent table with the matching primary key. The intersection table must be specified. The `findManyToManyRowset()` method instantiates the intersection table, queries the `_referenceMap` and retrieves a rowset from the intersection table. From this, it finds values to use in a `$where` clause to query the other table in the many-to-many relationship. If `Zend_Db_Table` supports joins or query by SQL (to allow us to use subqueries or joins), we can make this operation more efficient.

Has_one relationships

The "has_one" relationship is curious. This occurs when a foreign key in the dependent table is also its primary key. Therefore there is at most one row in the dependent table referencing the master table.

Neither of the use cases below are part of any logical relational model; the first is an object-oriented model, and the second is used for physical optimization.

Inheritance use case

Has-one relationships are sometimes used to model object-oriented inheritance.

If table B extends table A in an object-oriented fashion, then table B should have the attributes of table A, plus additional attributes. Store the attributes from the "superclass" in table A and only the extended attributes in table B. Thus every row in table B corresponds to exactly one row in table A, and the combination of these two rows forms one object instance of class B.

This design approximates some OO characteristics, in that alterations to the schema of table A are implicitly inherited by the table B model.

Column-partitioning use case

Has_one relationships can also be used to physically partition columns of a table.

For instance, if a table T contains a few compact columns (like integers and dates) that are queried frequently, and the same table also contains some bulky columns (like varchar and blob) that are queried infrequently, there is some performance advantage to separating these two sets of

columns into different tables, with a one-to-one relationship between them.

The advantage is based on some RDBMS implementations in which I/O efficiency or caching efficiency is benefited by smaller row size. A single disk read can read multiple rows if the rows are small. A fixed cache size can store more rows (perhaps even the entire table) if the rows are small.

Solution for has_one relationships

- TO BE WRITTEN

Cascading write operations

Model cascading operations, like those provided by declarative referential integrity clauses `ON UPDATE` and `ON DELETE`.

Cascading DELETE

If a row in a parent table is deleted, all dependent rows in the database should be deleted. This requires that the parent table's class have some declaration of which tables contain referencing foreign keys to the parent.

```
class Orders extends Zend_Db_Table
{
    protected $_dependentTables = array(
        'LineItems'
    );
}

class LineItems extends Zend_Db_Table
{
    protected $_referenceMap = array(
        'Order' => array(
            'columns' => array('order_id'),
            'refTable' => 'Orders',
            'refColumns' => array('order_id'),
            'onDelete' => self::CASCADE,
            'onUpdate' => self::CASCADE
        ),
    );
}
```

When the `delete()` method of a `Zend_Db_Table_Row` object is called, it checks its table class and gets a list of each table listed in the protected array `$_dependentTables`. Then it calls a method in that class to invoke

Objects of type `Zend_Db_Table_Row` may exist in the application, containing data previously fetched from the dependent table, data which are now deleted as a result of the cascading delete operation. These objects are *not* synchronized with the database; they represent phantom data that does not exist anymore in the database, and they have no automatic indication of this state.

The PHP application must delete rows in dependent tables first, before it deletes a row in the parent table.

If the RDBMS schema also contains DRI for an `ON DELETE CASCADE` rule, there is no conflict. Because the dependent rows are deleted explicitly before the row in the parent table is deleted, the cascading delete implemented in the server simply deletes zero rows.

Note that currently, `Zend_Db_Table_Row` has no `delete()` method.

Cascading UPDATE

If the primary key of a row in a parent table is changed, all dependent rows that refer to this primary key value should be updated to match. Like the cascading `DELETE`, this also requires that the parent table's class have a declaration of dependent tables. This is done in exactly the same

way as shown in the case of Cascading DELETE above.

When the `save()` method of a `Zend_Db_Table_Row` object is called, and the primary key column(s) are among the columns that have been modified, then a similar process is done to that in the DELETE case. The table(s) listed in `$_dependentTables` are consulted, and through the `$_referenceMap`, dependent rows are updated.

To preserve referential integrity, the update of the primary key of the parent table and the update of foreign keys of dependent tables must be done atomically; referential integrity constraints in the RDBMS server should be applied in a deferred way, or else the RDBMS will prohibit the update of dependent records. If the RDBMS does not support deferred constraint enforcement, then the schema must not be designed to enforce the RI at all, or else non-atomic cascading UPDATE performed by `Zend_Db_Table` will cause a violation in the RI enforcement.

If the RDBMS schema is designed to enforce the RI using constraints, then cascading UPDATE must also be performed by the RDBMS. The developer should declare `'onUpdate' => self::CASCADE` in the `$_referenceMap` of dependent tables **only** if the RDBMS schema does not enforce RI for these relationships.

Fortunately, cascading UPDATE is a fairly uncommon operation. There is no reason to change the primary key value of a pseudokey (such as an auto-generated integer `id` column), so this typically is used only with natural keys.

Note also that currently in `Zend_Db_Table_Row`, changing the values of a primary key is not permitted; it throws an exception. This will have to change.

Also note that currently `Zend_Db_Table_Row` does not track the original data values, but only the data values that may have been modified by the application. So if we permit the application to change the primary key values, we must preserve the original values internally, so that we can use them to update the correct rows in dependent tables.

Cascading INSERT

There is no such thing as a cascading INSERT. If I create a new record, there is no way the DRI knows what, if any, rows to create in dependent tables. New rows in dependent tables must be created explicitly, after the row to which they refer is created.

Entity Lifecycle

Some ORM technology implements the concept of entity lifecycle. That is, objects in application memory track changes to attributes, and therefore know when they are "dirty" and need to be posted to the database. They also track any dependent objects that have been fetched from the database and exist as objects in the application space, and therefore know to keep such objects in sync with their state in the database.

The feature of entity lifecycle management is beyond the scope of Zend Framework 1.0 and will not be addressed with this proposal. It might be addressed in future versions of Zend Framework.

Auto-discovery of relationships



Auto-discovery of metadata may be added as a future enhancement of `Zend_Db_Table`. In the short term, only explicit declaration of DRI information is supported.

If the user's class derived from `Zend_Db_Table` does not include an explicit declaration of the relationships in the `_referenceMap` array, then the class can query the database adapter for metadata information, and may cache this information.

This information should be based on standard system views conforming to the `INFORMATION_SCHEMA` where these views are available. Of the databases currently supported, here is the status of support for `INFORMATION_SCHEMA`:

RDBMS	Foreign Keys	INFORMATION_SCHEMA
PostgreSQL		use <code>pg_catalog</code> views
Oracle		use nonstandard views

IBM DB2	✔	⚠ use SYSCAT views
MS SQL Server	✔	✔ since circa 2000
MySQL	✔ except INNODB	✔ since 5.0
SQLite	✘	✘

6. Milestones / Tasks

Milestone 1: [DONE] Publish proposal.

Milestone 2: Revise proposal, approve for Incubator development.

Milestone 3: Commit working prototype to Incubator.

Milestone 4: Commit working unit tests for one RDBMS back-end.

Milestone 5: Write end-user documentation.

Milestone 6: Release prototype in Zend Framework 0.8 Preview Release.

Milestone 7: Revise implementation, tests, and documentation based on feedback.

Milestone 8: Merge changes from Incubator to Core in Zend Framework 0.9 Beta Release.

7. Class Index

- Zend_Db_Table
- Zend_Db_Table_Row
- Zend_Db_Table_Rowset

8. Use Cases

The use cases below are based on a schema for an order-processing database. Here is pseudocode for the SQL schema:

```
CREATE TABLE Customers (
    customer_id      PRIMARY KEY
)

CREATE TABLE Orders (
    order_id        PRIMARY KEY,
    customer_id     FOREIGN KEY REFERENCES Customers
)

CREATE TABLE Products (
    product_id      PRIMARY KEY
)

CREATE TABLE LineItems (
    order_id        FOREIGN KEY REFERENCES Orders,
    product_id      FOREIGN KEY REFERENCES Products,
    referer_order_id FOREIGN KEY REFERENCES Orders,
    PRIMARY KEY     (order_id, product_id)
)

CREATE TABLE Deliveries (
    order_id,
    product_id,
    date DATE,
    PRIMARY KEY     (order_id, product_id),
    FOREIGN KEY     (order_id, product_id)
        REFERENCES LineItems (order_id, product_id)
)
```

Here is an Entity Relationship diagram for the above schema:

