SDO Relational Data Access Service

Introduction

Warning
This extension is *EXPERIMENTAL*. The behaviour of this extension?including the names of its functions and any other documentation surrounding this extension?may change without notice in a future release of PHP. This extension should be used at your own risk.

In order to use the Relational Data Access Service for Service Data Objects, you will need to understand some of the concepts behind SDO: the data graph, the data object, the disconnected way of working, the change summary, XPath and property expressions, and so on. If you are not familiar with these ideas, you might want to look first at the section on SDO. In addition, the Relational DAS makes use of the PDO extension to isolate itself from specifics of different back-end relational databases. In order to use the Relational DAS you will need to be able to create and pass a PDO database connection; for this reason you might also want to take a look at the section on PDO.

The job of the Relational DAS is to move data between the application and a relational database. In order to do this it needs to be told the mapping between the database entities - tables, columns, primary keys and foreign keys - and the elements of the SDO model - types, properties, containment relationships and so on. You specify this information as metadata when you construct the Relational DAS.

Overview of Operation

1. The first step is to call the Relational DAS's constructor, passing the metadata that defines the mapping between database and SDO model. There are examples of this below.
2. The next step might be to call the **executeQuery()** or **executePreparedQuery()** methods on the Relational DAS, passing either a literal SQL statement for the DAS to prepare and execute, or a prepared statement with placeholders and a list of values to be inserted. You may also need to specify a small amount of metadata about the query itself, so that the Relational DAS knows exactly what columns will be returned from the database and in what order. You will also need to pass a PDO database connection. The return value from **executeQuery()** or **executePreparedQuery()** is a normalised data graph containing all the data from the result set. For a query that returns data obtained from a number of tables, this graph will contain a number of data objects, linked by SDO containment relationships. There may also be SDO non-containment references within the data. Once the query has been executed and the data graph constructed, there is no need for either that instance of the the Relational DAS or the database connection. There are no locks held on the database. Both the Relational DAS and the PDO database connection can be garbage collected.
3. Quite possibly the data in the data graph will go through a number of modifications. The data graph can be serialised into the PHP session and so may have a lifetime beyond just one client-server interaction. Data objects can be created and added to the

graph, the data objects already in the graph can be deleted, and data objects in the graph can be modified.

4. Finally, the changes made to the data graph can be applied back to the database using the **applyChanges()** method of the Relational DAS. For this, another instance of the Relational DAS must be constructed, using the same metadata, and another connection to the database obtained. The connection and the data graph are passed to **applyChanges()**. At this point the Relational DAS examines the change summary and generates the necessary INSERT, UPDATE and DELETE SQL statements to apply the changes. Any UPDATE and DELETE statements are qualified with the original values of the data so that should the data have changed in the database in the meantime this will be detected. Assuming no such collisions have occurred the changes will be committed to the database. The application can then continue to work with the data graph, make more changes and apply them, or can discard it.

There are other ways of working with the data in the database: it is possible to just create data objects and write them to the database without a preliminary call to **executeQuery()**, for example. This scenario and others are explored in the Examples section below.

Installing/Configuring

Requirements

The Relational DAS requires that the SDO extension be installed. The SDO extension requires a version of PHP 5.1, and the Relational DAS requires a recent version that contains an important fix for PDO. The most up-to-date information about required levels of PHP should be found in the changelog for the package on PECL. At the time of writing, though, the Relational DAS requires the most recent beta level of PHP 5.1, that is PHP 5.1.0.

The Relational DAS uses PDO to access the relational database, and so should run with a variety of different relational databases. At the time of writing it has been tested in the following configurations

- MySQL 4.1.14, on Windows. The Relational DAS operates correctly with the php_pdo_mysql driver that comes with the pre-built binaries for PHP 5.1.0.
- MySQL 4.1.13, on Linux. It is necessary to have the most up-to-date PDO driver for MySQL, which comes built in to PHP 5.1.0. It may be necessary to uninstall the usual driver that would have come from PECL using *pear uninstall pdo_mysql*. You will need to configure PHP with the *--with-pdo-mysql* option.
- DB2 8.2 Personal Edition, on Windows. The Relational DAS operates correctly with the php_pdo_odbc driver that comes with the pre-built binaries for PHP 5.1.0.
- DB2 8.2 Personal Developer's Edition, on Linux. The Developer's Edition is needed because it contains the include files needed when PHP is configured and built. You will need to configure PHP with the *--with-pdo-odbc=ibm-db2* option.

The Relational DAS applies changes to the database within a user-delimited transaction: that is, it issues a call to PDO::beginTransaction() before beginning to apply changes, and PDO::commit() or PDO::rollback() on completion. Whichever database is chosen, the database and the PDO driver for the database must support these calls.

Installation

The installation instructions for all the SDO components are in the SDO install section of the SDO documentation.

In any case, the essential facts are that the Relational DAS is written in PHP and it should be placed somewhere on the PHP include_path.

Your application will of course need to include the Relational DAS with a statement like this:

```
<?php
require_once 'SDO/DAS/Relational.php';
```

```
?>
```

Runtime Configuration

This extension has no configuration directives defined in *php.ini*.

Tracing

You may be interested in seeing the SQL statements that are generated in order to apply changes back to the database. At the top of the *SDO/DAS/Relational.php* you will find a number of constants which control whether the process of constructing and executing the SQL statements is to be traced. Try setting *DEBUG_EXECUTE_PLAN* to **TRUE** to see the generated SQL statements.

Resource Types

This extension has no resource types defined.

Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

The Relational DAS introduces no predefined constants.

Examples

Creating, retrieveing, updating and deleting data

This section illustrates how the Relational DAS can be used to create, retrieve, update and delete data in a relational database. Many of the examples are illustrated with a three-table database that contains companies, departments within those companies, and employees that work in those departments. This example is used in a number of places within the SDO literature. See the examples section of the » Service Data Objects specification or the Examples section of the documentation for the SDO extension.

The Relational DAS is constructed with metadata that defines the relational database and how it should be mapped to SDO. The long section that follows describes this metadata and how to construct the Relational DAS. The examples that follow it all assume that this metadata is in an included php file.

The examples below and others can all be found in the *Scenarios* directory in the Relational DAS package.

The Relational DAS throws exceptions in the event that it finds errors in the metadata or errors when executing SQL statements against the database. For brevity the examples below all omit the use of try/catch blocks around the calls to the Relational DAS.

These examples all differ from the expected use of SDO in two important respects.

First, they show all interactions with the database completed within one script. In this respect these scenarios are not realistic but are chosen to illustrate just the use of the Relational DAS. It is expected that interactions with the database will be separated in time and the data graph serialised and deserialised into the PHP session one or more times as the application interacts with an end user.

Second, all queries executed against the database use hard-coded queries with no variables substituted. In this case it is safe to use the simple **executeQuery()** call, and this is what the examples illustrate. In practice, though, it is unlikely that the SQL statement is known entirely ahead of time. In order to allow variables to be safely substituted into the SQL queries, without running the risk of injecting SQL with unknown effects, it is safer to use the **executePreparedQuery()** which takes a prepared SQL statement containing placeholders and a list of values to be substituted.

Specifying the metadata

This first long section describes in detail how the metadata describing the database and the required SDO model is supplied to the Relational DAS.

When the constructor for the Relational DAS is invoked, it needs to be passed several pieces of information. The bulk of the information, passed as an associative array in the first argument to the constructor, tells the Relational DAS what it needs to know about the

relational database. It describes the names of the tables, columns, primary keys and foreign keys. It should be fairly easy to understand what is required, and once written it can be placed in a php file and included when needed. The remainder of the information, passed in the second and third arguments to the constructor, tells the Relational DAS what it needs to know about the relationships between objects and the shape of the data graph; it ultimately determines how the data from the database is to be normalised into a graph.

Database metadata

The first argument to the constructor describes the target relational database.

Each table is described by an associative array with up to four keys.

Key	**Value**
name	The name of the table.
columns	An array listing the names of the columns, in any order.
PK	The name of the column containing the primary key.
FK	An array with two entries, 'from' and 'to', which define a column containing a foreign key, and a table to which the foreign key points. If there are no foreign keys in the table then the 'FK' entry does not need to be specified. Only one foreign key can be specified. Only a foreign key pointing to the primary key of a table can be specified.

```
<?php
/*****************************************************************
* METADATA DEFINING THE DATABASE
*****************************************************************/
$company_table = array (
 'name' => 'company',
 'columns' => array('id', 'name',  'employee_of_the_month'),
 'PK' => 'id',
 'FK' => array (
     'from' => 'employee_of_the_month',
     'to' => 'employee',
     ),
 );
$department_table = array (
 'name' => 'department',
 'columns' => array('id', 'name', 'location', 'number', 'co_id'),
 'PK' => 'id',
 'FK' => array (
     'from' => 'co_id',
```

```
     'to' => 'company',
     )
 );
$employee_table = array (
 'name' => 'employee',
 'columns' => array('id', 'name', 'SN', 'manager', 'dept_id'),
 'PK' => 'id',
 'FK' => array (
     'from' => 'dept_id',
     'to' => 'department',
     )
 );
$database_metadata = array($company_table, $department_table, $employee_table);
?>
```

This metadata corresponds to a relational database that might have been defined to MySQL as:

```
create table company (
id integer auto_increment,
name char(20),
employee_of_the_month integer,
primary key(id)
);
create table department (
id integer auto_increment,
name char(20),
location char(10),
number integer(3),
co_id integer,
primary key(id)
);
create table employee (
id integer auto_increment,
name char(20),
SN char(4),
manager tinyint(1),
dept_id integer,
primary key(id)
);
```

or to DB2 as:

```
create table company ( \
 id integer not null generated by default as identity,  \
 name varchar(20), \
 employee_of_the_month integer, \
 primary key(id) )
create table department ( \
 id integer not null generated by default as identity, \
 name varchar(20), \
 location varchar(10), \
 number integer, \
 co_id integer, \
 primary key(id) )
create table employee ( \
 id integer not null generated by default as identity, \
 name varchar(20), \
 SN char(4), \
 manager smallint, \
```

```
dept_id integer, \
primary key(id) )
```

Note that although in this example there are no foreign keys specified to the database and so the database is not expected to enforce referential integrity, the intention behind the *co_id* column on the department table and the *dept_id* column on the employee table is they should contain the primary key of their containing company or department record, respectively. So these two columns are acting as foreign keys.

There is a third foreign key in this example, that from the *employee_of_the_month* column of the company record to a single row of the employee table. Note the difference in intent between this foreign key and the other two. The *employee_of_the_month* column represents a single-valued relationship: there can be only one employee of the month for a given company. The *co_id* and *dept_id* columns represent multi-valued relationships: a company can contain many departments and a department can contain many employees. This distinction will become evident when the remainder of the metadata picks out the company-department and department-employee relationships as containment relationships.

There are a few simple rules to be followed when constructing the database metadata:

- All tables must have primary keys, and the primary keys must be specified in the metadata. Without primary keys it is not possible to keep track of object identities. As you can see from the SQL statements that create the tables, primary keys can be auto-generated, that is, generated and assigned by the database when a record is inserted. In this case the auto-generated primary key is obtained from the database and inserted into the data object immediately after the row is inserted into the database.
- It is not necessary to specify in the metadata all the columns that exist in the database, only those that will be used. For example, if the company table had another column that the application did not want to access with SDO, this need not be specified in the metadata. On the other hand it would have done no harm to specify it: if specified in the metadata but never retrieved, or assigned to by the application, then the unused column will not affect anything.
- In the database metadata note that the foreign key definitions identify not the destination column in the table which is pointed to, but the table name itself. Strictly, the relational model permits the destination of a foreign key to be a non-primary key. Only foreign keys that point to a primary key are useful for constructing the SDO model, so the metadata specifies the table name. It is understood that the foreign key points to the primary key of the given table.

Given these rules, and given the SQL statements that define the database, the database metadata should be easy to construct.

What the Relational DAS does with the metadata

The Relational DAS uses the database metadata to form most of the SDO model. For each table in the database metadata, an SDO type is defined. Each column which can

represent a primitive value (columns which are not defined as foreign keys) are added as properties to the SDO type.

All primitive properties are given a type of string in the SDO model, regardless of their SQL type. When writing values back to the database the Relational DAS will create SQL statements that treat the values as strings, and the database will convert them to the appropriate type.

Foreign keys are interpreted in one of two ways, depending on the metadata in the third argument to the constructor that defines the SDO containment relationships. A discussion of this is therefore deferred until the section on SDO containment relationships below.

Specifying the application root type

The second argument to the constructor is the application root type. The true root of each data graph is an object of a special root type and all application data objects come somewhere below that. Of the various application types in the SDO model, one has to be the application type immediately below the root of the data graph. If there is only one table in the database metadata, the application root type can be inferred, and this argument can be omitted.

Specifying the SDO containment relationships

The third argument to the constructor defines how the types in the model are to be linked together to form a graph. It identifies the parent-child relationships between the types which collectively form a graph. The relationships need to be supported by foreign keys to be found in the data, in a way shortly to be described.

The metadata is an array containing one or more associative arrays, each of which identifies a parent and a child. The example below shows a parent-child relationship from company to department, and another from department to employee. Each of these will become an SDO property defining a multi-valued containment relationship in the SDO model.

```
<?php
$department_containment = array( 'parent' => 'company', 'child' =>
'department');
$employee_containment = array( 'parent' => 'department', 'child' => 'employee');

$SDO_containment_metadata = array($department_containment,
$employee_containment);
?>
```

Foreign keys in the database metadata are interpreted as properties with either multi-valued containment relationships or single-valued non-containment references, depending on whether they have a corresponding SDO containment relationship specified in the metadata. In the example here, the foreign keys from department to company (the *co_id* column in the department table) and from employee to department (the *dept_id* column in the employee table) are interpreted as supporting the SDO containment relationships. Each containment relationship mentioned in the SDO containment

relationships metadata must have a corresponding foreign key present in the database and defined in the database metadata. The values of the foreign key columns for containment relationships do not appear in the data objects, instead each is represented by a containment relationship from the parent to the child. So the *co_id* column in the department row in the database, for example, does not appear as a property on the department type, but instead as a containment relationship called *department* on the company type. Note that the foreign key and the parent-child relationship appear to have opposite senses: the foreign key points from the department to the company, but the parent-child relationship points from company to department.

The third foreign key in this example, the *employee_of_the_month*, is handled differently. This is not mentioned in the SDO containment relationships metadata. As a consequence this is interpreted in the second way: it becomes a single-valued non-containment reference on the company object, to which can be assigned references to SDO data objects of the employee type. It does appear as a property on the company type. The way to assign a value to it in the SDO data graph is to have a graph that contains an employee object through the containment relationships, and to assign the object to it. This is illustrated in the later examples below.

One-table examples

The following set of examples all use the Relational DAS to work with a data graph containing just one application data object, a single company and the data just to be found the company table. These examples do not exercise the power of SDO or the Relational DAS and of course the same result could be achieved more economically with direct SQL statements but they are intended to illustrate how to work with the Relational DAS.

For this very simple scenario it would be possible to simplify the database metadata to include just the company table - if that were done the second and third arguments to the constructor and the column specifier used in the query example would become optional.

Example #1 - Creating a data object
The simplest example is that of creating a single data object and writing it to the database. In this example a single company object is created, its name is set to 'Acme', and the Relational DAS is called to write the changes to the database. The company name is set here using the property name method. See the Examples section on the SDO extension for other ways of accessing the properties of an object. Data objects can only be created when you have a data object to start with, however. It is for that reason that the first call to the Relational DAS here is to obtain a root object. This is in effect how to ask for an empty data graph - the special root object is the true root of the tree. The company data object is then created with a call to **createDataObject()** on the root object. This creates the company data object and inserts it in the graph by inserting into a multi-valued containment property on the root object called 'company'. When the Relational DAS is called to apply the changes a simple insert statement

'INSERT INTO company (name) VALUES ("Acme");' will be constructed and executed. The auto-generated primary key will be set into the data object and the change summary will be reset, so that it would be possible to continue working with the same data object, modify it, and apply the newer changes a second time.

```
<?php
require_once 'SDO/DAS/Relational.php';
require_once 'company_metadata.inc.php';

/*************************************************************
* Construct the DAS with the metadata
*************************************************************/
$das = new SDO_DAS_Relational
($database_metadata,'company',$SDO_containment_metadata);

/*************************************************************
* Obtain a root object and create a company object underneath.
* Make a simple change to the data object.
*************************************************************/
$root = $das  -> createRootDataObject();
$acme = $root -> createDataObject('company');

$acme->name = "Acme";

/*************************************************************
* Get a database connection and write the object to the database
*************************************************************/
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);
$das -> applyChanges($dbh, $root);
?>
```

Example #2 - Retrieving a data object

In this example a single data object is retrieved from the database - or possibly more than one if there is more than one company called 'Acme'. For each company returned, the *name* and *id* properties are echoed.

In this example the third argument to **executeQuery()**, the column specifier is needed as there are other tables in the metadata with column names of *name* and *id*. If there were no possible ambiguity it could be omitted.

```
<?php
require_once 'SDO/DAS/Relational.php';
require_once 'company_metadata.inc.php';

/*************************************************************
* Construct the DAS with the metadata
*************************************************************/
$das = new SDO_DAS_Relational
($database_metadata,'company',$SDO_containment_metadata);

/*************************************************************
* Get a database connection
```

```
************************************************************/
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);

/************************************************************
* Issue a query to obtain a company object - possibly more if they exist
************************************************************/
$root = $das->executeQuery($dbh,
        'select name, id from company where name="Acme"',
         array('company.name', 'company.id') );

/************************************************************
* Echo name and id
************************************************************/
foreach ($root['company'] as $company) {
 echo "Company obtained from the database has name = " .
 $company['name'] . " and id " . $company['id'] . "\n";
}
?>
```

Example #3 - Updating a data object

This example combines the previous two, in the sense that in order to be updated the object must first be retrieved. The application code reverses the company name (so 'Acme' becomes 'emcA') and then the changes are written back to the database in the same way that they were when the object was created. Because the query searches for the name both ways round the program can be run repeatedly to find the company and reverse its name each time.

In this example the same instance of the Relational DAS is reused for the **applyChanges()**, as is the PDO database handle. This is quite alright; it also alright to allow the previous instances to be garbage collected and to obtain new instances. No state data regarding the graph is held the Relational DAS once it has returned a data graph to the application. All necessary data is either within the graph itself, or can be reconstructed from the metadata.

```
<?php
require_once 'SDO/DAS/Relational.php';
require_once 'company_metadata.inc.php';

/************************************************************
* Construct the DAS with the metadata
************************************************************/
$das = new SDO_DAS_Relational
($database_metadata,'company',$SDO_containment_metadata);

/************************************************************
* Get a database connection
************************************************************/
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);

/************************************************************
* Issue a query to obtain a company object - possibly more if they exist
************************************************************/
```

```
$root = $das->executeQuery($dbh,
 'select name, id from company where name="Acme" or name="emcA"',
 array('company.name', 'company.id') );

/*************************************************************
* Alter the name of just the first company
*************************************************************/
$company = $root['company'][0];
echo "obtained a company with name of " . $company->name . "\n";
$company->name = strrev($company->name);

/*************************************************************
* Write the change back
*************************************************************/
$das->applyChanges($dbh,$root);
?>
```

Example #4 - Deleting a data object

Any companies called 'Acme' or its reverse 'emcA' are retrieved. They are then all deleted from the graph with unset.

In this example they are all deleted in one go by unsetting the containing property (the property defining the containment relationship). It is also possible to delete them individually.

```
<?php
require_once 'SDO/DAS/Relational.php';
require_once 'company_metadata.inc.php';

/*************************************************************
* Construct the DAS with the metadata
*************************************************************/
$das = new SDO_DAS_Relational
($database_metadata,'company',$SDO_containment_metadata);

/*************************************************************
* Get a database connection
*************************************************************/
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);

/*************************************************************
* Issue a query to obtain a company object - possibly more if they exist
*************************************************************/
$root = $das->executeQuery($dbh,
 'select name, id from company where name="Acme" or name="emcA"',
 array('company.name', 'company.id') );

/*************************************************************
* Delete any companies found from the data graph
*************************************************************/
unset($root['company']);

/*************************************************************
```

```
* Write the change(s) back
************************************************************/
$das->applyChanges($dbh,$root);
?>
```

Two-table examples

The following set of examples all use two tables from the company database: the company and department tables. These examples exercise more of the function of the Relational DAS.

In this series of examples a company and department are created, retrieved, updated, and finally deleted. This illustrates the lifecycle for a data graph containing more than one object. Note that this example clears out the company and department tables at the start so that the exact results of the queries can be known.

You can find these examples combined into one script called *1cd-CRUD* in the *Scenarios* directory in the Relational DAS package.

Example #5 - One company, one department - Create

As in the earlier example of creating just one company data object, the first action after constructing the Relational DAS is to call **createRootDataObject()** to obtain the special root object of the otherwise empty data graph. The company object is then created as a child of the root object, and the department object as a child of the company object.

When it comes to applying the changes, the Relational DAS has to perform special processing to maintain the foreign keys that support the containment relationships, especially if auto-generated primary keys are involved. In this example, the relationship between the auto-generated primary key *id* in the company table and the *co_id* column in the department table must be maintained. When inserting a company and department for the first time the Relational DAS has to first insert the company row, then call PDO's **getLastInsertId()** method to obtain the auto-generated primary key, then add that as the value of the *co_id* column when inserting the department row.

```
<?php
require_once 'SDO/DAS/Relational.php';
require_once 'company_metadata.inc.php';

/************************************************************************
**********
* Empty out the two tables
************************************************************************
*********/
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);
$pdo_stmt = $dbh->prepare('DELETE FROM COMPANY;');
$rows_affected = $pdo_stmt->execute();
$pdo_stmt = $dbh->prepare('DELETE FROM DEPARTMENT;');
```

```
$rows_affected = $pdo_stmt->execute();

/************************************************************
* Create a company with name Acme and one department, the Shoe department
************************************************************/
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);
$das = new SDO_DAS_Relational
($database_metadata,'company',$SDO_containment_metadata);

$root = $das -> createRootDataObject();

$acme = $root -> createDataObject('company');
$acme -> name = "Acme";

$shoe = $acme->createDataObject('department');
$shoe->name = 'Shoe';

$das -> applyChanges($dbh, $root);

?>
```

Example #6 - One company, one department - Retrieve and Update

In this case the SQL query passed to **executeQuery()** performs an inner join to join the data from the company and department tables. Primary keys for both the company and department tables must be included in the query. The result set is re-normalised to form a normalised data graph. Note that a column specifier is passed as the third argument to the **executeQuery()** call enabling the Relational DAS to know which column is which in the result set.

Note that the *co_id* column although used in the query is not needed in the result set. In order to understand what the Relational DAS is doing when it builds the data graph it may be helpful to visualise what the result set looks like. Although the data in the database is normalised, so that multiple department rows can point through their foreign key to one company row, the data in the result set is non-normalised: that is, if there is one company and multiple departments, the values for the company are repeated in each row. The Relational DAS has to reverse this process and turn the result set back into a normalised data graph, with just one company object.

In this example the Relational DAS will examine the result set and column specifier, find data for both the company and department tables, find primary keys for both, and interpret each row as containing data for a department and its parent company. If it has not seen data for that company before (it uses the primary key to check) it creates a company object and then a department object underneath it. If it has seen data for that company before and has already created the company object it just creates the department object underneath.

In this way the Relational DAS can retrieve and renormalise data for multiple companies and multiple departments underneath them.

```
<?php
```

```
require_once 'SDO/DAS/Relational.php';
require_once 'company_metadata.inc.php';

/*************************************************************
* Retrieve the company and Shoe department, then delete Shoe and add IT
*************************************************************/
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);
$das = new SDO_DAS_Relational
($database_metadata,'company',$SDO_containment_metadata);

$root = $das->executeQuery($dbh,
'select c.id, c.name, d.id, d.name from company c, department d where
d.co_id = c.id',
array('company.id','company.name','department.id','department.name'));

$acme = $root['company'][0];            // get the first company - will be
'Acme'
$shoe = $acme['department'][0];         // get the first department
underneath - will be 'Shoe'

unset($acme['department'][0]);

$it = $acme->createDataObject('department');
$it->name = 'IT';

$das -> applyChanges($dbh, $root);
?>
```

Example #7 - One company, two departments - Retrieve and Delete

In this example the company and department are retrieved and then deleted. It is not necessary to delete them individually (although that would be possible) - deleting the company object from the data graph also deletes any departments underneath it.

Note the way that the company object is actually deleted using the PHP unset call. The unset has to be performed on the containing property which in this case is the company property on the special root object. You must use:

```
<?php
unset($root['company'][0]);
?>
```

and not:

```
<?php
unset($acme); //WRONG
?>
```

Simply unsetting *$acme* would destroy the variable but leave the data in the data graph untouched.

```
<?php
require_once 'SDO/DAS/Relational.php';
require_once 'company_metadata.inc.php';

/*************************************************************
* Retrieve the company and IT department, then delete the whole company
*************************************************************/
```

```
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);
$das = new SDO_DAS_Relational
($database_metadata,'company',$SDO_containment_metadata);

$root = $das->executeQuery($dbh,
'select c.id, c.name, d.id, d.name from company c, department d where
d.co_id = c.id',
array('company.id','company.name','department.id','department.name'));

$acme = $root['company'][0];
$it = $acme['department'][0];

unset($root['company'][0]);

$das -> applyChanges($dbh, $root);

?>
```

Three-table example

The following examples use all three tables from the company database: the company, department, and employee tables. These introduce the final piece of function not exercised by the examples above: the non-containment reference *employee_of_the_month*.

Like the examples above for company and department, this set of examples is intended to illustrate the full lifecycle of such a data graph.

Example #8 - One company, one department, one employee - Create

In this example a company is created containing one department and just one employee. Note that this example clears out all three tables at the start so that the exact results of the queries can be known.

Note how once the company, department and employee have been created, the *employee_of_the_month* property of the company can be made to point at the new employee. As this is a non-containment reference, this cannot be done until the employee object has been created within the graph. Non-containment references need to be managed carefully. For example if the employee were now deleted from under the department, it would not be correct to try to save the graph without first clearing or re-assigning the *employee_of_the_month* property. The closure rule for SDO data graphs requires that any object pointed at by a non-containment reference must also be reachable by containment relationships.

When it comes to inserting the graph into the database, the procedure is similar to the example of inserting the company and department, but *employee_of_the_month* introduces an extra complexity. The Relational DAS needs to insert the objects working down the tree formed by containment relationships, so company, then department, then employee. This is necessary so that it always has the auto-generated primary key of a parent on hand to include in a child row. But when the company row is

inserted the employee who is employee of the month has not yet been inserted and the primary key is not known. The procedure is that after the employee record is inserted and its primary key known, a final step is performed in which the the company record is updated with the employee's primary key.

```
<?php
require_once 'SDO/DAS/Relational.php';
require_once 'company_metadata.inc.php';

/**************************************************************************
**********
* Empty out the three tables
**************************************************************************
*********/
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);
$pdo_stmt = $dbh->prepare('DELETE FROM COMPANY;');
$rows_affected = $pdo_stmt->execute();
$pdo_stmt = $dbh->prepare('DELETE FROM DEPARTMENT;');
$rows_affected = $pdo_stmt->execute();
$pdo_stmt = $dbh->prepare('DELETE FROM EMPLOYEE;');
$rows_affected = $pdo_stmt->execute();

/**************************************************************************
**********
* Create a tiny but complete company.
* The company name is Acme.
* There is one department, Shoe.
* There is one employee, Sue.
* The employee of the month is Sue.
**************************************************************************
*********/
$das = new SDO_DAS_Relational
($database_metadata,'company',$SDO_containment_metadata);
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);

$root             = $das  -> createRootDataObject();
$acme             = $root -> createDataObject('company');
$acme -> name     = "Acme";
$shoe             = $acme -> createDataObject('department');
$shoe -> name     = 'Shoe';
$shoe -> location = 'A-block';
$sue              = $shoe -> createDataObject('employee');
$sue -> name      = 'Sue';
$acme -> employee_of_the_month = $sue;

$das -> applyChanges($dbh, $root);

echo "Wrote back Acme with one department and one employee\n";
?>
```

Example #9 - One company, one department, one employee - Retrieve and update

The SQL statement passed to the Relational DAS is this time an inner join that

retrieves data from all three tables. Otherwise this example introduces nothing that has not appeared in a previous example.

The graph is updated by the addition of a new department and employee and some alterations to the name properties of the existing objects in the graph. The combined changes are then written back. The Relational DAS will process and apply an arbitrary mixture of additions, modifications and deletions to and from the data graph.

```
<?php
require_once 'SDO/DAS/Relational.php';
require_once 'company_metadata.inc.php';

/**************************************************************************
**********
* Find the company again and change various aspects.
* Change the name of the company, department and employee.
* Add a second department and a new employee.
* Change the employee of the month.
**************************************************************************
*********/
$das = new SDO_DAS_Relational
($database_metadata,'company',$SDO_containment_metadata);
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);

$root = $das->executeQuery($dbh,
 "select c.id, c.name, c.employee_of_the_month, d.id, d.name, e.id, e.name "
.
 "from company c, department d, employee e " .
 "where e.dept_id = d.id and d.co_id = c.id and c.name='Acme'",
  array('company.id','company.name','company.employee_of_the_month',
  'department.id','department.name','employee.id','employee.name'));
$acme         = $root['company'][0];

$shoe         = $acme->department[0];
$sue          = $shoe -> employee[0];

$it           = $acme->createDataObject('department');
$it->name     = 'IT';
$it->location = 'G-block';
$billy        = $it->createDataObject('employee');
$billy->name  = 'Billy';

$acme->name   = 'MegaCorp';
$shoe->name   = 'Footwear';
$sue->name    = 'Susan';

$acme->employee_of_the_month = $billy;
$das -> applyChanges($dbh, $root);
echo "Wrote back company with extra department and employee and all the
names changed (Megacorp/Footwear/Susan)\n";

?>
```

Example #10 - One company, two departments, two employees - Retrieve and delete

The company is retrieved as a complete data graph containing five data objects - the company, two departments and two employees. They are all deleted by deleting the company object. Deleting an object from the graph deletes all the object beneath it in the graph. Five SQL DELETE statements will be generated and executed. As always they will be qualified with a WHERE clause that contains all of the fields that were retrieved, so that any updates to the data in the database in the meantime by another process will be detected.

```
<?php
require_once 'SDO/DAS/Relational.php';
require_once 'company_metadata.inc.php';

/**************************************************************************
**********
* Now read it one more time and delete it.
* You can delete part, apply the changes, then carry on working with the
same graph but
* care is needed to keep closure - you cannot delete the employee who is
eotm without
* reassigning. For safety here we delete the company all in one go.
**************************************************************************
*********/
$das = new SDO_DAS_Relational
($database_metadata,'company',$SDO_containment_metadata);
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);

$root = $das->executeQuery($dbh,
 "select c.id, c.name, c.employee_of_the_month, d.id, d.name, e.id, e.name "
.
 "from company c, department d, employee e " .
 "where e.dept_id = d.id and d.co_id = c.id and c.name='MegaCorp';",
  array('company.id','company.name','company.employee_of_the_month',
  'department.id','department.name','employee.id','employee.name'));
$megacorp = $root['company'][0];

unset($root['company']);
$das -> applyChanges($dbh, $root);

echo "Deleted the company, departments and employees all in one go.\n";

?>
```

Limitations

There are the following limitations in the current release of the Relational DAS:

- No support for nulls. There is no support for SQL NULL type. It is not legal to assign PHP NULL to a data object property and the Relational DAS will not write that back as a NULL to the database. If nulls are found in the database on a query, the property will remain unset.
- Only two types of SDO relationship. The metadata described below allows the Relational DAS to model just two types of SDO relationship: multi-valued containment relationships and single-valued non-containment references. In SDO, whether a property describes a single- or multi-valued relationship, and whether it is containment or non-containment, are independent. The full range of possibilities that SDO allows cannot all be defined. There may be relationships that it would be useful to model but which the current implementation cannot manage. One example is a single-valued containment relationship.
- No support for the full range of SDO data types. The Relational DAS defines all primitive properties in the SDO model as being of type string. SDO defines a richer set of types containing various integer, float, boolean and data and time types. String is adequate for the purposes of the Relational DAS since the combination of PHP, PDO and the database will ensure that values passed as strings will be converted to the proper type before being put in the database. This does affect some scenarios in which the Relational DAS has to work with a data graph that has come from or will go to a different DAS.
- Only one foreign key per table. The metadata only provides the means to specify one foreign key per table. This foreign key may be mapped to one of the two types of SDO relationship supported. Obviously there are some scenarios that cannot be described under this limitation - it is not possible to have two non-containment references from one table to another for example.

SDO-DAS-Relational Functions

Predefined Classes

The Relational DAS provides two classes: the Relational DAS itself and the subclass of Exception that can be thrown. The Relational DAS has four publicly useful calls: the constructor, the **createRootDataObject()** call to obtain the root object of an empty data graph, the **executeQuery()** call to obtain a data graph containing data from a relational database, and the **applyChanges()** call to write changes made to a data graph back to the relational database.

SDO_DAS_Relational

The only object other than an SDO_DAS_Relational_Exception with which the application is expected to interact.

Methods

- __construct - construct the Relational DAS with a model derived from the passed metadata
- createRootDataObject - obtain an otherwise empty data graph containing just the special root object
- executeQuery - execute an SQL query passed as a literal string and return the results as a normalised data graph
- executePreparedQuery - execute an SQL query passed as a prepared statement, with a list of values to substitute for placeholders, and return the results as a normalised data graph
- applyChanges - examine the change summary in the data graph and apply those changes back to the database, subject to an assumption of optimistic concurrency

SDO_DAS_Relational_Exception

Is a subclass of PHP's Exception. It adds no behaviour to Exception. Thrown, with useful descriptive text, to signal errors in the metadata or unexpected failures to perform SQL operations.

SDO_DAS_Relational::applyChanges

SDO_DAS_Relational::applyChanges -- Applies the changes made to a data graph back to the database.

Description

void **SDO_DAS_Relational::applyChanges** (PDO `$database_handle`, SDODataObject `$root_data_object`)

Warning
This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

Given a PDO database handle and the special root object of a data graph, examine the change summary in the datagraph and applies the changes to the database. The changes that it can apply can be creations of data objects, deletes of data objects, and modifications to properties of data objects.

Parameters

PDO_database_handle
: Constructed using the PDO extension. A typical line to construct a PDO database handle might look like this:

```
$dbh = new
PDO("mysql:dbname=COMPANYDB;host=localhost",DATABASE_USER,DATABASE_PASSWORD)
;
```

root_data_object
: The special root object which is at the top of every SDO data graph.

Return Values

None. Note however that the datagraph that was passed is still intact and usable. Furthermore, if data objects were created and written back to a table with autogenerated primary keys, then those primary keys will now be set in the data objects. If the changes were successfully written, then the change summary associated with the datagraph will have been cleared, so that it is possible to now make further changes to the data graph and apply those changes in turn. In this way it is possible to work with the same data graph and apply changes repeatedly.

Errors/Exceptions

SDO_DAS_Relational::applyChanges() can throw an SDO_DAS_Relational_Exception if it is unable to apply all the changes correctly.

The Relational DAS starts a database transaction before beginning to apply the changes and will commit the transaction only if they are all successful. The Relational DAS generates qualified update and delete statements which contain a where clause that specifies that the row to be updated or deleted must contain the same values that it did when the data was first retrieved. This is how the the optimistic concurrency is implemented. If any of the qualified update or delete statements fails to update or delete their target row, it may be because the data has been altered in the database in the meantime. In any event, if any update fails for any reason, the transaction is rolled back and an exception thrown. The exception will contain the generated SQL statement that failed.

The Relational DAS also catches any PDO exceptions and obtains PDO diagnostic information which it includes in an SDO_DAS_Relational_Exception which it then throws.

Examples

Please see the Examples section in the general information about the Relational DAS for many examples of calling this method. Please see also the section on Tracing to see how you can see what SQL statements are generated by the Relational DAS.

SDO_DAS_Relational::__construct

SDO_DAS_Relational::__construct -- Creates an instance of a Relational Data Access Service

Description

SDO_DAS_Relational **SDO_DAS_Relational::__construct** (array `$database_metadata` [, string `$application_root_type` [, array `$SDO_containment_references_metadata`]])

Warning
This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

Constructs an instance of a Relational Data Access Service from the passed metadata.

Parameters

database_metadata
: An array containing one or more table definitions, each of which is an associative array containing the keys *name*, *columns*, *PK*, and optionally, *FK*. For a full discussion of the metadata, see the metadata section in the general information about the Relational DAS.

application_root_type
: The root of each data graph is an object of a special root type and the application data objects come below that. Of the various application types in the SDO model, one has to be the the application type immediately below the root of the data graph. If there is only one table in the database metadata, so the application root type can be inferred, this argument can be omitted.

SDO_containment_references_metadata
: An array containing one or more definitions of a containment relation, each of which is an associative array containing the keys *parent* and *child*. The containment relations describe how the types in the model are connected to form a tree. The type specified as the application root type must be present as one of the parent types in the containment references. If the application only needs to work with one table at a time, and there are no containment relations in the model, this argument can be omitted. For a full discussion of the metadata, see the metadata section in the general information about the Relational DAS.

Return Values

Returns an SDO_DAS_Relational object on success.

Errors/Exceptions

SDO_DAS_Relational::__construct() throws a SDO_DAS_Relational_Exception if any problems are found in the metadata.

Examples

For a full discussion of the metadata, see the metadata section in the general information about the Relational DAS.

SDO_DAS_Relational::createRootDataObject

SDO_DAS_Relational::createRootDataObject -- Returns the special root object in an otherwise empty data graph. Used when creating a data graph from scratch.

Description

SDODataObject **SDO_DAS_Relational::createRootDataObject** (void)

Warning
This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

Returns the special root object at the top of an otherwise empty data graph. This call is used when the application wants to create a data graph from scratch, without having called **executeQuery()** to create a data graph.

The special root object has one multi-valued containment property, with a name of the application root type that was passed when the Relational DAS was constructed. The property can take values of only that type. The only thing that the application can usefully do with the root type is to call **createDataObject()** on it, passing the name of the application root type, in order to create a data object of their own application type.

Parameters

None.

Return Values

The root object.

Errors/Exceptions

None.

Examples

Please see the Examples section in the general information about the Relational DAS for many examples of calling this method.

SDO_DAS_Relational::executePreparedQuery

SDO_DAS_Relational::executePreparedQuery -- Executes an SQL query passed as a prepared statement, with a list of values to substitute for placeholders, and return the results as a normalised data graph.

Description

SDODataObject **SDO_DAS_Relational::executePreparedQuery** (PDO `$database_handle`, PDOStatement `$prepared_statement`, array `$value_list` [, array `$column_specifier`])

Warning
This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

Executes a given query against the relational database, using the supplied PDO database handle. Differs from the simpler **executeQuery()** in that it takes a prepared statement and a list of values. This is the appropriate call to use either when the statement is to executed a number of times with different arguments, and there is therefore a performance benefit to be had from preparing the statement only once, or when the the SQL statement is to contain varying values taken from a source that cannot be completely trusted. In this latter case it may be unsafe to construct the SQL statement by simply concatenating the parts of the statement together, since the values may contain pieces of SQL. To guard against this, a so-called SQL injection attack, it is safer to prepare the SQL statement with placeholders (also known as parameter markers, denoted by '?') and supply a list of the values to be substituted as a separate argument. Otherwise this function is the same as **executeQuery()** in that it uses the model that it built from the the metadata to interpret the result set and returns a data graph.

Parameters

PDO_database_handle
: Constructed using the PDO extension. A typical line to construct a PDO database handle might look like this:

```
$dbh = new
PDO("mysql:dbname=COMPANYDB;host=localhost",DATABASE_USER,DATABASE_PASSWORD)
;
```

prepared_statement
: A prepared SQL statement to be executed against the database. This will have been prepared by PDO's **prepare()** method.

value_list

An array of the values to be substituted into the SQL statement in place of the placeholders. In the event that there are no placeholders or parameter markers in the SQL statement then this argument can be specified as **NULL** or as an empty array;

column_specifier

The Relational DAS needs to examine the result set and for every column, know which table and which column of that table it came from. In some circumstances it can find this information for itself, but sometimes it cannot. In these cases a column specifier is needed, which is an array that identifies the columns. Each entry in the array is simply a string in the form *table-name.column_name*. The column specifier is needed when there are duplicate column names in the database metadata, For example, in the database used within the examples, all the tables have both a *id* and a *name* column. When the Relational DAS fetches the result set from PDO it can do so with the PDO_FETCH_ASSOC attribute, which will cause the columns in the results set to be labelled with the column name, but will not distinguish duplicates. So this will only work when there are no duplicates possible in the results set. To summarise, specify a column specifier array whenever there is any uncertainty about which column could be from which table and only omit it when every column name in the database metadata is unique. All of the examples in the Examples use a column specifier. There is one example in the *Scenarios* directory of the installation that does not: that which works with just the employee table, and because it works with just one table, there can not exist duplicate column names.

Return Values

Returns a data graph. Specifically, it returns a root object of a special type. Under this root object will be the data from the result set. The root object will have a multi-valued containment property with the same name as the application root type specified on the constructor, and that property will contain one or more data objects of the application root type.

In the event that the query returns no data, the special root object will still be returned but the containment property for the application root type will be empty.

Errors/Exceptions

SDO_DAS_Relational::executePreparedQuery() can throw an SDO_DAS_Relational_Exception if it is unable to construct the data graph correctly. This can occur for a number of reasons: for example if it finds that it does not have primary keys in the result set for all the objects. It also catches any PDO exceptions and obtains PDO diagnostic information which it includes in an SDO_DAS_Relational_Exception which it then throws.

Examples

Example #11 - Retrieving a data object using executePreparedQuery()
In this example a single data object is retrieved from the database - or possibly more

than one if there is more than one company called 'Acme'. For each company returned, the *name* and *id* properties are echoed.

Other examples of the use of **executePreparedQuery()** can be found in the example code supplied in *sdo/DAS/Relational/Scenarios*.

```
<?php
require_once 'SDO/DAS/Relational.php';
require_once 'company_metadata.inc.php';

/*************************************************************
* Construct the DAS with the metadata
*************************************************************/
$das = new SDO_DAS_Relational
($database_metadata,'company',$SDO_reference_metadata);

/*************************************************************
* Get a database connection
*************************************************************/
$dbh = new PDO(PDO_DSN,DATABASE_USER,DATABASE_PASSWORD);

/*************************************************************
* Issue a query to obtain a company object - possibly more if they exist
* Use a prepared query with a placeholder.
*************************************************************/
$name = 'Acme';
$pdo_stmt = $dbh->prepare('select name, id from company where name=?');
$root = $das->executePreparedQuery(
   $dbh,
   $pdo_stmt,
   array($name),
   array('company.name', 'company.id'));

/*************************************************************
* Echo name and id
*************************************************************/
foreach ($root['company'] as $company) {
   echo "Company obtained from the database has name = " .
   $company['name'] . " and id " . $company['id'] . "\n";
}
?>
```

SDO_DAS_Relational::executeQuery

SDO_DAS_Relational::executeQuery -- Executes a given SQL query against a relational database and returns the results as a normalised data graph.

Description

SDODataObject **SDO_DAS_Relational::executeQuery** (PDO `$database_handle`, string `$SQL_statement` [, array `$column_specifier`])

Warning
This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

Executes a given query against the relational database, using the supplied PDO database handle. Uses the model that it built from the the metadata to interpret the result set. Returns a data graph.

Parameters

PDO_database_handle
: Constructed using the PDO extension. A typical line to construct a PDO database handle might look like this:

```
$dbh = new
PDO("mysql:dbname=COMPANYDB;host=localhost",DATABASE_USER,DATABASE_PASSWORD)
;
```

SQL_statement
: The SQL statement to be executed against the database.

column_specifier
: The Relational DAS needs to examine the result set and for every column, know which table and which column of that table it came from. In some circumstances it can find this information for itself, but sometimes it cannot. In these cases a column specifier is needed, which is an array that identifies the columns. Each entry in the array is simply a string in the form *table-name.column_name*. The column specifier is needed when there are duplicate column names in the database metadata. For example, in the database used within the examples, all the tables have both a *id* and a *name* column. When the Relational DAS fetches the result set from PDO it can do so with the PDO_FETCH_ASSOC attribute, which will cause the columns in the results set to be labelled with the column name, but will not distinguish duplicates. So this will only work when there are no duplicates possible in the results set. To summarise, specify a column specifier array whenever there is any uncertainty about which column could be from which table and only omit it when every column name in the database metadata

is unique. All of the examples in the Examples use a column specifier. There is one example in the *Scenarios* directory of the installation that does not: that which works with just the employee table, and because it works with just one table, there can not exist duplicate column names.

Return Values

Returns a data graph. Specifically, it returns a root object of a special type. Under this root object will be the data from the result set. The root object will have a multi-valued containment property with the same name as the application root type specified on the constructor, and that property will contain one or more data objects of the application root type.

In the event that the query returns no data, the special root object will still be returned but the containment property for the application root type will be empty.

Errors/Exceptions

SDO_DAS_Relational::executeQuery() can throw an SDO_DAS_Relational_Exception if it is unable to construct the data graph correctly. This can occur for a number of reasons: for example if it finds that it does not have primary keys in the result set for all the objects. It also catches any PDO exceptions and obtains PDO diagnostic information which it includes in an SDO_DAS_Relational_Exception which it then throws.

Examples

Please see the Examples section in the general information about the Relational DAS for many examples of calling this method.