SQL PDO and Microsoft SQL Server

By: Blue Parabola, LLC

Contents

Accessing Databases using PHP

Since the early days of the language, one of PHP's primary tasks has been taking data from the web and storing it in a database or pulling it from the database to display it on a web page. One of the primary drivers of PHP adoption on the web is that database access is baked into the language. Most databases can be accessed through native function calls in the core of the language. A handful of less common databases are available through PECL[1] extensions. Either way, PHP developers can easily access most databases in use now or in the recent past.

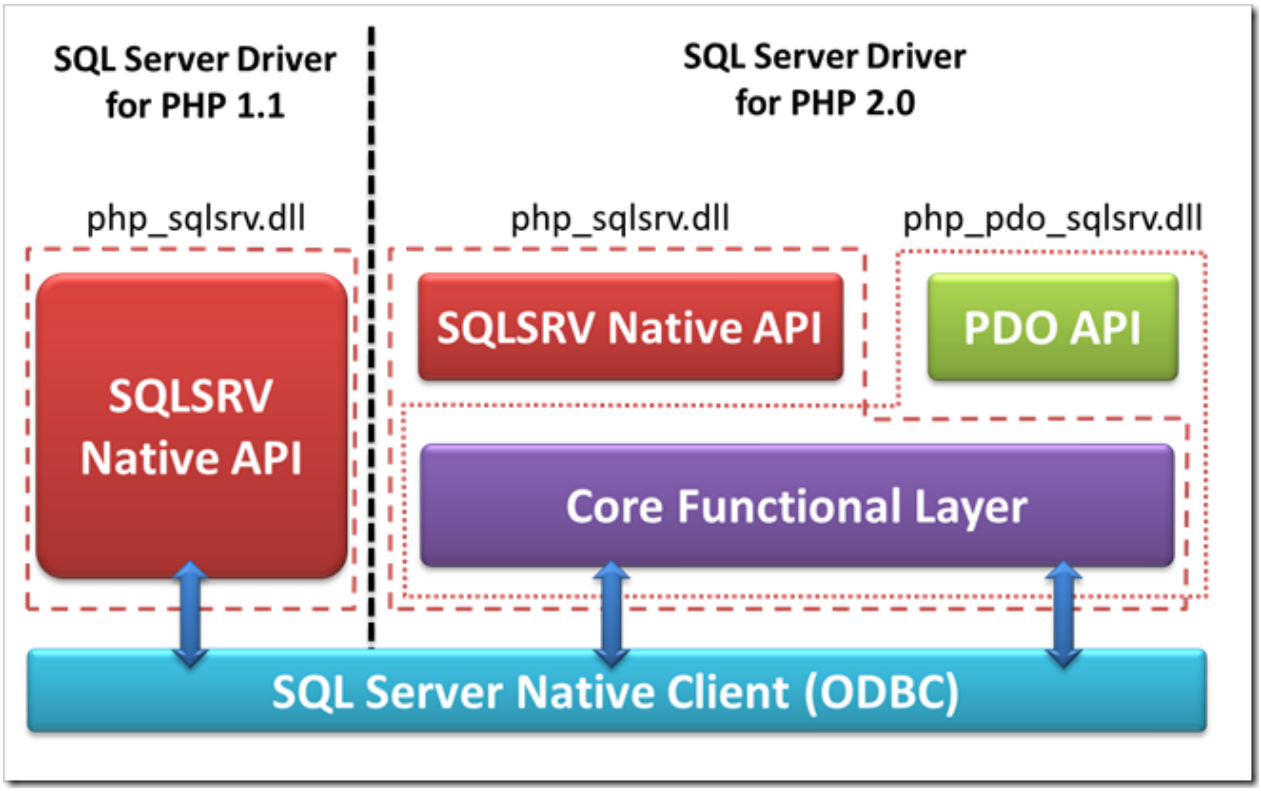
It didn't take the core PHP developers long to realize that a different implementation for accessing each and every database was not the best plan of action. Each database has a slightly different way of doing things and therefore, a different set of method calls. The PHP community approached this from two different directions. First, there are several projects that attempt to isolate and abstract the variations in SQL dialects to protect developers from having to learn the intricacies of each version. Second, the core developers began to abstract the connectivity to the different database layers so that all databases could be accessed using a standard and consistent set of methods. The resulting access layer was named PHP Data Objects or PDO for short. Not all database drivers support PDO but for those that do, PDO has become the standard way of accessing them.

The most notable missing driver within PDO was always Microsoft's SQL Server. SQL Server Driver for PHP have been available for some time, but they lagged behind SQL Server development and acquired a poor reputation among PHP developers. Recently, however, that has begun to change. The Microsoft SQL Server Team released new native PHP drivers. These drivers make SQL Server a first class citizen among PHP database drivers. Additionally, the SQL Server team released PDO drivers for SQL Server. This allows any PHP developer working on Windows, who understand and uses PDO drivers for other databases, to quickly interact with and use SQL Server databases.

SQL Server Driver for PHP 2.0 is fundamentally different than the previous 1.1 version. The previous version provided a native API for interacting with SQL Server directly. While this was useful and effective, it required the PHP developer to know and understand some of the intricacies of the SQL Server engine itself and as a result, made adoption more difficult. Although

1 http://pecl.php.net

Version 2.0 includes the native API of the 1.1 version, the addition of the PDO driver is a completely new approach. Developers who are already familiar with the PDO methods and operations can immediately be productive. In many cases, their applications will work immediately or with only minor adjustments.



Installing SQL Server Driver for PHP 2.0

Installing SQL Server Driver for PHP 2.0 is quite simple and can be accomplished in under five minutes. Visit the SQL Server Driver for PHP 2.0[2] website, and select "Download." After the download completes, run the executable to extract the files. You will also need to install SQL Server Native Access Client 2008 R2 if you do not already have it installed.

The next question is: Which version of the driver do you need? This depends on your stack:

- If you are running PHP 5.2, you have little choice but to run PHP with FastCGI. If you are using IIS, you use the Non-Thread Safe (NTS) version. If you are using Apache, use the Thread Safe (TS) version.
- If you are running PHP 5.3 under IIS with FastCGI, you need the Non-Thread Safe (NTS) version compiled with Visual Studio 2008 (VC9).
- If you are running PHP 5.3 under Apache as mod_php, you need the Thread Safe (TS) version compiled with Visual Studio 6. (VC6)
- If you are running PHP 5.3 under Apache as FastCGI, you need the Non-Thread Safe (NTS) version compiled with Visual Studio 6. (VC6)

Once you have determined which version of the drivers you need, the next step is to place them in the proper directory. By default, all of your extensions are placed in the ext directory within the php directory. Place the appropriate DLLs in the ext directory. Then, with a text editor, open your php.ini file.

Find the section labeled:

```
;;;;;;;;;;;;;;;;;;;;;
; Dynamic Extensions ;
;;;;;;;;;;;;;;;;;;;;;
```

2 http://www.microsoft.com/downloads/details.aspx?FamilyID=df4d9cc9-459c-4d75-a503-ae3fceb85860&displaylang=en

You will see a series of extensions already defined in this area. At the bottom of the list, add yours. For example:

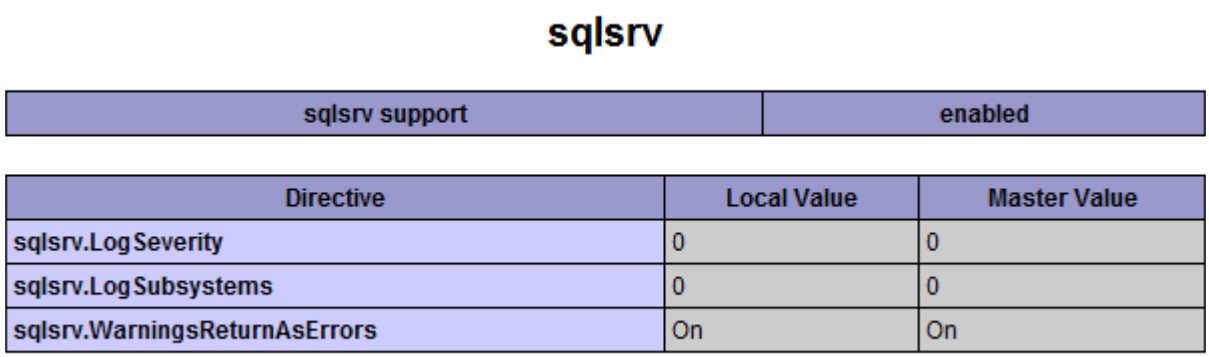
```
extension=php _ sqlsrv _ 53 _ ts _ vc6.dll
extension=php _ pdo _ sqlsrv _ 53 _ ts _ vc6.dll
```

Make sure you install both the `php_sqlsrv_*.dll` and the `php_pdo_sqlsrv_*.dll` extensions.

Once you have done this, you need to restart your web server. If everything worked, your web server will restart. Create an info.php file in the web root of your web site that contains the following:

```
<?php
php _ info();
?>
```

Viewing this page in a web browser will show you your PHP configuration. Extensions are listed alphabetically, and if the SQL Server drivers have been properly installed, you will see a section that looks like this for the main driver

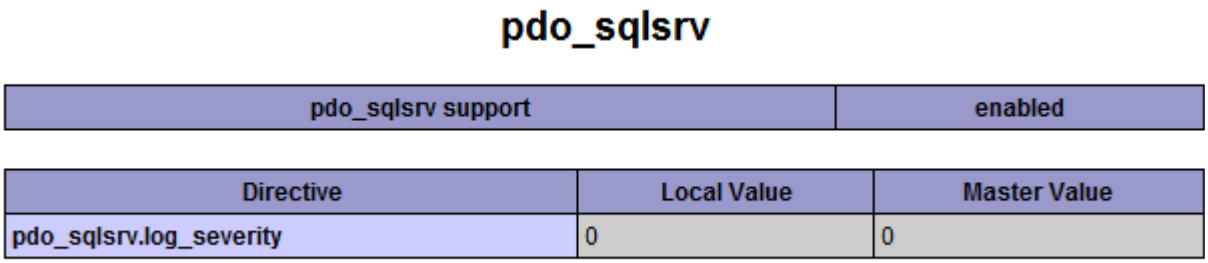


sqlsrv

sqlsrv support	enabled

Directive	Local Value	Master Value
sqlsrv.LogSeverity	0	0
sqlsrv.LogSubsystems	0	0
sqlsrv.WarningsReturnAsErrors	On	On

If you properly installed the PDO driver, you should see this:



pdo_sqlsrv

pdo_sqlsrv support	enabled

Directive	Local Value	Master Value
pdo_sqlsrv.log_severity	0	0

Installing the drivers sounds like a complex procedure, but in reality, it should take you about five minutes to complete.

Accessing SQL Server from PHP

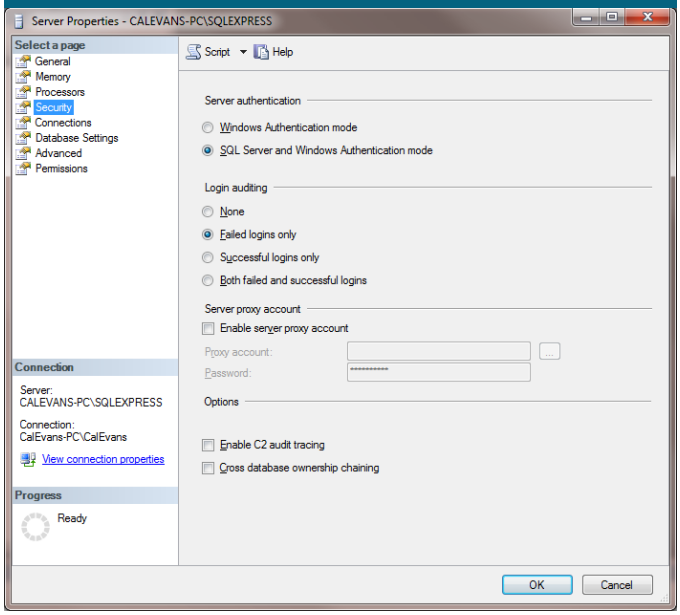
Once you have the drivers installed, you need a database to access. Installing and configuring SQL Server is outside the scope of this document. We will assume that if you have made it this far, you have already completed that step. For the first few samples, we will use the AdventureWorks database. If you don't have a copy of AdventureWorks, you can find it here: http://msftdbprodsamples.codeplex.com/.

Once you have the database installed on your SQL Server, the final thing you need to do is add a user account for your web application. Again, the ramifications of doing this in production are beyond the scope of this document. If you do not understand SQL Server authentication and how it relates to web applications, it is strongly suggested that you consult with someone who does before moving an application into production. For testing and your own prototyping, Technet[3] can be a useful resource for information and guidance.

For the purposes of the sample code, a user, "testuser", was created with the password "testuser" and access was limited to the AdventureWorks database. Make sure that your SQL Server security is set for "SQL Server and Windows Authentication mode", otherwise your user won't be allowed to connect (Figure 1).

Once you have your connection set up, create an index.php file in your web root with this sample code:

FIGURE 1



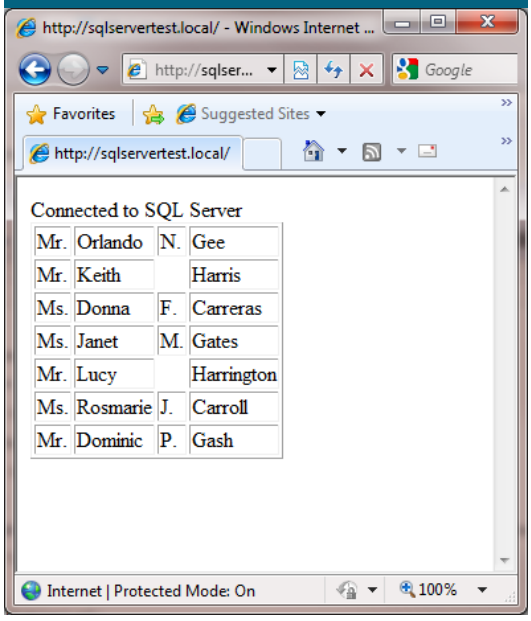
3 http://technet.microsoft.com/en-us/sqlserver/bb895929.aspx

```
<?php
/*
 * Specify the server and connection string attributes.
 */
$serverName = 'CALEVANS-PC\SQLEXPRESS';
$uid        = 'testuser';
$pwd        = 'testuser';
$database   = 'AdventureWorks';
try {
    $conn = new PDO( "sqlsrv:server=$serverName;database=$database", $uid, $pwd);
    $conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
    $conn->setAttribute(PDO::SQLSRV_ATTR_DIRECT_QUERY => true);
    echo "Connected to SQL Server<br /><br />\n";
    echo "Customers with a CustomerID less than 10<br />\n";
    $sql  = 'select * from SalesLT.Customer where CustomerID<10';
    $results = $conn->query( $sql );
    outputRows($results);

    echo "<br /><br />Customers with the title Mr. and a CustomerId less than 10<br />\n";
    $sql = 'select Title, FirstName, MiddleName, LastName from SalesLT.Customer where Title = :title and
CustomerId<10';
    $query = $conn->prepare($sql);
    $title = 'Mr.';
    $query->execute(array(':title'=>$title));
    outputRows($query);

    // Free statement and connection resources.
    $stmt = null;
    $conn = null;
} catch( PDOException $e ) {
    echo "<h1>Error connecting to SQL Server</h1><pre>";
    echo $e->getMEssage();
    exit();
}

function outputRows($results)
{
    echo "<table border='1'>\n";
    while ( $row = $results->fetch( PDO::FETCH_ASSOC ) ){
    echo "<tr><td>{$row['Title']}</td><td>{$row['FirstName']}</
td><td>{$row['MiddleName']}</td><td>{$row['LastName']}</td>\n";
    }
    echo "</table>\n";
    return;
}
?>
```

If everything is working correctly, you should see a simple web page that looks like Figure 1.

Congratulations, things are working. Now that your system is working and connected, we will look at the sample code that got it working and take a more in-depth look at PDO in general.

FIGURE 2



PDO: The Why and the How

The Why

While drivers for PHP to access SQL Server have been available for some time now, this new version is a milestone as it gives SQL Server users access not only to native PHP drivers, but to PDO drivers as well. The PDO section of the PHP manual[4] describes PDO as such:

> PDO provides a *data-access* abstraction layer, which means that, regardless of which database you're using, you use the same functions to issue queries and fetch data. PDO does *not* provide a *database* abstraction; it doesn't rewrite SQL or emulate missing features. You should use a full-blown abstraction layer if you need that facility.

This means that while you can use the same commands to execute queries on both SQL Server and MySQL, the actual SQL is not necessarily the same. For reference, a database abstraction layer insulates you from the differences between dialects of SQL. PDO, being a data-access abstraction layer, insulates you from the differences in the command structures inherent in the native drivers.

SQL Server's commands mirror the command structure normally used to access SQL Server in other languages but may not be familiar to PHP programmers.

The How

As a PHP programmer, you are most likely familiar with PDO as it has been the preferred way to talk to databases for some time now. For those who are new to PDO, here is a quick primer.

4 http://php.net/manual/en/book.pdo.php

Let's use our sample code above as a reference.

```
$conn = new PDO( "sqlsrv:server=$serverName;database=$database", $uid, $pwd);
```

This line is how the actual connection to the database is made. To remain consistent across vendors, PDO accepts four parameters.

1. DSN. This is the connection string that tells PHP the name of the server and the database. The most common SQL Server DSN properties are listed below, while the complete list can be found at http://msdn.microsoft.com/en-US/library/ff628167%28v=SQL.90%29.aspx
    1. Name of the driver, in this case: sqlsrv
    2. The server name, identified by the property "server"
    3. The database name, identified by the property "database"
    4. An application name used for tracing, identified by the property "app"
2. (Optional) UserName. If you want to specify a specific user. If you omit the UserName option then Windows authentication is assumed and the system will attempt to authenticate the user that the web server is running under.
3. (Optional) Password. If you want to specify the user's password.
4. (Optional) DriverOptions array. If there are additional properties or configuration options desired, you can set them here.

As you can see above, the sample code sets all of the necessary properties in variables and then opens the connection. One aspect to note is that instantiating a new instance of a PDO connection is wrapped in a Try/Catch construct. If the connection fails, it will throw a PDOException.

Once you have made your connection, querying the database is as easy as building your SQL and executing it:

```
$sql  = 'select * from SalesLT.Customer where CustomerID<10';
$conn->setAttribute(PDO::SQLSRV_ATTR_DIRECT_QUERY => true);
$results = $conn->query( $sql );
```

This simple query serves to show that the script is connected to the database but misses one of the big advantages of PDO itself: parameterized queries. Parameterized queries allow you to safely insert variables into a script and minimizes the danger of SQL injection. From a performance perspective, parameterized queries allow the SQL Server engine to build a query plan and cache it for later executions. Normally, this only happens when the execute() method is called directly, but by setting PDO::SQLSRV_ATTR_DIRECT_QUERY to true, we can benefit in all cases.

The next query in our example shows a prepared statement. SQL Server supports prepared statements natively, so PHP allows the database to do the heavy lifting. Prepared statements are beneficial for several reasons:

- Complex queries that need to be executed multiple times with different parameters can be parsed once and then executed multiple times with different values. The act of preparing the statement is the time consuming part. For programs that execute the same complex SQL query multiple times, the speed difference is noticeable.
- Bound parameters do not need to have their quotes escaped, PDO and SQL Server will take care of this. This helps with the security of applications as it helps protect against SQL injection attacks.

In this case, we are using named parameters in our query. The parameter `:title` is a placeholder in the query that the parser will pick up on. We are passing its value as an array into the execute() statement.

```
$sql = 'select Title,
            FirstName,
            MiddleName,
            LastName
       from SalesLT.Customer
            where Title = :title and CustomerId<10';
$query = $conn->prepare($sql);
$query->execute(array(':title'=>'Mr.'));
```

The final line is the actual execution of the query and is where we specify the values for our parameters as an array. Notice that the execute() statement does not actually return a value. Once you have executed the query, you can use `fetch()` (as we do in the function `outputRows()`) or `fetchAll()` to get the resulting records.

In the above example, we passed the value of our parameter directly into the `execute()` statement. Alternatively, we could execute a `bindParam()` statement to set the value of the parameter before executing:

```
$sql = 'select Title,
            FirstName,
            MiddleName,
            LastName
       from SalesLT.Customer
            where Title = :title and CustomerId<10';
$query = $conn->prepare($sql);
$title = 'Mr.';
$query->bindParam(':title', $title);
$query->execute();
```

Finally, instead of named parameters, we could use positional parameters. In this case, the parameter would be represented by a `?` instead of `:name`. The bindParam() method takes two parameters: the first is the position of the number of the ?, and the second is the value. It is important to note that the position count starts at one as opposed to zero.

```
$sql = 'select Title,
            FirstName,
            MiddleName,
            LastName
       from SalesLT.Customer
            where Title = ? and CustomerId<10';
$query = $conn->prepare($sql);
$query->bindParam(1,'Mr.');
$query->execute();
```

As stated earlier, because PDO is a data-access abstraction layer, the commands, but not the SQL, are the same for MySQL and Oracle as well.

The Inners and Outers of JOINS

One important aspect of storing data in a SQL database is the ability to join data between tables - and even databases - for retrieval. Simple joins can be accomplished by using the WHERE clause of your statement while more complex joins (inner, outer, left, right, etc.) will require the From/Join clauses.

```
$sql = “select o.OrderDate,
                od.OrderQty,
                od.UnitPrice
        from AdventureWorks.SalesLT.SalesOrderHeader o,
                AdventureWorks.SalesLT.SalesOrderDetail od
        where o.SalesOrderID = od.SalesOrderID and
                 CustomerID = :custId
        order by OrderDate;”;
$query = $conn->prepare($sql);
$query->bindParam(‘:custId’,29847);
$query->execute();
```

This query joins the OrderHeader table with the OrderDetail table to give us a quick overview of the items that a customer has ordered. This simple type of query is one with which most developers will be familiar. However, as needs grow and the number of tables and fields grow, so does the complexity of the queries that need to be executed. The query above could be re-written as:

```
$sql = “select o.OrderDate,
                od.OrderQty,
                                od.UnitPrice
              from AdventureWorks.SalesLT.SalesOrderHeader o
                Left Join AdventureWorks.SalesLT.SalesOrderDetail od on
                         o.SalesOrderID = od.SalesOrderID
           where CustomerID = :custId
              order by OrderDate;”;
$query = $conn->prepare($sql);
$query->bindParam(‘:custId’,29847);
$query->execute();
```

Both queries return the exact same result set. There is no hard and fast rule as to why you should use JOINs over WHERE clauses other than the fact that as of SQL ANSI-92, JOINS are the preferred way of joining tables. In simple queries where you don’t need the finesse of a JOIN over a WHERE clause, the result sets are the same and the optimizer sees them as the same query. However, the JOIN syntax does give developers advantages over the WHERE clause in complex queries. To keep from having to maintain code using multiple standards, it is recommended that developers use the JOIN syntax.

The JOIN syntax gives us control over the query that would not otherwise be possible. For example:

```
$sql = “select c.CompanyName,
                a.AddressLine1,
                a.AddressLine2,
                a.City,
                a.StateProvince,
                a.PostalCode,
                a.CountryRegion
        from AdventureWorks.SalesLT.Customer c
                Left Join AdventureWorks.SalesLT.CustomerAddress ca on c.CustomerID = ca.CustomerID
                Left join AdventureWorks.SalesLT.Address a on ca.AddressID=a.AddressID”;
$query = $conn->prepare($sql);
$query->execute();
```

Executing this query gives us a result set of 857 records. It includes every customer in the database. For each customer, we get one record for each address referenced in the CustomerAddress table. We also get each customer that does not have an address listed. The Left clause in the Join gives us the ability to get all customers - whether they have an address or not - and the missing information is filled by NULL's.

```
$sql = "select c.CompanyName,
                a.AddressLine1,
                a.AddressLine2,
                a.City,
                a.StateProvince,
                a.PostalCode,
                a.CountryRegion
        from AdventureWorks.SalesLT.Customer c
                Right Join AdventureWorks.SalesLT.CustomerAddress ca on c.CustomerID = ca.CustomerID
                Left join AdventureWorks.SalesLT.Address a on ca.AddressID=a.AddressID";
$query = $conn->prepare($sql);
$query->execute();
```

Changing the Left to a Right gives us an entirely different result set. Now the query will only contain customers who have an address referenced in the CustomerAddress table. This flexibility, along with the ease of maintenance, are the biggest reasons developers are encouraged to adopt the JOIN syntax over joining tables in the WHERE clause.

Most developers familiar with MySQL's dialect of SQL should be able to start creating Transact SQL (the name of SQL Server's SQL) without a problem. Using the tools included with SQL Server, like "Microsoft SQL Management Studio", developers can create and test SQL statements. Then, once they are ready, integrate them into their applications.

SQL Azure

On the other side, there is SQL Azure. As part of the cloud, SQL Azure is the big database in the sky that looks, acts, and can effectively be treated as part of a standard SQL Server infrastructure. It provides all of the features and functionality of a standard SQL Server instance, but adds the scalability and high availability nature of cloud-based solutions. More importantly, it removes the need for most of the standard administrative tasks like installation, infrastructure configuration, provisioning, and applying patches. This allows the team to focus on their application and customers' needs. From a development perspective, the most important part to remember is that SQL Azure looks and behaves similarly enough to SQL Server that many will not recognize the difference.

To get started, ensure that your Windows Live account is activated, and go to Windows Azure Services[5]. Fill in an admin username and a password, then and select a region to get your first database started. In just a moment, you'll see it appear under your list of Projects. Select your database and you'll get basic information on configuration as shown in the Figure below. By default, any authenticated user can connect and query the database, but you can choose to add allowed IP ranges by changing the Firewall Settings tab. From there, you can connect to the SQL Azure instance through your SQL Server Management Studio just as you normally would by specifying the proper username, password, and server as shown in Figure 3.

Once you have your database instance, the next step is to connect. Just like normal, start your SQL Server Management Studio, and connect using the credentials you've just set (Figure 4).

This will add your SQL Azure instance to the left-hand database browser just as always. From here, you can use the standard "New Query..." dialog to run ad hoc queries or expand into the general Query Browser to examine your queries. Finally, you can create, edit, and even remove logins according to your needs. By all appearances, this looks, interacts, and behaves the same as SQL Server and should be familiar to the point of being indistinguishable.

To accomplish something useful, most developers will want to move an existing database to SQL Azure and determine

5 https://sql.azure.com/

the application's compatibility and performance. To do this, the first step is to export the existing database through the SQL Server Management Studio. On export, be sure to set "Convert UDDTs to base types" and "Script Data" to True and "Script extended properties" and "Script USE DATABASE" to False. Unfortunately, in the latest version of SQL Server Management Studio, a number of manual changes are required to the create script in order to make it SQL Azure compatible. Most are simple find and replace operations, but a number of the changes require modifying indexes, changing the database configurations, or removing unsupported features, instead of modifying data types or logical structures.[6]

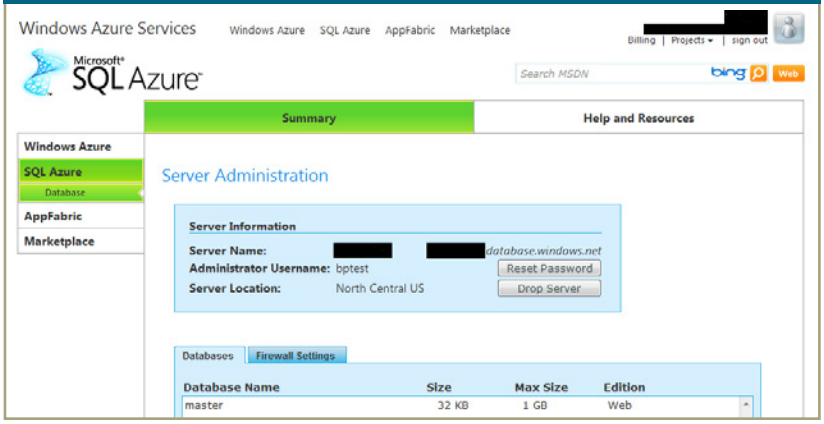
Two useful tools for working with SQL Azure are:

- SQL Server Tools and Utilities Support
- sqlcmd, SQL Server Management Studio, and other useful tools supplied by Microsoft.
- SQL Azure Migration Wizard v3.3.5
- The SQL Azure Migration Wizard helps you migrate your local SQL Server 2005 / 2008 databases into SQL Azure. The wizard walks you through the selection of your SQL objects, creates SQL scripts suitable for SQL Azure, and allows you to migrate your data.

To update a PHP-based SQL Server application to run with SQL Azure instead, the updates are trivial. First, ensure that the SQL Azure instance is configured as above. Next, ensure that the database is created and properly populated as above. Finally, the database connection string needs to be updated to use the proper credentials. Here it is, using the original SQL Server connection script from above:

```
<?php
/*
 * Specify the server and connection string attributes.
 */
$serverName = 'CALEVANS-PC\SQLEXPRESS';
$uid        = 'testuser';
$pwd        = 'testuser';
$database   = 'AdventureWorks';
try {
    $conn = new PDO( "sqlsrv:server=$serverName;database=$database", $uid, $pwd);
[trimmed for brevity]
```

FIGURE 3



6 http://channel9.msdn.com/learn/courses/Azure/SQLAzure/MigratingDatabasestoSQLAzure/Exercise-1-Moving-an-Existing-Database-to-the-Cloud/

We simply update the credentials with the proper information:

```
<?php
/*
 * Specify the server and connection string attributes.
 */
$serverName = 'tcp:servername.database.windows.net,1433';
$uid        = 'servername';
$pwd        = 'testuser';
$database    = 'AdventureWorks';
try {
    $conn = new PDO( "sqlsrv:server=$serverName;database=$database", $uid, $pwd);
[trimmed for brevity]
```

The rest of the connection information, queries, and interactions remain the same. Of course, due to the modifications related to indexes and slight feature differences between SQL Server and SQL Azure, application profiling and review is recommended.

Overall, while the initial database conversion itself is cumbersome, the benefits are significant. SQL Azure offers the features and interface common to a standard SQL Server configuration but adds the high availability and highly scalable infrastructure inherent to the cloud.

FIGURE 4

Migrating from MySQL

It is beyond the intended scope of this document to provide a comprehensive migration guide for those interested in moving databases stored in a MySQL RDBMS into SQL Server. It should go without saying for those that are considering that move that for all but the most trivial databases, pre-planning and attention to detail are the keys to a successful migration. There is a list of comprehensive migration guides located at http://www.microsoft.com/sqlserver/2008/en/us/migration.aspx including 2 for MySQL->SQL Server migrations. The correct one for you will depend on the version of SQL Server to which you are migrating.

While the details are way too numerous to document in this guide, there are several things that are important enough that they need to be discussed.

Users and permissions

For users familiar with the MySQL permissions model, SQL Server may be disorienting at first when dealing with the SQL Server permissions model. First, SQL Server has two separate methods for authenticating users, Windows Authentication and SQL Server Authentication. Since this document deals mainly with databases that will be used on the web, we will ignore Windows Authentication. SQL Server Authentication is most like the MySQL model and is sufficient for web applications. NOTE: SQL Server allows you to specify that both authentication models be active at once. This allows you to use SQL Server Authentication for your web applications while maintaining tighter control on domain-based users with Windows Authentication.

Creating users can be easier or more difficult than MySQL, depending on how you create your users in MySQL. MySQL gives you the ability to create users programmatically using DML to insert users into the users table and manipulating permissions in a similar way. SQL Server requires the use of a tool, like the SQL Server Management Console, to manipulate logins.

In Figure 5, you can see both Windows Authentication users (CalEvans-PC/CalEvans) and SQL Server Authentication users (testuser).

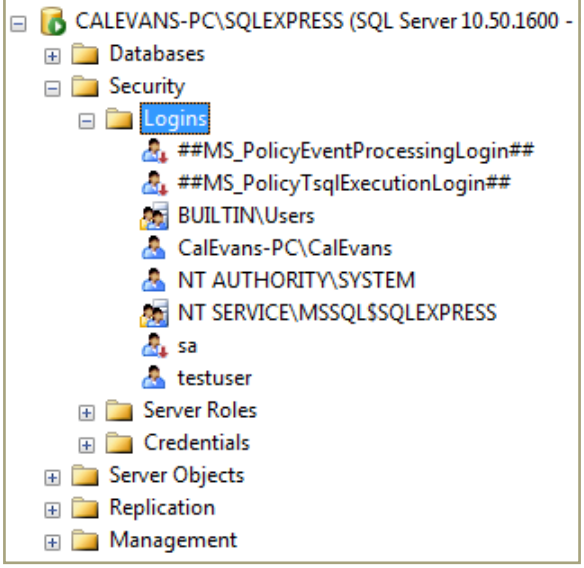
Once you have created the user(s) for your application, you have to assign them roles. SQL Server comes with a series

of default roles - public being the one you would consider for your web application user - and you can add your own to meet your specific needs.

After creating your user and assigning a role, the final step necessary is to grant that user access to the database and schema needed by your application.

Viewed in this context, users are similar to, if only slightly more complicated, than MySQL. The additional complexity gives server administrators more flexibility.

FIGURE 5



Data type issues

What is discussed in this document is a brief overview of the data type issues that developers face when migrating from MySQL to SQL Server. It is not intended to be an exhaustive list, but rather a list of the most common or most important issues.

In the document "Guide to Migrating from MySQL to SQL Server 2008", available at http://www.microsoft.com/sqlserver/2008/en/us/migration.aspx, there is a comprehensive chart that shows MySQL data types and their recommended SQL Server equivalents. This is an excellent place for developers to begin the migration process. Pages 8 through 12 give developers recommendations and hints on converting data types. (as well as a glimpse into the complexity of the task at hand)

The document goes on to list a series of edge cases that may or may not affect developers, depending on the structure of their MySQL database. Some of these deal with the difference in how MySQL and SQL Server handle unsigned integers, NULL values and ENUM data types. Out of all of the additional notes on conversions, no area receives more attention than Date and DateTime fields. The difference in how the two systems process and validate dates and time is significant, and great care should be taken when converting MySQL Date and DateTime fields to SQL Server. MySQL tends to be more forgiving with respect to the actual content of these data types. If developers have set ALLOW_INVALID_DATES, then MySQL will not complain if a date has a year of 0000. SQL Server, on the other hand, does not allow these values, causing migration scripts to fail when attempting to insert these values into Dates.

Another Date gotcha is that MySQL supports a data type of YEAR whereas SQL Server does not.

The final, and possibly the most contentious, issue that developers will deal with when converting dates is the fact

that allowable ranges differ between the two systems. By way of example, SQL Server only allows Year values to be between 1901 and 2155. Inserting a date with a year outside of the valid range will not cause a failure, but SQL Server will truncate the data and insert the date with a year of 0000. Obviously, this resulting data loss could cause serious problems to a system.

Care must not only be taken to make sure that your migration scripts have run properly, but that once they have run error-free, the resulting data is compared to the original to ensure that no data or precision has been lost.

Once you have planned your data migration and thought through the issues, there is a conversion tool you can use to actually move the schema and data. Using the tool does not alleviate the need to properly plan the migration but it will create the scripts for you to automate - and if necessary, replicate - the process. The Microsoft SQL Server Migration Assistant 2008 for MySQL v1.0 CTP1 will help developers, especially with large and complex data migrations - move not only the schema but the actual data from a MySQL instance to a SQL Server instance. As with the migration itself, the tool requires forethought and planning. Unlike many tools that are simple wizards that can create more problems than they solve, SSMA is a tool designed to be used by knowledgeable DBAs. It is not a simple tool to operate and the potential for damage is high. However, in the right hands, SSMA is a powerful tool for ensuring that data and schema are both migrated properly.

Troubleshooting

Logs and Query Plans

For those developers familiar with MySQL, you are undoubtedly familiar with the Slow Query log. The slow query log is an indispensable tool for developers looking to find problems in new systems and squeeze performance out of existing systems. Unfortunately, because of the size and complexity of SQL Server, log analysis is broken out into a module that may or may not be installed on a given server. All is not lost, though. Thanks to a thriving community of SQL developers helping each other, solutions do exist.

In 2008, a developer named Brian Hartstock posted a blog post describing a query he found in another article that identified potential problematic queries. Brian's blog post can be found here:
http://blog.brianhartsock.com/2008/12/16/quick-and-dirty-sql-server-slow-query-log/

The original article on query optimization can be found here:
http://technet.microsoft.com/en-us/magazine/2007.11.sqlquery.aspx

The actual query is:

```
SELECT TOP 20 SUBSTRING(qt.text, (qs.statement_start_offset/2)+1,
         ((CASE qs.statement_end_offset
           WHEN -1 THEN DATALENGTH(qt.text)
          ELSE qs.statement_end_offset
          END - qs.statement_start_offset)/2)+1),
        qs.execution_count,
        qs.total_logical_reads, qs.last_logical_reads,
        qs.min_logical_reads, qs.max_logical_reads,
        qs.total_elapsed_time, qs.last_elapsed_time,
        qs.min_elapsed_time, qs.max_elapsed_time,
        qs.last_execution_time,
        qp.query_plan
FROM sys.dm_exec_query_stats qs
```

```
    CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
    CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
WHERE qt.encrypted=0
    ORDER BY qs.total_logical_reads DESC
```

This query, while complex, will return a list of the 20 most read intensive queries. Most interestingly though, is the last field, query_plan. Clicking on that link inside of the SQL Server Management Studio will bring up a graphic (Figure 6) showing the query and what the server has to do to execute it.

This simple example shows our query used previously to pull data from the AdventureWorks database. More complex queries generate a more complex - and more useful - query plan.

By examining the results for the query above, developers can find queries that are read intensive and thus slower than

FIGURE 6

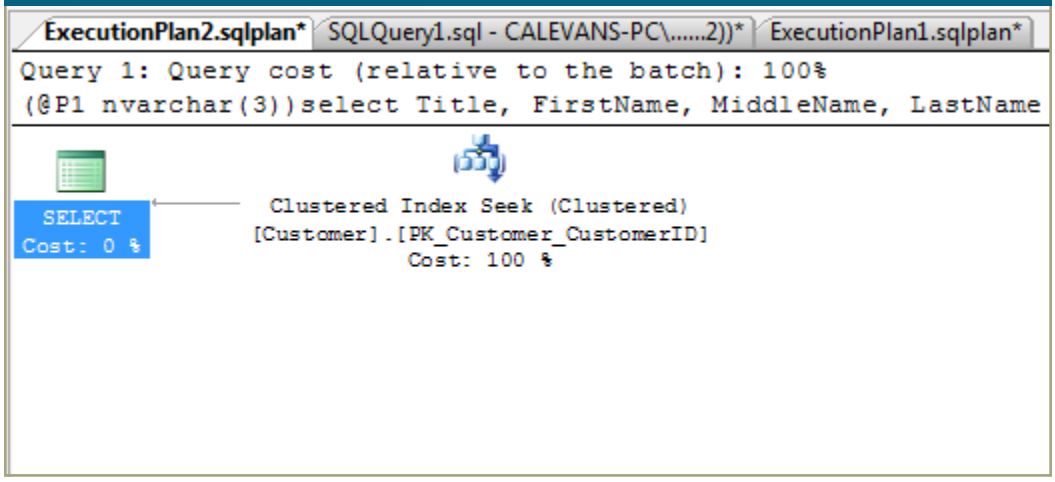
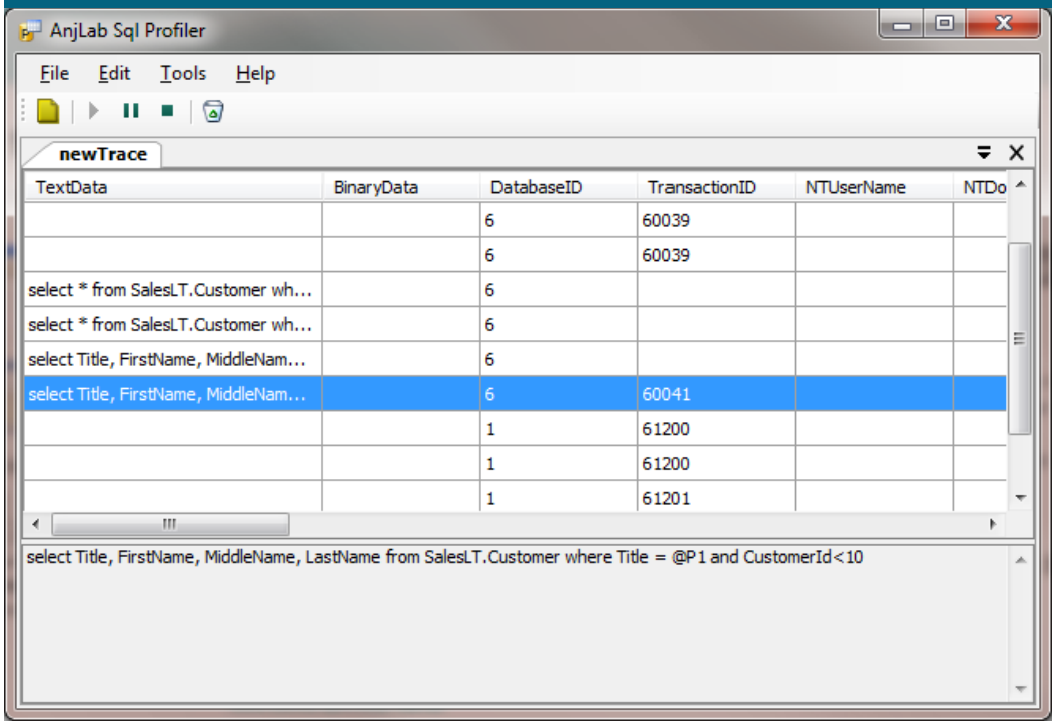


FIGURE 7



other queries. By examining the query plan, developers can understand what is being done and can spot ways that the query can either be simplified or sped up.

Profiling

Another tool of note for developers, especially those working with the Express edition of SQL Server is the AnjLab SQL Profiler (Figure 7). While this will connect to any SQL Server instance, it is specifically designed to work with SQL Server Express as the Express package does not include a profiling tool.

AnjLab allows you to see the events that are taking place on your server in real time. When creating a trace, you have the option of selecting specific types of events to trace or tracing them all. Additionally, you can specify filters on the events using the fields provided. Unlike the query above and the execution plan, it will not help developers solve long-running transactions. It will, however, help spot long running transactions and help developers understand what is happening while the application is running.

AnjLab is an open source project that works with bot SQL Server Express 2005 and 2008. More information and downloads can be found at their homepage, http://sites.google.com/site/sqlprofiler/.

Other versions of SQL Server include a profiling tool in the distribution.

Conclusion

This document has attempted to show PHP developers interested in SQL Server the ease with which it can now be used. For the longest time, PHP and MySQL have been joined at the hip. When developers thought of PHP, they automatically thought of MySQL and vice versa. As technologies change, however, so do mindsets and PHP developers are now beginning to see that there are a variety of storage engines available to them. For environments built on a Windows platform with SQL Server already well established as the corporate data store of choice, introducing another data store makes no sense. It was only an option to be considered while PHP could not reliably connect to SQL Server. Now, however, with not only native drivers but PDO drivers to boot, the idea of introducing MySQL into an existing stack just because applications are being written in PHP is laughable.

As the SQL Server driver for PHP matures over time there will be fewer and fewer items in the "you can't do that" column (currently, most of them boil down to edge cases). We can look forward to PHP being a first class citizen and an equal amongst equals when connecting to SQL Server databases.

Additional Resources and Reading

- SQL Server: www.microsoft.com/sqlserver
- SQL Server PHP Driver: http://msdn.microsoft.com/en-us/sqlserver/cc299381.aspx
- SQL Server PHP Driver Blog: http://blogs.msdn.com/sqlphp
- SQL Server PHP Driver Forum: http://social.technet.microsoft.com/Forums/en-US/sqldriverforphp/threads
- SQL Server PHP Driver Source Code: http://sqlsrvphp.codeplex.com