

User name:

Book: SQL in a Nutshell, 2nd Edition

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

5.7. Bound Parameters

One of the easiest ways to optimize a slow database application that executes many similar SQL statements is to parameterize the most commonly used SQL statements. Parameterization is a way of reusing a SQL statement by writing it with placeholders for frequently changing values. This provides two major benefits: the potential for reduced server roundtrips and better processing efficiency on the database server, since the server doesn't need to parse, plan, and optimize the execution of frequently used SQL statements on every execution.

Parametrized statements are useful in using string or binary values within SQL statements that contain unfriendly characters, such as quote marks or terminal NULL characters that indicate the end of a string. Usage in this way provides security benefits to a database application where user input is used within statements. Not binding a parameter coming from an end user could allow clever modification of the statement in a way that may divulge secret information from the database.



Since there is a slight cost involved in setting up a parameterized statement, it's good practice to only parameterize statements that will be executed at least three times prior to the statement closing. If a statement is only executed once prior to freeing the statement, the costs of preparing the statement and binding the parameters could result in a performance net loss to the application.

5.7.1. ADO.NET Bound Parameters

The following C# code fragment executes a SQL *INSERT* statement that adds new sales to the **sales** table in the **pubs** database. The *INSERT* statement is parameterized to provide better performance, since the statement object needs to be parsed only once on the server.

```
// Create a Command object for the SQL statement
Statement statement = connection.CreateCommand( );
statement.CommandText =
    "INSERT INTO SALES(stor_id,
                      ord_num,
                      ord_date,
                      qty,
                      payterms,
                      title_id) " +
    "VALUES(@stor_id, @ord_num, @ord_date, @qty, @payterms, @title_id)";

//Prepare the statement on the server
statement.Prepare( );

// Declare parameters that will be bound
{Odbc|OleDb|Sql}Parameter stor_id, ord_num, ord_date,
                          qty, payterms, title_id;

stor_id = statement.Parameters.Add( "@stor_id", DbType.String );
ord_num = statement.Parameters.Add( "@ord_num", DbType.String );
ord_date = statement.Parameters.Add( "@ord_date", DbType.DateTime);
qty = statement.Parameters.Add( "@qty", DbType.Int16 );
payterms = statement.Parameters.Add( "@payterms", DbType.String );
title_id = statement.Parameters.Add( "@title_id", DbType.String );

while( GetNextSale(stor_id, ord_num, ord_date, qty, payterms, title_id) )
{
    // Execute the statement
    int result = statement.ExecuteNonQuery( );
    if( result != 1 )
    {

```

```

        // If result isn't 1, then the insert failed.
        System.Console.WriteLine( "The INSERT failed." );
        break;
    }
}

```

5.7.1.1. Use the following steps to execute statements with bound parameters in ADONET:

1. As done in previous sections, we create an ADO.NET `Command` object and assign a SQL statement to it. The difference for using bound parameters within the statement is in the *VALUES* clause of the *INSERT* statement. Contained within the *VALUES* clause are six named placeholders for the parameters that will be later bound to the `Command` object. In ADO.NET the placeholders begin with the `@` symbol and are followed by an identifier that is unique amongst all placeholders in the statement.

```

Statement statement = connection.CreateCommand( );
statement.CommandText =
    "INSERT INTO SALES(stor_id,
                      ord_num,
                      ord_date,
                      qty,
                      payterms,
                      title_id) " +
    "VALUES(@stor_id, @ord_num, @ord_date, @qty, @payterms, @title_id)";

```

2. Invoke the `Prepare` method on the `Command` object to prepare the SQL statement for execution. This notifies the database layer that the `Command` object will be executed with bound parameters.

```
statement.Prepare( );
```

3. Declare the parameter objects using the parameter type that matches the `Command` object: `OdbcParameter`, `OleDbParameter`, or `SqlParameter`. After declaring the parameters, create parameter objects by invoking the `Add` method of the `Command` object's `Parameter` collection and assign the return value to the `Parameter` object. The first argument to the `Add` method is the name of the placeholder that the parameter will map to during execution. In the example below, the `stor_id Parameter` object will map to the first item in the *VALUES* clause, which has the placeholder named `@stor_id`. The second argument in the `Add` method is the column type that the placeholder maps to on the server. Look to [Table 5-8](#) for a list of frequently used types.

```

{Odbc|OleDb|Sql}Parameter stor_id, ord_num, ord_date,
                           qty, payterms, title_id;

stor_id = statement.Parameters.Add( "@stor_id", DbType.String );
ord_num = statement.Parameters.Add( "@ord_num", DbType.String );
ord_date = statement.Parameters.Add( "@ord_date", DbType.DateTime);
qty = statement.Parameters.Add( "@qty", DbType.Int16 );
payterms = statement.Parameters.Add( "@payterms", DbType.String );
title_id = statement.Parameters.Add( "@title_id", DbType.String );

```

4. In this example, the `Parameter` objects are assigned a value by the user-defined `GetNextSale` function call, which could be implemented like this:

```

static bool GetNextSale(SqlParameter stor_id,
                       SqlParameter ord_num,
                       SqlParameter ord_date,
                       SqlParameter qty,
                       SqlParameter payterms,
                       SqlParameter title_id)
{
    // Omitted is the code that would
    // Read a sale record from a file, or user input, etc.

    // If there are no more sale records, return false.
    if( !more_records ) return false;

    // Assign values to the parameter objects
    stor_id.Value = 1234;
}

```

```

ord_num.Value = "ABCD.123";
ord_date.Value = new DateTime(2003,2,24);
qty.Value = 50;
payterms.Value = "Net 60";
title_id.Value = "SD2043";
return true;
}

```

Notice that the parameters are assigned a value by assigning directly to the `Value` property on the `Parameter` object. The `stor_id` parameter object is assigned a value of 1234, whereas `ord_date` is assigned a C# ADO.NET `DateTime` object. The function returns false if there are no more sales to insert into the table; otherwise the function returns true.

Combined with a `while` loop, the program will continue to insert new sales into the database until the `GetNextSale` function runs out of new records to process.

```

while(GetNextSale(stor_id, ord_num, ord_date, qty, payterms, title_id) )
{

```

- Invoking the `ExecuteNonQuery` method on the `Command` object executes the `INSERT` statement, with the bound parameter values replaced for their corresponding placeholders. The `ExecuteNonQuery` method returns the number of rows affected, which will be 1 in this case on a successful single-row insert. This return value is used in error handling and the application will exit if the statement should ever fail to execute.

```

// Execute the statement
int result = statement.ExecuteNonQuery( );
if( result != 1 )
{
    // If result isn't 1, then the insert failed.
    System.Console.WriteLine( "The INSERT failed." );
    break;
}

```

Table 5-8. Frequently used parameter object types

DbType object type	Description
<code>AnsiString</code>	Variable-length, non-Unicode character string between 1 and 8,000 characters.
<code>AnsiStringFixedLength</code>	Fixed-length, non-Unicode character string.
<code>Binary</code>	Variable-length binary string between 1 and 8,000 bytes.
<code>Boolean</code>	Represents a true or false value.
<code>Byte</code>	Unsigned integer value ranging 0 to 255.
<code>Currency</code>	Currency value ranging from -2^{63} to $2^{63}-1$.
<code>DateTime</code>	Date and time value.
<code>Decimal</code>	Numeric value ranging from 10^{-28} to $7.9 \cdot 10^{28}$ with about 28 significant digits.
<code>Double</code>	Floating point type ranging from $5.0 \cdot 10^{-324}$ to $1.7 \cdot 10^{308}$ with about 15 digits.
<code>Int{16,32,64}</code>	Signed integer; the number suffix indicates the number of bits of precision. For example, <code>Int32</code> is a 32-bit integer.
<code>Single</code>	Floating point type ranging from $5.0 \cdot 10^{-324}$ to $1.7 \cdot 10^{308}$, with about 15 digits.
<code>String</code>	Variable-length, Unicode character string.
<code>StringFixedLength</code>	Fixed-length, Unicode character string.
<code>UInt{16,32,64}</code>	Unsigned integer; the number suffix indicates the number of bits of precision. For example, <code>UInt32</code> is a 32-bit unsigned integer.

5.7.2. Binding Parameters with JDBC

The following Java code fragment executes a SQL *INSERT* statement that adds new sales to the **sales** table in the **pubs** database. The *INSERT* statement is parameterized to provide better performance.

```
// Create a Command object for the SQL statement
PreparedStatement statement = connection.prepareStatement(
    "INSERT INTO SALES(stor_id,
                        ord_num,
                        ord_date,
                        qty,
                        payterms,
                        title_id) " +
    "VALUES(?, ?, ?, ?, ?, ?)" );

while( getNextSale(statement) )
{
    // Execute the statement
    int result = statement.executeUpdate( );
    if( result != 1 )
    {
        // If result isn't 1, then the insert failed.
        System.out.println( "The INSERT failed." );
        break;
    }
}
```

5.7.2.1. Use the following steps to execute statements with bound parameters in JDBC:

1. Create a JDBC `PreparedStatement` object and pass the parameterized SQL statement into its constructor. The difference for using bound parameters within the statement is in the *VALUES* clause of the *INSERT* statement. Contained within the *VALUES* clause are six placeholders (the question marks) for the parameters that will later be bound to the `PreparedStatement` object.

```
PreparedStatement statement = connection.prepareStatement(
    "INSERT INTO SALES(stor_id,
                        ord_num,
                        ord_date,
                        qty,
                        payterms,
                        title_id) " +
    "VALUES(?, ?, ?, ?, ?, ?)" );
```

1. In this example, the parameters are assigned a value by the user-defined `getNextSale` function call, which could be implemented like this:

```
static boolean getNextSale( PreparedStatement statement )
    throws SQLException
{
    // Omitted is the code that would
    // Read a sale record from a file, or user input, etc.

    // If there are no more sale records, return false.
    if( !more_records ) return false;

    statement.setString(1, "1234");
    statement.setString(2, "ABCD.123");
    statement.setDate(3, new java.sql.Date(2003, 2, 24));
    statement.setInt(4,50);
    statement.setString(5, "Net 60");
    statement.setString(6, "SD2043");
    return true;
}
```

1. Each binding position is referenced by its ordinal position in the SQL statement, with the first position starting at 1. The values are assigned to placeholders using the `set` methods found on the `PreparedStatement` object. Table 5-9 contains a list of frequently used `set` methods.
2. The function returns false if there are no more sales to insert into the table; otherwise the function

returns true.

3. Combined with a `while` loop, the program will continue to insert new sales into the database until the `getNextSale` function runs out of new records to process.

```
while( getNextSale(statement) )
{
```

1. Invoking the `executeUpdate` method on the `PreparedStatement` object executes the `INSERT` statement, with the bound parameter values replaced for their corresponding placeholders. The `executeUpdate` method returns the number of rows affected, which will be 1 in this case on a successful single-row insert. This return value is used in error handling and the application will exit if the statement should ever fail to execute.

```
// Execute the statement
int result = statement.executeUpdate( );
if( result != 1 )
{
    // If result isn't 1, then the insert failed.
    System.out.println( "The INSERT failed." );
    break;
}
}
```

Table 5-9. Frequently used PreparedStatement set methods

Method name	Description
<code>setBlob(int i, Blob value)</code>	Sets the placeholder at position <i>i</i> to the <code>Blob</code> contained in <i>value</i> .
<code>setBoolean(int i, boolean value)</code>	Sets the placeholder at position <i>i</i> to the <code>boolean</code> contained in <i>value</i> .
<code>setByte(int i, byte value)</code>	Sets the placeholder at position <i>i</i> to the <code>byte</code> contained in <i>value</i> .
<code>setClob(int i, Clob value)</code>	Sets the placeholder at position <i>i</i> to the <code>Clob</code> contained in <i>value</i> .
<code>setDate(int i, Date value[, Calendar cal])</code>	Sets the placeholder at position <i>i</i> to the <code>Date</code> contained in <i>value</i> . If <i>cal</i> is provided, it'll be used to interpret the <code>Date value</code> .
<code>setDouble(int i, double value)</code>	Sets the placeholder at position <i>i</i> to the <code>double</code> contained in <i>value</i> .
<code>setFloat(int i, float value)</code>	Sets the placeholder at position <i>i</i> to the <code>float</code> contained in <i>value</i> .
<code>setInt(int i, int value)</code>	Sets the placeholder at position <i>i</i> to the <code>int</code> contained in <i>value</i> .
<code>setLong(int i, long value)</code>	Sets the placeholder at position <i>i</i> to the <code>long</code> contained in <i>value</i> .
<code>setNull(int i, int v)</code>	Sets the placeholder at position <i>i</i> to a NULL value when <i>v</i> is true.
<code>setString(int i, String value)</code>	Sets the placeholder at position <i>i</i> to the <code>String</code> contained in <i>value</i> .
<code>setTimestamp(int i, Timestamp value[, Calendar cal])</code>	Sets the placeholder at position <i>i</i> to the <code>Timestamp</code> contained in <i>value</i> . If <i>cal</i> is provided, it'll be used to interpret the <code>Timestamp value</code> .