FREE eBook

LEARNING Dapper.NET

Free unaffiliated eBook created from **Stack Overflow contributors.**

#dapper

Table of Contents

About 1

Chapter 1: Getting started with Dapper.NET 2

Remarks 2

What is Dapper? 2

How do I get it? 2

Common Tasks 2

Versions 2

Examples 2

Install Dapper from Nuget 2

Using Dapper in C# 3

Using Dapper in LINQPad 3

Chapter 2: Basic Querying 5

Syntax 5

Parameters 5

Examples 5

Querying for a static type 5

Querying for dynamic types 6

Query with Dynamic Parameters 6

Chapter 3: Bulk inserts 7

Remarks 7

Examples 7

Async Bulk Copy 7

Bulk Copy 7

Chapter 4: Dynamic Parameters 9

Examples 9

Basic Usage 9

Dynamic Parameters in Dapper 9

Using a template object 9

Chapter 5: Executing Commands 11

Examples 11

Execute a command that returns no results 11
Stored Procedures 11
Simple usage 11
Input, Output and Return parameters 11
Table Valued Parameters 11
Chapter 6: Handling Nulls 13
Examples 13
null vs DBNull 13
Chapter 7: Multimapping 14
Syntax 14
Parameters 14
Examples 15
Simple multi-table mapping 15
One-to-many mapping 16
Mapping more than 7 types 18
Custom Mappings 19
Chapter 8: Multiple Results 21
Syntax 21
Parameters 21
Examples 21
Base Multiple Results Example 21
Chapter 9: Parameter Syntax Reference 22
Parameters 22
Remarks 22
Examples 22
Basic Parameterized SQL 22
Using your Object Model 23
Stored Procedures 23
Value Inlining 24
List Expansions 24
Performing Operations Against Multiple Sets of Input 25
Pseudo-Positional Parameters (for providers that don't support named parameters) 26

Chapter 10: Temp Tables 27
Examples 27
Temp Table that exists while the connection remains open 27
How to work with temp tables 27
Chapter 11: Transactions 29
Syntax 29
Examples 29
Using a Transaction 29
Speed up inserts 30
Chapter 12: Type Handlers 31
Remarks 31
Examples 31
Converting varchar to IHtmlString 31
Installing a TypeHandler 31
Chapter 13: Using Async 32
Examples 32
Calling a Stored Procedure 32
Calling a stored procedure and ignoring the result 32
Chapter 14: Using DbGeography and DbGeometry 33
Examples 33
Configuration required 33
Using geometry and geography 33
Credits 35

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: dapper-net

It is an unofficial and free Dapper.NET ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Dapper.NET.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Dapper.NET

Remarks

What is Dapper?

Dapper is a micro-ORM for .Net that extends your `IDbConnection`, simplifying query setup, execution, and result-reading.

How do I get it?

- github: https://github.com/StackExchange/dapper-dot-net
- NuGet: https://www.nuget.org/packages/Dapper

Common Tasks

- Basic Querying
- Executing Commands

Versions

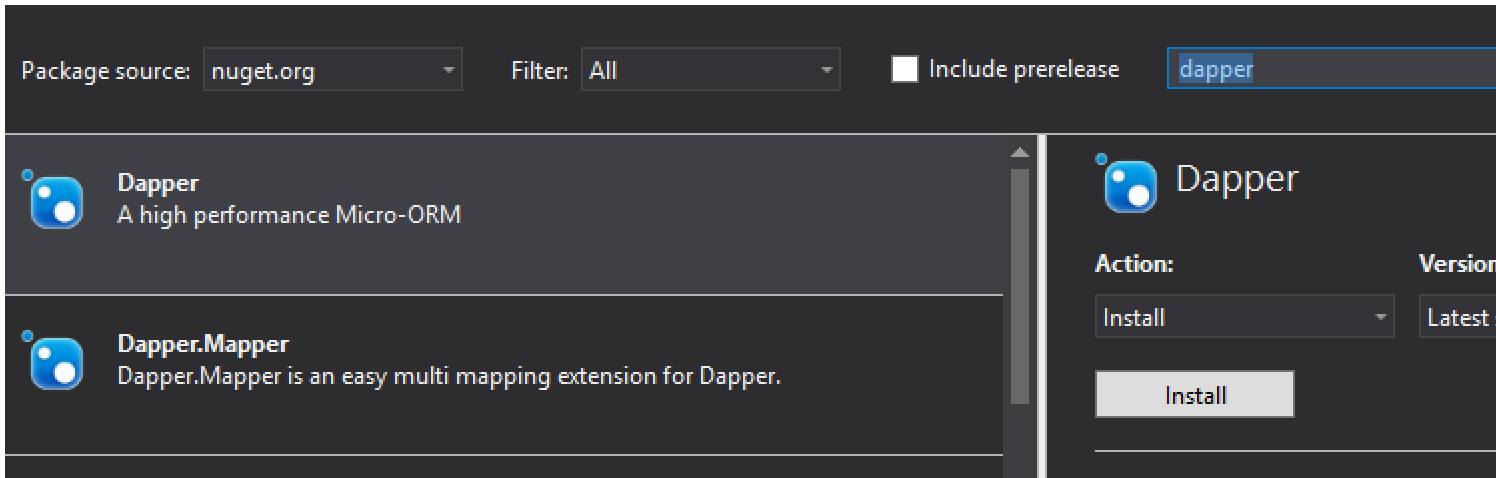
Version	Notes	Release Date
1.50.0	core-clr / asp.net 5.0 build against RTM	2016-06-29
1.42.0		2015-05-06
1.40.0		2015-04-03
1.30.0		2014-08-14
1.20.0		2014-05-08
1.10.0		2012-06-27
1.0.0		2011-04-14

Examples

Install Dapper from Nuget

Either search in the Visual Studio GUI:

Tools > NuGet Package Manager > Manage Packages for Solution... (Visual Studio 2015)



Or run this command in a Nuget Power Shell instance to install the latest stable version

```
Install-Package Dapper
```

Or for a specific version

```
Install-Package Dapper -Version 1.42.0
```

Using Dapper in C#

```
using System.Data;
using System.Linq;
using Dapper;

class Program
{
    static void Main()
    {
        using (IDbConnection db = new
SqlConnection("Server=myServer;Trusted_Connection=true"))
        {
            db.Open();
            var result = db.Query<string>("SELECT 'Hello World'").Single();
            Console.WriteLine(result);
        }
    }
}
```

Wrapping the connection in a `Using` block will close the connection

Using Dapper in LINQPad

LINQPad is great for testing database queries and includes NuGet integration. To use Dapper in LINQPad press **F4** to open the Query Properties and then select **Add NuGet**. Search for **dapper dot net** and select **Add To Query**. You will also want to click **Add namespaces** and highlight Dapper to include the Extension Methods in your LINQPad query.

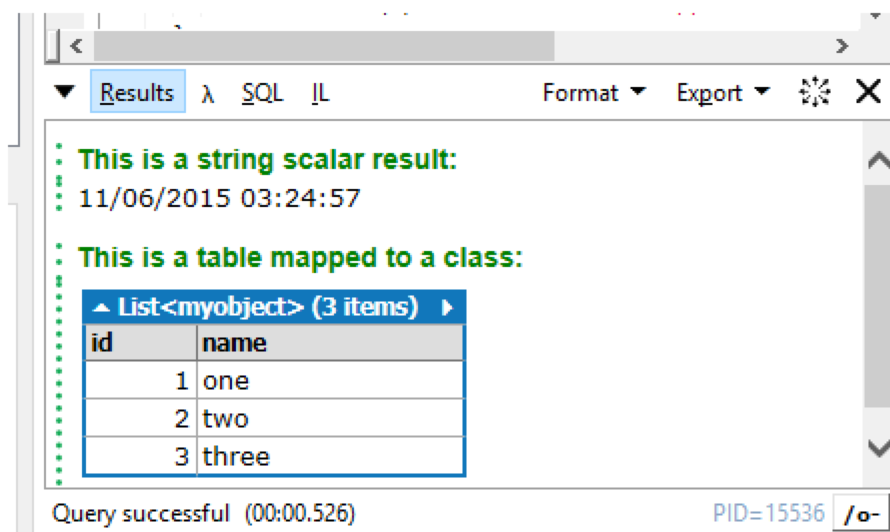
Once Dapper is enabled you can change the Language drop down to **C# Program**, map query results to C# classes, and use the .Dump() method to inspect the results:

```
void Main()
{
    using (IDbConnection db = new SqlConnection("Server=myServer;Trusted_Connection=true")){
        db.Open();
        var scalar = db.Query<string>("SELECT GETDATE()").SingleOrDefault();
        scalar.Dump("This is a string scalar result:");

        var results = db.Query<myobject>(@"
        SELECT * FROM (
        VALUES (1,'one'),
            (2,'two'),
            (3,'three')
        ) AS mytable(id,name)");
        results.Dump("This is a table mapped to a class:");
    }
}

// Define other methods and classes here
class myobject {
    public int id { get; set; }
    public string name { get; set; }
}
```

The results when executing the program would look like this:



Read Getting started with Dapper.NET online: https://riptutorial.com/dapper/topic/2/getting-started-with-dapper-net

Chapter 2: Basic Querying

Syntax

- public static IEnumerable<T> Query<T>(this IDbConnection cnn, string sql, object param = null, SqlTransaction transaction = null, bool buffered = true)
- public static IEnumerable<dynamic> Query (this IDbConnection cnn, string sql, object param = null, SqlTransaction transaction = null, bool buffered = true)

Parameters

Parameter	Details
cnn	Your database connection, which must already be open.
sql	Command to execute.
param	Object to extract parameters from.
transaction	Transaction which this query is a part of, if any.
buffered	Whether or not to buffer reading the results of the query. This is an optional parameter with the default being true. When buffered is true, the results are buffered into a `List<T>` and then returned as an `IEnumerable<T>` that is safe for multiple enumeration. When buffered is false, the sql connection is held open until you finish reading allowing you to process a single row at time in memory. Multiple enumerations will spawn additional connections to the database. While buffered false is highly efficient for reducing memory usage if you only maintain very small fragments of the records returned it has a sizeable performance overhead compared to eagerly materializing the result set. Lastly if you have numerous concurrent unbuffered sql connections you need to consider connection pool starvation causing requests to block until connections become available.

Examples

Querying for a static type

For types known at compile-time, use a generic parameter with `Query<T>`.

```
public class Dog
{
    public int? Age { get; set; }
    public Guid Id { get; set; }
    public string Name { get; set; }
```

```
    public float? Weight { get; set; }

    public int IgnoredProperty { get { return 1; } }
}

//
IDBConnection db = /* ... */;

var @params = new { age = 3 };
var sql = "SELECT * FROM dbo.Dogs WHERE Age = @age";

IEnumerable<Dog> dogs = db.Query<Dog>(sql, @params);
```

Querying for dynamic types

You can also query dynamically if you leave off the generic type.

```
IDBConnection db = /* ... */;
IEnumerable<dynamic> result = db.Query("SELECT 1 as A, 2 as B");

var first = result.First();
int a = (int)first.A; // 1
int b = (int)first.B; // 2
```

Query with Dynamic Parameters

```
var color = "Black";
var age = 4;

var query = "Select * from Cats where Color = :Color and Age > :Age";
var dynamicParameters = new DynamicParameters();
dynamicParameters.Add("Color", color);
dynamicParameters.Add("Age", age);

using (var connection = new SqlConnection(/* Your Connection String Here */))
{
    IEnumerable<dynamic> results = connection.Query(query, dynamicParameters);
}
```

Read Basic Querying online: https://riptutorial.com/dapper/topic/3/basic-querying

Chapter 3: Bulk inserts

Remarks

The `WriteToServer` and `WriteToServerAsync` have overloads that accept IDataReader (seen in the examples), DataTable, and DataRow arrays (`DataRow[]`) as the source of the data for the Bulk Copy.

Examples

Async Bulk Copy

This sample uses a `ToDataReader` method described here Creating a Generic List DataReader for SqlBulkCopy.

This can also be done using non-async methods.

```
public class Widget
{
    public int WidgetId {get;set;}
    public string Name {get;set;}
    public int Quantity {get;set;}
}

public async Task AddWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        await conn.OpenAsync();

        using(var bulkCopy = new SqlBulkCopy(conn)) {
            bulkCopy.BulkCopyTimeout = SqlTimeoutSeconds;
            bulkCopy.BatchSize = 500;
            bulkCopy.DestinationTableName = "Widgets";
            bulkCopy.EnableStreaming = true;

            using(var dataReader = widgets.ToDataReader())
            {
                await bulkCopy.WriteToServerAsync(dataReader);
            }
        }
    }
}
```

Bulk Copy

This sample uses a `ToDataReader` method described here Creating a Generic List DataReader for SqlBulkCopy.

This can also be done using async methods.

```
public class Widget
{
    public int WidgetId {get;set;}
    public string Name {get;set;}
    public int Quantity {get;set;}
}

public void AddWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        conn.Open();

        using(var bulkCopy = new SqlBulkCopy(conn)) {
            bulkCopy.BulkCopyTimeout = SqlTimeoutSeconds;
            bulkCopy.BatchSize = 500;
            bulkCopy.DestinationTableName = "Widgets";
            bulkCopy.EnableStreaming = true;

            using(var dataReader = widgets.ToDataReader())
            {
                bulkCopy.WriteToServer(dataReader);
            }
        }
    }
}
```

Read Bulk inserts online: https://riptutorial.com/dapper/topic/6279/bulk-inserts

Chapter 4: Dynamic Parameters

Examples

Basic Usage

It isn't always possible to neatly package all the parameters up in a single object / call. To help with more complicated scenarios, dapper allows the `param` parameter to be an `IDynamicParameters` instance. If you do this, your custom `AddParameters` method is called at the appropriate time and handed the command to append to. In most cases, however, it is sufficient to use the pre-existing `DynamicParameters` type:

```
var p = new DynamicParameters(new { a = 1, b = 2 });
p.Add("c", dbType: DbType.Int32, direction: ParameterDirection.Output);
connection.Execute(@"set @c = @a + @b", p);
int updatedValue = p.Get<int>("@c");
```

This shows:

- (optional) population from an existing object
- (optional) adding additional parameters on the fly
- passing the parameters to the command
- retrieving any updated value after the command has finished

Note that due to how RDBMS protocols work, it is usually only reliable to obtain updated parameter values **after** any data (from a `Query` or QueryMultiple` operation) has been **fully** consumed (for example, on SQL Server, updated parameter values are at the *end* of the TDS stream).

Dynamic Parameters in Dapper

```
connection.Execute(@"some Query with @a,@b,@c", new
{a=somevalueOfa,b=somevalueOfb,c=somevalueOfc});
```

Using a template object

You can use an instance of an object to form your parameters

```
public class SearchParameters {
  public string SearchString { get; set; }
  public int Page { get; set; }
}

var template= new SearchParameters {
  SearchString = "Dapper",
  Page = 1
};
```

```
var p = new DynamicParameters(template);
```

You can also use an anonymous object or a `Dictionary`

Read Dynamic Parameters online: https://riptutorial.com/dapper/topic/12/dynamic-parameters

Chapter 5: Executing Commands

Examples

Execute a command that returns no results

```
IDBConnection db = /* ... */
var id = /* ... */

db.Execute(@"update dbo.Dogs set Name = 'Beowoof' where Id = @id",
   new { id });
```

Stored Procedures

Simple usage

Dapper fully supports stored procs:

```
var user = conn.Query<User>("spGetUser", new { Id = 1 },
                            commandType: CommandType.StoredProcedure)
           .SingleOrDefault();
```

Input, Output and Return parameters

If you want something more fancy, you can do:

```
var p = new DynamicParameters();
p.Add("@a", 11);
p.Add("@b",
      dbType: DbType.Int32,
      direction: ParameterDirection.Output);
p.Add("@c",
      dbType: DbType.Int32,
      direction: ParameterDirection.ReturnValue);

conn.Execute("spMagicProc", p,
             commandType: CommandType.StoredProcedure);

var b = p.Get<int>("@b");
var c = p.Get<int>("@c");
```

Table Valued Parameters

If you have a stored procedure that accepts a Table Valued Parameter, you need to pass a DataTable which has the same structure as the table type in SQL Server has. Here's a definition for a table type and procedure utilizing it:

```
CREATE TYPE [dbo].[myUDTT] AS TABLE([i1] [int] NOT NULL);
GO
CREATE PROCEDURE myProc(@data dbo.myUDTT readonly) AS
SELECT i1 FROM @data;
GO
/*
-- optionally grant permissions as needed, depending on the user you execute this with.
-- Especially the GRANT EXECUTE ON TYPE is often overlooked and can cause problems if omitted.
GRANT EXECUTE ON TYPE::[dbo].[myUDTT] TO [user];
GRANT EXECUTE ON dbo.myProc TO [user];
GO
*/
```

To call that procedure from within c#, you need to do the following:

```
// Build a DataTable with one int column
DataTable data = new DataTable();
data.Columns.Add("i1", typeof(int));
// Add two rows
data.Rows.Add(1);
data.Rows.Add(2);

var q = conn.Query("myProc", new {data}, commandType: CommandType.StoredProcedure);
```

Read Executing Commands online: https://riptutorial.com/dapper/topic/5/executing-commands

Chapter 6: Handling Nulls

Examples

null vs DBNull

In ADO.NET, correctly handling `null` is a constant source of confusion. The key point in dapper is that *you don't have to*; it deals with it all internally.

- parameter values that are `null` are correctly sent as `DBNull.Value`
- values read that are `null` are presented as `null`, or (in the case of mapping to a known type) simply ignored (leaving their type-based default)

It just works:

```
string name = null;
int id = 123;
connection.Execute("update Customer set Name=@name where Id=@id",
    new {id, name});
```

Read Handling Nulls online: https://riptutorial.com/dapper/topic/13/handling-nulls

Chapter 7: Multimapping

Syntax

- `public static IEnumerable<TReturn> Query<TFirst, TSecond, TReturn>(this IDbConnection cnn, string sql, Func<TFirst, TSecond, TReturn> map, object param = null, IDbTransaction transaction = null, bool buffered = true, string splitOn = "Id", int? commandTimeout = null, CommandType? commandType = null)`
- `public static IEnumerable<TReturn> Query<TFirst, TSecond, TThird, TFourth, TFifth, TSixth, TSeventh, TReturn>(this IDbConnection cnn, string sql, Func<TFirst, TSecond, TThird, TFourth, TFifth, TSixth, TSeventh, TReturn> map, object param = null, IDbTransaction transaction = null, bool buffered = true, string splitOn = "Id", int? commandTimeout = null, CommandType? commandType = null)`
- `public static IEnumerable<TReturn> Query<TReturn>(this IDbConnection cnn, string sql, Type[] types, Func<object[], TReturn> map, object param = null, IDbTransaction transaction = null, bool buffered = true, string splitOn = "Id", int? commandTimeout = null, CommandType? commandType = null)`

Parameters

Parameter	Details
cnn	Your database connection, which must already be open.
sql	Command to execute.
types	Array of types in the record set.
map	`Func<>` that handles construction of the return result.
param	Object to extract parameters from.
transaction	Transaction which this query is a part of, if any.
buffered	Whether or not to buffer reading the results of the query. This is an optional parameter with the default being true. When buffered is true, the results are buffered into a `List<T>` and then returned as an `IEnumerable<T>` that is safe for multiple enumeration. When buffered is false, the sql connection is held open until you finish reading allowing you to process a single row at time in memory. Multiple enumerations will spawn additional connections to the database. While buffered false is highly efficient for reducing memory usage if you only maintain very small fragments of the records returned it has a sizeable performance overhead compared to eagerly materializing the result set. Lastly if you have numerous concurrent unbuffered sql connections you need to consider connection pool starvation causing requests to block until connections become available.
splitOn	The Field we should split and read the second object from (default: id). This can be a comma delimited list when more than 1 type is contained

Parameter	Details
	in a record.
commandTimeout	Number of seconds before command execution timeout.
commandType	Is it a stored proc or a batch?

Examples

Simple multi-table mapping

Let's say we have a query of the remaining horsemen that needs to populate a Person class.

Name	Born	Residence
Daniel Dennett	1942	United States of America
Sam Harris	1967	United States of America
Richard Dawkins	1941	United Kingdom

```
public class Person
{
    public string Name { get; set; }
    public int Born { get; set; }
    public Country Residience { get; set; }
}

public class Country
{
    public string Residence { get; set; }
}
```

We can populate the person class as well as the Residence property with an instance of Country using an overload `Query<>` that takes a `Func<>` that can be used to compose the returned instance. The `Func<>` can take up to 7 input types with the final generic argument always being the return type.

```
var sql = @"SELECT 'Daniel Dennett' AS Name, 1942 AS Born, 'United States of America' AS
Residence
UNION ALL SELECT 'Sam Harris' AS Name, 1967 AS Born, 'United States of America' AS Residence
UNION ALL SELECT 'Richard Dawkins' AS Name, 1941 AS Born, 'United Kingdom' AS Residence";

var result = connection.Query<Person, Country, Person>(sql, (person, country) => {
        if(country == null)
        {
            country = new Country { Residence = "" };
        }
        person.Residience = country;
        return person;
    },
```

```
    splitOn: "Residence");
```

Note the use of the `splitOn: "Residence"` argument which is the 1st column of the next class type to be populated (in this case `Country`). Dapper will automatically look for a column called *Id* to split on but if it does not find one and `splitOn` is not provided a `System.ArgumentException` will be thrown with a helpful message. So although it is optional you will usually have to supply a `splitOn` value.

One-to-many mapping

Let's look at a more complex example that contains a one-to-many relationship. Our query will now contain multiple rows containing duplicate data and we will need to handle this. We do this with a lookup in a closure.

The query changes slightly as do the example classes.

Id	Name	Born	CountryId	CountryName	BookId	BookName
1	Daniel Dennett	1942	1	United States of America	1	Brainstorms
1	Daniel Dennett	1942	1	United States of America	2	Elbow Room
2	Sam Harris	1967	1	United States of America	3	The Moral Landscape
2	Sam Harris	1967	1	United States of America	4	Waking Up: A Guide to Spirituality Without Religion
3	Richard Dawkins	1941	2	United Kingdom	5	The Magic of Reality: How We Know What`s Really True
3	Richard Dawkins	1941	2	United Kingdom	6	An Appetite for Wonder: The Making of a Scientist

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Born { get; set; }
    public Country Residience { get; set; }
    public ICollection<Book> Books { get; set; }
}

public class Country
{
```

```
    public int CountryId { get; set; }
    public string CountryName { get; set; }
}

public class Book
{
    public int BookId { get; set; }
    public string BookName { get; set; }
}
```

The dictionary`remainingHorsemen` will be populated with fully materialized instances of the person objects. For each row of the query result the mapped values of instances of the types defined in the lambda arguments are passed in and it is up to you how to handle this.

```
            var sql = @"SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS 
CountryId, 'United States of America' AS CountryName, 1 AS BookId, 'Brainstorms' AS BookName
UNION ALL SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS CountryId, 'United 
States of America' AS CountryName, 2 AS BookId, 'Elbow Room' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States 
of America' AS CountryName, 3 AS BookId, 'The Moral Landscape' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States 
of America' AS CountryName, 4 AS BookId, 'Waking Up: A Guide to Spirituality Without Religion' 
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United 
Kingdom' AS CountryName, 5 AS BookId, 'The Magic of Reality: How We Know What`s Really True' 
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United 
Kingdom' AS CountryName, 6 AS BookId, 'An Appetite for Wonder: The Making of a Scientist' AS 
BookName";

var remainingHorsemen = new Dictionary<int, Person>();
connection.Query<Person, Country, Book, Person>(sql, (person, country, book) => {
    //person
    Person personEntity;
    //trip
    if (!remainingHorsemen.TryGetValue(person.Id, out personEntity))
    {
        remainingHorsemen.Add(person.Id, personEntity = person);
    }

    //country
    if(personEntity.Residience == null)
    {
        if (country == null)
        {
            country = new Country { CountryName = "" };
        }
        personEntity.Residience = country;
    }

    //books
    if(personEntity.Books == null)
    {
        personEntity.Books = new List<Book>();
    }

    if (book != null)
    {
        if (!personEntity.Books.Any(x => x.BookId == book.BookId))
```

```
        {
            personEntity.Books.Add(book);
        }
    }

    return personEntity;
},
splitOn: "CountryId,BookId");
```

Note how the `splitOn` argument is a comma delimited list of the first columns of the next type.

Mapping more than 7 types

Sometimes the number of types you are mapping exceeds the 7 provided by the Func<> that does the construction.

Instead of using the `Query<>` with the generic type argument inputs, we will provide the types to map to as an array, followed by the mapping function. Other than the initial manual setting and casting of the values, the rest of the function does not change.

```
            var sql = @"SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS
CountryId, 'United States of America' AS CountryName, 1 AS BookId, 'Brainstorms' AS BookName
UNION ALL SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS CountryId, 'United
States of America' AS CountryName, 2 AS BookId, 'Elbow Room' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States
of America' AS CountryName, 3 AS BookId, 'The Moral Landscape' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States
of America' AS CountryName, 4 AS BookId, 'Waking Up: A Guide to Spirituality Without Religion'
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United
Kingdom' AS CountryName, 5 AS BookId, 'The Magic of Reality: How We Know What`s Really True'
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United
Kingdom' AS CountryName, 6 AS BookId, 'An Appetite for Wonder: The Making of a Scientist' AS
BookName";

var remainingHorsemen = new Dictionary<int, Person>();
connection.Query<Person>(sql,
    new[]
    {
        typeof(Person),
        typeof(Country),
        typeof(Book)
    }
    , obj => {

        Person person = obj[0] as Person;
        Country country = obj[1] as Country;
        Book book = obj[2] as Book;

        //person
        Person personEntity;
        //trip
        if (!remainingHorsemen.TryGetValue(person.Id, out personEntity))
        {
            remainingHorsemen.Add(person.Id, personEntity = person);
```

```
        }

        //country
        if(personEntity.Residience == null)
        {
            if (country == null)
            {
                country = new Country { CountryName = "" };
            }
            personEntity.Residience = country;
        }

        //books
        if(personEntity.Books == null)
        {
            personEntity.Books = new List<Book>();
        }

        if (book != null)
        {
            if (!personEntity.Books.Any(x => x.BookId == book.BookId))
            {
                personEntity.Books.Add(book);
            }
        }

        return personEntity;
},
splitOn: "CountryId,BookId");
```

Custom Mappings

If the query column names do not match your classes you can setup mappings for types. This example demonstrates mapping using `System.Data.Linq.Mapping.ColumnAttribute`as well as a custom mapping.

> The mappings only need to be setup once per type so set them on application startup or somewhere else that they are only initialized once.

Assuming the same query as the One-to-many example again and the classes refactored toward better names like so:

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Born { get; set; }
    public Country Residience { get; set; }
    public ICollection<Book> Books { get; set; }
}

public class Country
{
    [System.Data.Linq.Mapping.Column(Name = "CountryId")]
    public int Id { get; set; }

    [System.Data.Linq.Mapping.Column(Name = "CountryName")]
```

```
    public string Name { get; set; }
}

public class Book
{
    public int Id { get; set; }

    public string Name { get; set; }
}
```

> Note how `Book` doesn't rely on `ColumnAttribute` but we would need to maintain the `if` statement

Now place this mapping code somewhere in your application where it is only executed once:

```
Dapper.SqlMapper.SetTypeMap(
    typeof(Country),
    new CustomPropertyTypeMap(
        typeof(Country),
        (type, columnName) =>
            type.GetProperties().FirstOrDefault(prop =>
                prop.GetCustomAttributes(false)
                    .OfType<System.Data.Linq.Mapping.ColumnAttribute>()
                    .Any(attr => attr.Name == columnName)))
);


var bookMap = new CustomPropertyTypeMap(
    typeof(Book),
    (type, columnName) =>
    {
        if(columnName == "BookId")
        {
            return type.GetProperty("Id");
        }

        if (columnName == "BookName")
        {
            return type.GetProperty("Name");
        }

        throw new InvalidOperationException($"No matching mapping for {columnName}");
    }
);
Dapper.SqlMapper.SetTypeMap(typeof(Book), bookMap);
```

Then the query is executed using any of the previous `Query<>` examples.

A simpler way of adding the mappings is shown in this answer.

Read Multimapping online: https://riptutorial.com/dapper/topic/351/multimapping

Chapter 8: Multiple Results

Syntax

- public static SqlMapper.GridReader QueryMultiple(this IDbConnection cnn, string sql, object param = null, IDbTransaction transaction = null, int? commandTimeout = null, CommandType? commandType = null)
- public static SqlMapper.GridReader QueryMultiple(this IDbConnection cnn, CommandDefinition command)

Parameters

Parameter	Details
cnn	Your database connection, must already be open
sql	The sql string to process, contains multiple queries
param	Object to extract parameters from
SqlMapper.GridReader	Provides interfaces for reading multiple result sets from a Dapper query

Examples

Base Multiple Results Example

To fetch multiple grids in a single query, the `QueryMultiple` method is used. This then allows you to retrieve each grid *sequentially* through successive calls against the `GridReader` returned.

```
var sql = @"select * from Customers where CustomerId = @id
            select * from Orders where CustomerId = @id
            select * from Returns where CustomerId = @id";

using (var multi = connection.QueryMultiple(sql, new {id=selectedId}))
{
   var customer = multi.Read<Customer>().Single();
   var orders = multi.Read<Order>().ToList();
   var returns = multi.Read<Return>().ToList();
}
```

Read Multiple Results online: https://riptutorial.com/dapper/topic/8/multiple-results

Chapter 9: Parameter Syntax Reference

Parameters

Parameter	Details
this cnn	The underlying database connection - the this denotes an extension method; the connection does not need to be open - if it is not open, it is opened and closed automatically.
<T> / Type	(optional) The type of object to return; if the non-generic / non-Type API is used, a dynamic object is returned per row, simulating a property named per column name returned from the query (this dynamic object also implements IDicionary<string,object>).
sql	The SQL to execute
param	(optional) The parameters to include.
transaction	(optional) The database transaction to associate with the command
buffered	(optional) Whether to pre-consume the data into a list (the default), versus exposing an open IEnumerable over the live reader
commandTimeout	(optional) The timeout to use on the command; if not specified, SqlMapper.Settings.CommandTimeout is assumed (if specified)
commandType	The type of command being performed; defaults to CommandText

Remarks

The syntax for expressing parameters varies between RDBMS. All the examples above use SQL Server syntax, i.e. @foo; however, ?foo and :foo should also work fine.

Examples

Basic Parameterized SQL

Dapper makes it easy to follow best practice by way of fully parameterized SQL.

Parameters are important, so dapper makes it easy to get it right. You just express your parameters in the normal way for your RDBMS (usually `@foo`, `?foo` or `:foo`) and give dapper an object that *has a member called* `foo`. The most common way of doing this is with an anonymous type:

```
int id = 123;
string name = "abc";
connection.Execute("insert [KeyLookup](Id, Name) values(@id, @name)",
    new { id, name });
```

And... that's it. Dapper will add the required parameters and everything should work.

Using your Object Model

You can also use your existing object model as a parameter:

```
KeyLookup lookup = ... // some existing instance
connection.Execute("insert [KeyLookup](Id, Name) values(@Id, @Name)", lookup);
```

Dapper uses the command-text to determine which members of the object to add - it won't usually add unnecessary things like `Description`, `IsActive`, `CreationDate` because the command we've issued clearly doesn't involve them - although there are cases when it might do that, for example if your command contains:

```
// TODO - removed for now; include the @Description in the insert
```

It doesn't attempt to figure out that the above is just a comment.

Stored Procedures

Parameters to stored procedures work exactly the same, except that dapper cannot attempt to determine what should/should-not be included - everything available is treated as a parameter. For that reason, anonymous types are usually preferred:

```
connection.Execute("KeyLookupInsert", new { id, name },
    commandType: CommandType.StoredProcedure);
```

Value Inlining

Sometimes the convenience of a parameter (in terms of maintenance and expressiveness), may be outweighed by its cost in performance to treat it as a parameter. For example, when page size is fixed by a configuration setting. Or a status value is matched to an `enum` value. Consider:

```
var orders = connection.Query<Order>(@"
select top (@count) * -- these brackets are an oddity of SQL Server
from Orders
where CustomerId = @customerId
and Status = @open", new { customerId, count = PageSize, open = OrderStatus.Open });
```

The only *real* parameter here is `customerId` - the other two are pseudo-parameters that won't actually change. Often the RDBMS can do a better job if it detects these as constants. Dapper has a special syntax for this - `{=name}` instead of `@name` - which *only* applies to numeric types. (This minimizes any attack surface from SQL injection). An example is as follows:

```
var orders = connection.Query<Order>(@"
select top {=count} *
from Orders
where CustomerId = @customerId
and Status = {=open}", new { customerId, count = PageSize, open = OrderStatus.Open });
```

Dapper replaces values with literals before issuing the SQL, so the RDBMS actually sees something like:

```
select top 10 *
from Orders
where CustomerId = @customerId
and Status = 3
```

This is particularly useful when allowing RDBMS systems to not just make better decisions, but to open up query plans that actual parameters prevent. For example, if a column predicate is against a parameter, then a filtered index with specific values on that columns cannot be used. This is because the *next* query may have a parameter apart from one of those specified values.

With literal values, the query optimizer is able to make use of the filtered indexes since it knows the value cannot change in future queries.

List Expansions

A common scenario in database queries is `IN (...)` where the list here is generated at runtime. Most RDBMS lack a good metaphor for this - and there is no universal *cross-RDBMS* solution for this. Instead, dapper provides some gentle automatic command expansion. All that is requires is a supplied parameter value that is `IEnumerable`. A command involving `@foo` is expanded to `(@foo0,@foo1,@foo2,@foo3)` (for a sequence of 4 items). The most common usage of this would be `IN`:

```
int[] orderIds = ...
```

```
var orders = connection.Query<Order>(@"
select *
from Orders
where Id in @orderIds", new { orderIds });
```

This then automatically expands to issue appropriate SQL for the multi-row fetch:

```
select *
from Orders
where Id in (@orderIds0, @orderIds1, @orderIds2, @orderIds3)
```

with the parameters `@orderIds0` etc being added as values taken from the arrray. Note that the fact that it isn't valid SQL originally is intentional, to ensure that this feature is not used mistakenly. This feature also works correctly with the `OPTIMIZE FOR` / `UNKNOWN` query-hint in SQL Server; if you use:

```
option (optimize for
    (@orderIds unknown))
```

it will expand this correctly to:

```
option (optimize for
    (@orderIds0 unknown, @orderIds1 unknown, @orderIds2 unknown, @orderIds3 unknown))
```

Performing Operations Against Multiple Sets of Input

Sometimes, you want to do the same thing multiple times. Dapper supports this on the `Execute` method if the *outermost* parameter (which is usually a single anonymous type, or a domain model instance) is actually provided as an `IEnumerable` sequence. For example:

```
Order[] orders = ...
// update the totals
connection.Execute("update Orders set Total=@Total where Id=@Id", orders);
```

Here, dapper is just doing a simple loop on our data, essentially the same as if we had done:

```
Order[] orders = ...
// update the totals
foreach(Order order in orders) {
    connection.Execute("update Orders set Total=@Total where Id=@Id", order);
}
```

This usage becomes *particularly* interesting when combined with the `async` API on a connection that is explicitly configured to all "Multiple Active Result Sets" - in this usage, dapper will automatically *pipeline* the operations, so you aren't paying the latency cost per row. This requires a slightly more complicated usage,

```
await connection.ExecuteAsync(
    new CommandDefinition(
        "update Orders set Total=@Total where Id=@Id",
```

```
        orders, flags: CommandFlags.Pipelined))
```

Note, however, that you might also wish to investigate table valued parameters.

Pseudo-Positional Parameters (for providers that don't support named parameters)

Some ADO.NET providers (most notably: OleDB) do not support *named* parameters; parameters are instead specified only by *position*, with the `?` place-holder. Dapper would not know what member to use for these, so dapper allows an alternative syntax, `?foo?`; this would be the same as `@foo` or `:foo` in other SQL variants, except that dapper will **replace** the parameter token completely with `?` before executing the query.

This works in combination with other features such as list expansion, so the following is valid:

```
string region = "North";
int[] users = ...
var docs = conn.Query<Document>(@"
     select * from Documents
     where Region = ?region?
     and OwnerId in ?users?", new { region, users }).AsList();
```

The `.region` and `.users` members are used accordingly, and the SQL issued is (for example, with 3 users):

```
     select * from Documents
     where Region = ?
     and OwnerId in (?,?,?)
```

Note, however, that dapper **does not** allow the same parameter to be used multiple times when using this feature; this is to prevent having to add the same parameter value (which could be large) multiple times. If you need to refer to the same value multiple times, consider declaring a variable, for example:

```
declare @id int = ?id?; // now we can use @id multiple times in the SQL
```

If variables are not available, you can use duplicate member names in the parameters - this will also make it obvious that the value is being sent multiple times:

```
int id = 42;
connection.Execute("... where ParentId = $id0$ ... SomethingElse = $id1$ ...",
      new { id0 = id, id1 = id });
```

Read Parameter Syntax Reference online: https://riptutorial.com/dapper/topic/10/parameter-syntax-reference

Chapter 10: Temp Tables

Examples

Temp Table that exists while the connection remains open

When the temp table is created by itself, it will remain while the connection is open.

```
// Widget has WidgetId, Name, and Quantity properties
public async Task PurchaseWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        await conn.OpenAsync();

        await conn.ExecuteAsync("CREATE TABLE #tmpWidget(WidgetId int, Quantity int)");

        // populate the temp table
        using(var bulkCopy = new SqlBulkCopy(conn)) {
            bulkCopy.BulkCopyTimeout = SqlTimeoutSeconds;
            bulkCopy.BatchSize = 500;
            bulkCopy.DestinationTableName = "#tmpWidget";
            bulkCopy.EnableStreaming = true;

            using(var dataReader = widgets.ToDataReader())
            {
                await bulkCopy.WriteToServerAsync(dataReader);
            }
        }

        await conn.ExecuteAsync(@"
            update w
            set Quantity = w.Quantity - tw.Quantity
            from Widgets w
                join #tmpWidget tw on w.WidgetId = tw.WidgetId");
    }
}
```

How to work with temp tables

The point about temporary tables is that they're limited to the scope of the connection. Dapper will automatically open and close a connection if it's not already opened. That means that any temp table will be lost directly after creating it, if the connection passed to Dapper has not been opened.

This will not work:

```
private async Task<IEnumerable<int>> SelectWidgetsError()
{
  using (var conn = new SqlConnection(connectionString))
  {
    await conn.ExecuteAsync(@"CREATE TABLE #tmpWidget(widgetId int);");

    // this will throw an error because the #tmpWidget table no longer exists
    await conn.ExecuteAsync(@"insert into #tmpWidget(WidgetId) VALUES (1);");
```

```
    return await conn.QueryAsync<int>(@"SELECT * FROM #tmpWidget;");
  }
}
```

On the other hand, these two versions will work:

```
private async Task<IEnumerable<int>> SelectWidgets()
{
  using (var conn = new SqlConnection(connectionString))
  {
    // Here, everything is done in one statement, therefore the temp table
    // always stays within the scope of the connection
    return await conn.QueryAsync<int>(
      @"CREATE TABLE #tmpWidget(widgetId int);
        insert into #tmpWidget(WidgetId) VALUES (1);
        SELECT * FROM #tmpWidget;");
  }
}

private async Task<IEnumerable<int>> SelectWidgetsII()
{
  using (var conn = new SqlConnection(connectionString))
  {
    // Here, everything is done in separate statements. To not loose the
    // connection scope, we have to explicitly open it
    await conn.OpenAsync();

    await conn.ExecuteAsync(@"CREATE TABLE #tmpWidget(widgetId int);");
    await conn.ExecuteAsync(@"insert into #tmpWidget(WidgetId) VALUES (1);");
    return await conn.QueryAsync<int>(@"SELECT * FROM #tmpWidget;");
  }
}
```

Read Temp Tables online: https://riptutorial.com/dapper/topic/6594/temp-tables

Chapter 11: Transactions

Syntax

- conn.Execute(sql, transaction: tran); // specify the parameter by name
- conn.Execute(sql, parameters, tran);
- conn.Query(sql, transaction: tran);
- conn.Query(sql, parameters, tran);
- await conn.ExecuteAsync(sql, transaction: tran); // Async
- await conn.ExecuteAsync(sql, parameters, tran);
- await conn.QueryAsync(sql, transaction: tran);
- await conn.QueryAsync(sql, parameters, tran);

Examples

Using a Transaction

This example uses SqlConnection, but any IDbConnection is supported.

Also any IDbTransaction is supported from the related IDbConnection.

```
public void UpdateWidgetQuantity(int widgetId, int quantity)
{
    using(var conn = new SqlConnection("{connection string}")) {
        conn.Open();

        // create the transaction
        // You could use `var` instead of `SqlTransaction`
        using(SqlTransaction tran = conn.BeginTransaction()) {
            try
            {
                var sql = "update Widget set Quantity = @quantity where WidgetId = @id";
                var parameters = new { id = widgetId, quantity };

                // pass the transaction along to the Query, Execute, or the related Async
methods.
                conn.Execute(sql, parameters, tran);

                // if it was successful, commit the transaction
                tran.Commit();
            }
            catch(Exception ex)
            {
                // roll the transaction back
                tran.Rollback();

                // handle the error however you need to.
                throw;
            }
        }
    }
}
```

Speed up inserts

Wrapping a group of inserts in a transaction will speed them up according to this StackOverflow Question/Answer.

You can use this technique, or you can use Bulk Copy to speed up a series of related operations to perform.

```
// Widget has WidgetId, Name, and Quantity properties
public void InsertWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        conn.Open();

        using(var tran = conn.BeginTransaction()) {
            try
            {
                var sql = "insert Widget (WidgetId,Name,Quantity) Values(@WidgetId, @Name,
@Quantity)";
                conn.Execute(sql, widgets, tran);
                tran.Commit();
            }
            catch(Exception ex)
            {
                tran.Rollback();
                // handle the error however you need to.
                throw;
            }
        }
    }
}
```

Read Transactions online: https://riptutorial.com/dapper/topic/6601/transactions

Chapter 12: Type Handlers

Remarks

Type Handlers allow database types to be converted to .Net custom types.

Examples

Converting varchar to IHtmlString

```
public class IHtmlStringTypeHandler : SqlMapper.TypeHandler<IHtmlString>
{
    public override void SetValue(
        IDbDataParameter parameter,
        IHtmlString value)
    {
        parameter.DbType = DbType.String;
        parameter.Value = value?.ToHtmlString();
    }

    public override IHtmlString Parse(object value)
    {
        return MvcHtmlString.Create(value?.ToString());
    }
}
```

Installing a TypeHandler

The above type handler can be installed into `SqlMapper` using the `AddTypeHandler` method.

```
SqlMapper.AddTypeHandler<IHtmlString>(new IHtmlStringTypeHandler());
```

Type inference allows you to omit the generic type parameter:

```
SqlMapper.AddTypeHandler(new IHtmlStringTypeHandler());
```

There's also a two-argument overload which takes an explicit `Type` argument:

```
SqlMapper.AddTypeHandler(typeof(IHtmlString), new IHtmlStringTypeHandler());
```

Read Type Handlers online: https://riptutorial.com/dapper/topic/6/type-handlers

Chapter 13: Using Async

Examples

Calling a Stored Procedure

```
public async Task<Product> GetProductAsync(string productId)
{
    using (_db)
    {
        return await _db.QueryFirstOrDefaultAsync<Product>("usp_GetProduct", new { id =
productId },
            commandType: CommandType.StoredProcedure);
    }
}
```

Calling a stored procedure and ignoring the result

```
public async Task SetProductInactiveAsync(int productId)
{
    using (IDbConnection con = new SqlConnection("myConnectionString"))
    {
        await con.ExecuteAsync("SetProductInactive", new { id = productId },
            commandType: CommandType.StoredProcedure);
    }
}
```

Read Using Async online: https://riptutorial.com/dapper/topic/1353/using-async

Chapter 14: Using DbGeography and DbGeometry

Examples

Configuration required

1. install the required `Microsoft.SqlServer.Types` assembly; they are not installed by default, and are available from Microsoft here as "Microsoft® System CLR Types for Microsoft® SQL Server® 2012" - note that there are separate installers for x86 and x64.

2. install `Dapper.EntityFramework` (or the strong-named equivalent); this could be done via the IDE's "Manage NuGet Packages..." UI, or (at the Package Manager Console):

   ```
   install-package Dapper.EntityFramework
   ```

3. add the required assembly binding redirects; this is because Microsoft ships v11 of the assemblies, but Entity Framework asks for v10; you can add the following to `app.config` or `web.config` under the `<configuration>` element:

   ```
   <runtime>
     <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
       <dependentAssembly>
         <assemblyIdentity name="Microsoft.SqlServer.Types"
             publicKeyToken="89845dcd8080cc91" />
         <bindingRedirect oldVersion="10.0.0.0" newVersion="11.0.0.0" />
       </dependentAssembly>
     </assemblyBinding>
   </runtime>
   ```

4. tell "dapper" about the new type handlers available, by adding (somewhere in your startup, before it tries using the database):

   ```
   Dapper.EntityFramework.Handlers.Register();
   ```

Using geometry and geography

Once the type handlers are registered, everything should work automatically, and you should be able to use these types as either parameters or return values:

```
string redmond = "POINT (122.1215 47.6740)";
DbGeography point = DbGeography.PointFromText(redmond,
    DbGeography.DefaultCoordinateSystemId);
DbGeography orig = point.Buffer(20); // create a circle around a point


var fromDb = connection.QuerySingle<DbGeography>(
```

```
  "declare @geos table(geo geography); insert @geos(geo) values(@val); select * from @geos",
  new { val = orig });

Console.WriteLine($"Original area: {orig.Area}");
Console.WriteLine($"From DB area: {fromDb.Area}");
```

Read Using DbGeography and DbGeometry online: https://riptutorial.com/dapper/topic/3984/using-dbgeography-and-dbgeometry

Credits

S. No	Chapters	Contributors
1	Getting started with Dapper.NET	Adam Lear, balpha, Community, Eliza, Greg Bray, Jarrod Dixon, Kevin Montrose, Matt McCabe, Nick, Rob, Shog9
2	Basic Querying	Adam Lear, Chris Marisic, Cigano Morrison Mendez, Community, cubrr, Jarrod Dixon, jrummell, Kevin Montrose, Matt McCabe
3	Bulk inserts	jhamm
4	Dynamic Parameters	Marc Gravell, Matt McCabe, Meer
5	Executing Commands	Adam Lear, Jarrod Dixon, Sklivvz, takrl
6	Handling Nulls	Marc Gravell
7	Multimapping	Devon Burriss
8	Multiple Results	Marc Gravell, Yaakov Ellis
9	Parameter Syntax Reference	4444, Marc Gravell, Nick Craver
10	Temp Tables	jhamm, Rob, takrl
11	Transactions	jhamm
12	Type Handlers	Benjamin Hodgson, Community, Marc Gravell
13	Using Async	Dean Ward, Matt McCabe, Nick, Woodchipper
14	Using DbGeography and DbGeometry	Marc Gravell